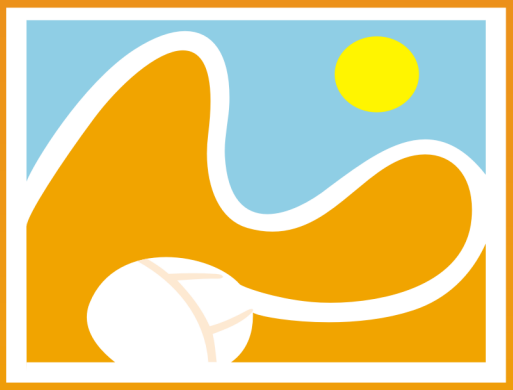
SugarLoafPLoP 2003 Proceedings

The Third Latin American Conference on Pattern Languages of Programming

SugarLoafPLoP '2003



August 12-15-2003
Porto de Galinhas
Pernambuco, Brazil

http://www.cin.ufpe.br/~sugarloafplop

editors

Robert Hanmer
Rossana M. C. Andrade

organized by

CIn / UFPE
UFC
Lucent Technologies

Sponsored by the Hillside Group
Supported by SBC (Brazilian Computer Society)

SugarLoafPLoP 2003 Proceedings

The Third Latin American Conference on Pattern Languages of Programming

SugarLoafPLoP '2003

August 12-15-2003
Porto de Galinhas
Pernambuco, Brazil

http://www.cin.ufpe.br/~sugarloafplop

editors

Robert Hanmer
Rossana M. C. Andrade

organized by

CIn / UFPE
UFC
Lucent Technologies

Sponsored by the Hillside Group
Supported by SBC (Brazilian Computer Society)

Publisher: CIn / UFPE

Editors: Rossana M. C. Andrade and Robert Hanmer

Latin American Conference on Pattern Languages of
Programming SugarLoafPLoP 2003 (3.: 2003 : Porto de Galinhas, PE)
Proceedings... / Editors Rossana M. C. Andrade
Robert Hanmer. -- Recife, PE: ICMC/USP, 2003.
330 p.
ISBN 85-87837-08-7

1. Padrões de Software. 2. Linguagens de Padrões. I. Andrade, Rossana M. C., ed. II. Robert Hanmer, ed. III. Título

Program Comittee

Conference Co-Chairs

Paulo Borba (CIn/UFPE, BR)
Sérgio Soares (CIn/UFPE, BR)

Program Co-Chairs

Rossana Andrade (DC/UFC, BR)
Robert Hanmer (Lucent Technologies, US)

Local Organization

Ayla Souza (CIn/UFPE, BR)
Rohit Gheyi (CIn/UFPE, BR)

Shepherds

Ed Fernandez - Florida Atlantic University, USA
Federico Balaguer - University of Illinois at Urbana-Champaign, USA
Jerffeson Teixeira de Souza - University of Ottawa, Canada
John P. Letourneau - Lucent Technologies - Bell Labs, USA
Jorge Ortega Arjona - University College London, UK
Joseph W. Yoder - University of Illinois/The Refactory, Inc., USA
Márcio de Oliveira Barros - COPPE/Universidade Federal do Rio de Janeiro, Brasil
Neil B. Harrison - Avaya Labs, USA
Paulo Cesar Masiero - ICMC/Universidade de São Paulo, Brasil
Robert Hanmer - Lucent Technologies - Bell Labs, USA
Rosana Teresinha Vaccare Braga - ICMC/Universidade de São Paulo, Brasil
Rossana Maria de Castro Andrade - DC/Universidade Federal do Ceará, Brasil
Sérgio Castelo Branco Soares - CIn/Universidade Federal de Pernambuco, Brasil
Tiago Lima Massoni - CIn/Universidade Federal de Pernambuco, Brasil
Vander Ramos Alves - CIn/Universidade Federal de Pernambuco, Brasil

Referees

Adenilso da Silva Simão - ICMC/Universidade de São Paulo
Carlo Giovano S. Pires - Instituto Atlantico, Brasil
Claudia Maria Lima Werner - COPPE/Universidade Federal do Rio de Janeiro, Brasil
Fabiana G. Marinho - Instituto Atlantico, Brasil
Fernando de Carvalho Gomes - DC/Universidade Federal do Ceará, Brasil
Fernão Stella de Rodrigues Germano - ICMC/Universidade de São Paulo, Brasil
Maria Istela Cagnin - ICMC/Universidade de São Paulo, Brasil
Paulo Henrique Monteiro Borba - CIn/Universidade Federal de Pernambuco, Brasil
Paulo Cesar Masiero - ICMC/Universidade de São Paulo, Brasil
Rosana Teresinha Vaccare Braga - ICMC/Universidade de São Paulo, Brasil
Rosângela Dellosso Penteado DC/Universidade Federal de São Carlos, Brasil

Table of Contents

Foreword

Software developers have all experienced and read about software reuse. The emerging interest in software patterns represents an effort to catalog and communicate recurring and enduring themes across different applications and systems. In this context, we are glad to present the third edition of the proceedings of the Latin American Conference on Pattern Languages of Programming (SugarloafPLoP 2003). These proceedings provide a catalog of proven solutions to common problems in a broad range of applications and systems such as distributed component development, adaptive applications, telecommunications, computer networks, parallel computing, aspect-oriented programming, software agents, information systems, reverse engineering and legacy systems. Besides this, we include the final versions of the articles presented in the conference special session called "Software Pattern Applications – SPA" that involves applications of patterns in industry and academia. There is another special session in the conference called Writing Patterns (WP) that receives newcomers who want to learn how to better elaborate an idea intended to evolve to a pattern. This year, we chose one WP article to be published in the final proceedings.

SugarLoafPLoP 2003 brought together researchers, educators and practitioners whose interests span a remarkably broad range of topics as mentioned before and who share an interest in exploring the power of the pattern form. The conference allowed articles written in English, Portuguese and Spanish and we also maintain this three-lingual venue in the proceedings. We hope these articles will give readers a glimpse of the current state of the software pattern capture and application and a sense of where this area is taking us.

In the conference, the accepted papers for the writers´ workshop were split in four groups. Group A includes three articles in English and one in Spanish. The first article of group A, "Process Patterns for the Distributed Component Development" by Alexandre Alvaro et al, describes a sequence of steps for the Distributed Component Development integrating different known principles to support the process. The second article, "Switch Strategy Pattern" by Luis César Maiarú, introduces a pattern that allows to select different implementation of the same interface, depending on certain condition, without hard-coding a switch statement or a sequence of conditional statements. The third article, "GIG-Pattern" by Maria Lencastre et al, proposes a pattern to represent a generic interface graph that deals with process definition and control. The last article of group A, "AdapPE: An Architectural Pattern for Structuring Adaptive Applications with Aspects" by Ayla Dantas and Paulo Borba, presents an architectural pattern for structuring adaptive applications using aspect-oriented programming in order to obtain separation of concerns.

Group B includes one paper in Spanish and three in English, as follows: "Un Patrón para lexicones de patrones" by Alan Calderón Castro, describes a pattern aimed at building systems to support pattern language evolution; "A Pattern System to Supervisory Control of

Automated Manufacturing System" by Paulo Cézar Stadzisz et al, proposes a system of patterns, which is composed by an architectural pattern and three design patterns, to be applied in SC-AMS domain; "Operations and Maintenance 2" by Robert Hanmer, introduces patterns, which are part of a larger collection of patterns, that describe the architecture of a telephone switching system; and "A Parallel Algorithm Pattern" by Marcos Cordeiro d´Ornellas, describes a parallel pattern in terms of image scannings and its relationship within mathematical morphology.

Group C and Group D contain articles in Portuguese. Group C includes the following four articles: "Padrões de Reengenharia Auxiliados por Diretrizes de Qualidade de Software" by Gizelle Sandrini Lemos et al describes using seven pattern clusters the Object-oriented Reenginnering Process (PRE/OO) for procedural legacy systems; "Padrões para o Processo de Engenharia Avante na Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais" by Edson Luiz Recchia presents step by step forward engineering process patterns, illustrating their use; "Multi Locale Entity: Um padrão de projeto para persistência de entidades de internacionalização" by Carlo Giovano S. Pires introduces a design pattern to support internationalization and location; and "TB-REPP - Padrões de Processo para a Engenharia Reversa baseado em Transformações" by Darley Rosa Peres et al proposes patterns to obtain the system project in a high abstraction level, assuring its evolution and maintenance, making it more expressive and easy to understand.

Group D gets the following three articles: "PATI-MVC: Padrões MVC para Sistemas de Informação" by Gabriela T. de Souza et al presents recurrent design problems in information systems as patterns that deal with user interface, business control and business entities; "Padrões Arquiteturais e de Projeto para a Modelagem de Usuários baseada em Agentes" by Ismênia Ribeiro de Oliveira and Rosario Girardi proposes solutions for frequent user modeling problems in the form of agent-based architectural and detailed software patterns; and "Um Padrão para Gerenciamento Inteligente de Redes" by Calebe de Paula Bianchini et al presents a pattern for developing intelligent software agents that help in the network monitoring.

The following articles were presented in the conference during the SPA special session: "MVCASE Tool – Working with Design Patterns" by Daniel Lucrédio et al presents an extension of MVCASE tool to help software engineers to create and reuse design patterns in a high level of abstraction during the software development process; "Utilização do design pattern Architecture Configurator em um ambiente de suporte para Configuração de Arquiteturas" by Jonivan Coutinho Lisbôa and Orlando Gomes Loques Filho shows how the design pattern Architecture Configurator provides a sound basis for conception and implementation of support environments for the configuration of software architectures; and "Modelado Estructural de Patrones de Diseño: Diagramas REP" by José Luis Isla Montes and Francisco Luis Gutiérrez Vela proposes a simple and intuitive model for the structural specification of design patterns and its integration with UML.

The last article in the proceedings was presented in the WP special session and it is entitled "Observer Pattern using Aspect -Oriented Programming" by Eduardo Kessler Piveta. It discusses the representation and implementation of the Observer design pattern using aspect-oriented techniques.

Bringing together this set of articles and going through the various steps prior to the publication that include the shepherding process and the writers´ workshop requires the dependable support of many people. We thank them all. We specially thank the shepherds and the reviewers who provided valuable feedback in a timely fashion. We also thank the authors who promptly responded to the requests made to revise their articles. We could not forget to thank the organization committee, Paulo Borba, Ayla Souza, Rohit Gheyi and Sérgio Soares for their excellent and hard work for the organization of the conference. Finally, we would like to thank the cooperation of the Brazilian Computer Society (SBC), CIn-UFPE, DC-UFC and Lucent Technologies and the sponsorship of the Hillside Group FACEPE, and Microsoft.

We hope you will enjoy these proceedings and be inspired to submit an article yourself in the next conference to be held in Fortaleza, Ceará.

Please provide us your feedback and comments for future issues of the proceedings. Our contact information can be found below.

Rossana Andrade (rossana@ufc.br)
Robert Hanmer (hanmer@lucent.com)
Program co-chairs

December, 2003

Writers' Workshops

Each workshop session was about 1 hour and 15 minutes in length. There was a special 30 minute session at the beginning of the conference so that each workshop group could be introduced and work out logistics, such as how much they wanted non authors to participate and in which order the papers would be presented.

The Workshop Process

The writers workshop format has proven to be useful in past PLoP conferences, so it was followed by each group. Although this format may seem unfamiliar, it has shown to be useful for developing an environment where patterns authors can share their ideas.

Introduction/Reading: Moderator introduces the Author and the Author reads a selection from the paper. This is the last we hear of the authors til the end. (Allow 5 minutes).

Summary: One of workshop participants summarizes the paper. (Allow 5 minutes)

Positive Feedback: Moderator asks for things people liked about the patterns. The comments can be about presentation or content, and at the discretion of the moderator comments about presentation and content can be intermingled, or done separately. (Allow 15 minutes)

Constructive Criticism: Moderator asks for ways in which the paper can be improved, both in content and presentation. (Allow 20 – 40 minutes)

Positive Closure: Moderator asks participants for a final closure, in which they reinforce the positive aspects of the pattern. (Allow 5 minutes)

Author Feedback: The author asks for clarification on comments made during the session. The Author should pick a few of the most important points (or ones which were made by the most people.) Further clarification can be had during off line discussions. (Allow 10 minutes)

Closing: The workshop participants thank the author.

Process Patterns for the Distributed Component Development

Alexandre Alvaro [1]
Daniel Lucrédio [1]
Eduardo Santana de Almeida [2]
Antonio Francisco do Prado [1]
Luis Carlos Trevelin [1]

[1] Computing Departament – Federal University of São Carlos
Rod. Washington Luiz, km 235 – São Carlos/SP - Brasil
P.O box 676 – Zip.Code 13565-905
Phone/Fax: + 55-16-260-8232
{aalvaro, lucredio, prado, trevelin}@dc.ufscar.br

[2] Informatics Center - Federal University of Pernambuco
Recife Center for Advanced Studies and Systems
Av. Professor Luiz Freire - Recife/PE - Brasil
University City - Zip. Code: 50740-540
Phone: + 55-81-3271-8430
esa2@cin.ufpe.br

Abstract

The proposed patterns presented in this paper describe a sequence of steps for the Distributed Component Development integrating different known principles to support the process. The involved principles are: part of Catalysis method used as a Component-Based Development (CBD) method to define, specify and design the distributed components; the middleware to support components distribution and accessing; a framework to facilitate the database access; and a CASE tool used to facilitate the patterns application.

1 Introduction

One of the most compelling reasons for adopting component-based approaches to software development, with or without objects, is the premise of reuse. The idea is to build software from existing components primarily by assembling and replacing interoperable parts. The implications for reduced development time and improved product quality make this approach very attractive [1].

Software Patterns provide a high reuse degree of software architecture and design. Using this, the system becomes more comprehensible, flexible, easy to develop and to maintain. Another objective of the software patterns is the spread of already experienced software developing solutions.

Considering the accelerated growth of the Internet over the last decade, where distribution has become an essential non-functional requirement of most applications, the problem becomes bigger.

In this context, motivated by ideas of reuse, component-based development and distribution, this paper proposes the evolution of the Distributed Component Development Pattern (DCDP) presented in [6], refining it into process patterns for the Distribution Component Development, using different known principles, which are, Catalysis as a CBD

method, a middleware to accomplish the components distribution, a framework for database access, and a CASE tool to partially automate this process. The proposed process patterns differ from the previous one as they present solutions for the different phases of the development process [6]. Another difference is that now the requirements are treated in a separated way, as will be seen later in this paper. Finally, these new patterns were applied, producing some preliminary results, as can be seen in the *Known Uses* item of the patterns descriptions.

The pattern catalog in Table 1 outlines the patterns discussed in this paper. It lists each pattern's name along with a short description of their function.

Pattern Name	Description
Define Problem	Divides the problem domain in smaller pieces for a better understanding.
Specify Components	Provides a internal specification and relationships between the components.
Design Components	Fulfilling the requirements.
Implement Components	Coding the components, based in all the documentation produced.

Table 1. A Process Patterns Catalog.

2 Define Problem

2.1 Motivation:

Consider an initial phase of software development, when you don't know the problem to solve and the totality of the domain. The emphasis must be placed on understanding the problem and specifying what the components must deal with. In other words, the requirements of the domain must be understood in order to take the appropriate decisions in directions of the components development.

2.2 Problem:

To get a good understanding of the problem domain is the main difficulty of every software development.

2.3 Forces:

- With a good definition of the problem, the developers have an easy understanding of the problem domain and, consequentially, the software can be developed faster;
- A good understanding of the problem is crucial for the consistence of the development;
- Communication between developers and customers is crucial to the success of a system, but there is a natural distancing and mistrust between customers and developers;
- Without a good definition phase, there is the risk to create a solution that solves the wrong problem. This could prejudice the whole project, requiring great effort to correct this;

- Storyboards or Mind-Maps [1] aid the problem understanding, since they offer an easy way to identify the main elements and operations involved; and
- Use Case Models are modeling techniques used to supply a clean description and consistent what the system must do, such that the use cases model can be used along the process development to document the system requirements and to serve as base for the project modeling.

2.4 Solution:

Initially, the requirements of the domain are identified, using techniques such as storyboards or mind-maps, aiming to represent the different situations and problem domain sceneries. Next, the identified requirements are specified in Collaboration Models [1, 7], representing the action collections and the participant objects. Finally, the collaboration models are refined in Use Cases Model [1, 7].

This pattern is summarized in Figure 1, where a mind-map defined in the Service Order domain requirements identification is specified in a Collaboration Model and later refined and partitioned in a Use Cases Model, aiming to reduce the complexity and improve the problem domain understanding.

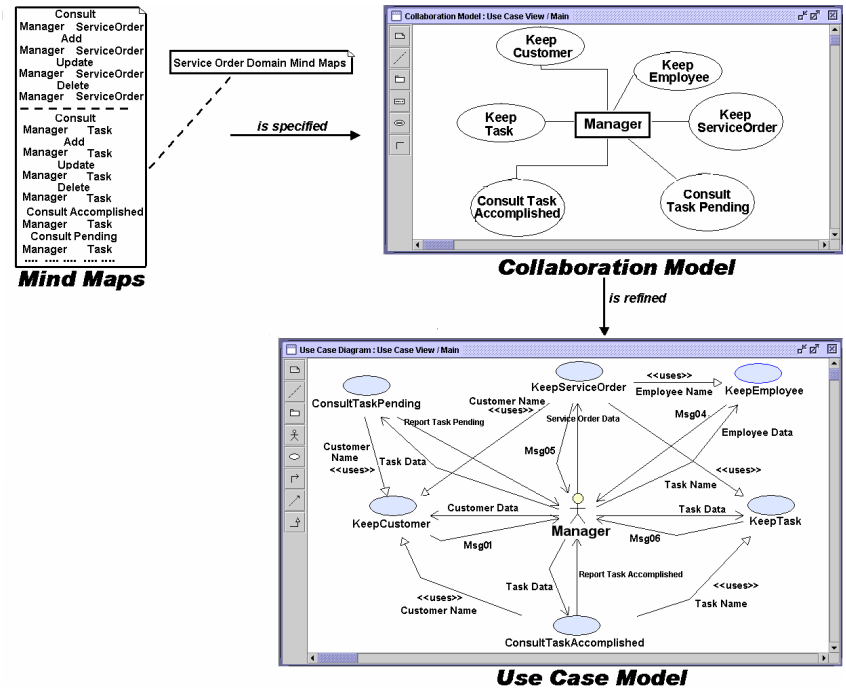


Figure 1. Models generated from Define Problem pattern.

2.5 Consequences:

- ✓ *Identification and definition of the problem:* The Define Problem pattern helps the software engineer to understand the problem and display it in a way that can be clearly understood.
- ✓ *Production of models:* Several models are generated to aid the software engineer in the understanding and documentation of the domain of the problem domain.
- ✗ *Extra Work:* The production of models cause an extra work by Software Engineering.

2.6 Related or Interacting Patterns:

- This pattern is a refinement of the first step from Distributed Component Development Pattern (DCDP), proposed in a previous work [6].

- It should be used together with the next pattern presented in this paper, **Specify Components**, producing models that should be used as an input to this pattern.
- This pattern can be seen as part of the domain analysis activity.

2.7 Known Uses:

- The Laboratory of Software Engineering in Federal University of São Carlos (UFSCar) uses this pattern in their projects.
- This pattern was used in defining the problem of a Cars Rental Company project domain, that was developed in the Computing Department of Federal University of São Carlos (UFSCar). This project generated 14 models, 16 classes and 10 components implemented.

3 Specify Components

3.1 Motivation:

The development of component-based systems considerably increases the reusability degree. However, in order to correctly design and implement the components, it is necessary to first specify the components and their roles.

3.2 Problem:

How many components are needed, what behaviors are assigned to each one and how do they relate to each other? These questions must be answered before starting to build the components.

3.3 Forces:

- Without previously planning the components, clearly defining their roles and responsibilities, there is the risk of an erroneous design. This would certainly damage the development, causing for example the reduction of the degree of reusability of a certain component.
- The impact of identifying conceptual problems and obstacles in the later phases is larger, causing extra costs to correct these problems;
- When translating the problem domain definitions, which are problem-oriented, into solution-oriented specifications, there is a possibility of misunderstandings between developers;
- Model Frameworks are templates that can be imported into some applications design. This model makes the specifications and their modeling reusable. The more generic this model is, the more it can be reused in other applications; and
- The Framework Application Model shows which types from the Model Framework are used in the system is being constructed.

3.4 Solution:

Successively refine the problem definitions, through use case models, type models, and interaction models, obtaining detailed definitions that clearly specifies “what” the components must do in order to solve the problem.

Initially, the software engineer identifies the main types of the problem domain. The use cases models produced in the **Define Problem** pattern are used in this phase. Next,

the Model of Types is specified, according to Figure 2, showing attributes and object's type operations, without worrying about implementation. Still in this step, the data dictionary can be used to specify each identified type, and the Object Constraint Language (OCL) [1] to detail the objects behavior, with no ambiguity.

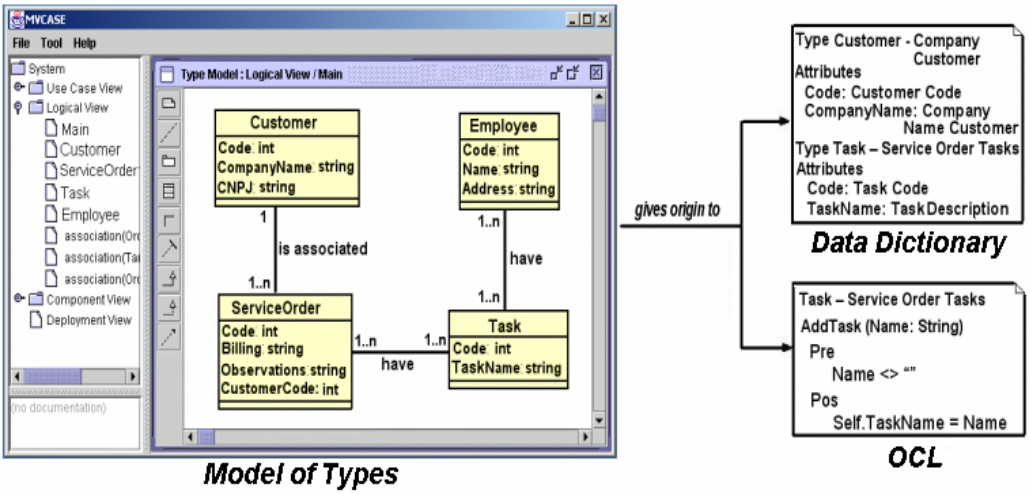


Figure 2. Model of Types from Specify Components pattern.

Once identified and specified, the types are put together in Model Frameworks. Model Frameworks are designed at a higher level of abstraction establishing a generic scheme that can be imported, at the design level, with substitutions and extensions in order to generate specific applications [1]. Figure 3 shows this model. The fact that the Model Framework is small, thus narrowly focused, increases its reuse potential in a well-defined application domain, the Service Order domain in this case. In addition, conceived as a Model Framework, it is a reusable asset at the design level, thus it is intended to be customizable to more specific applications down to the code component level [1]. As a design represents much of the major decisions that go into finished code, it can specify frameworks at a design level and offer a process to refine these frameworks down to the level of a set of interoperable code components.

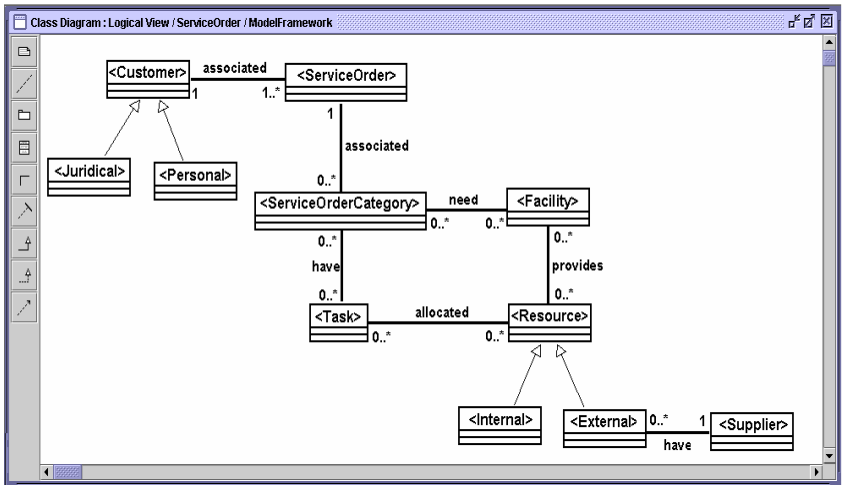


Figure 3. Service Order Model Framework.

The types with names written between brackets are defined as placeholders [1]. These types can be substituted in the specific application. The concept is similar to the extensibility of classes of the object-oriented paradigm. The framework for Service Order can be reused in several of the application's domains. Figure 4 shows the Framework Application of Service Order domain. In this framework, the types with placeholders are substituted by respective types.

Besides, the Use Case Models are refined through Interaction Models represented by sequence diagrams [7] to detail the utility scenarios of components in different applications of the problem domain.

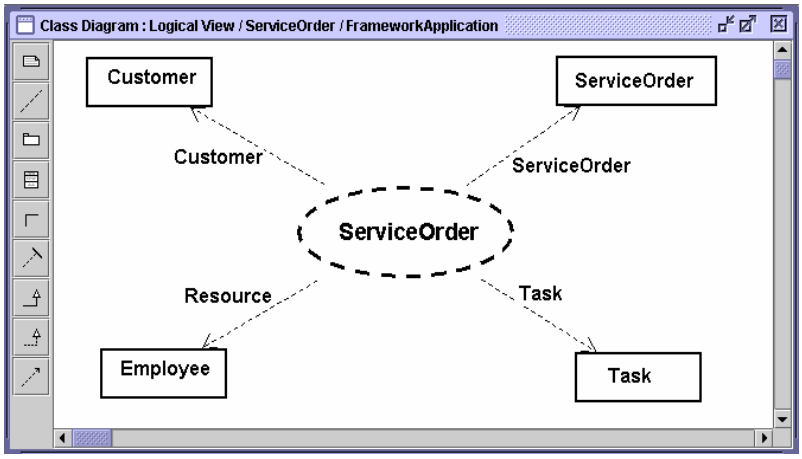


Figure 4. Service Order Framework Application.

In summary, the activities from this pattern, accomplished by the software engineer in the CASE tool [12,13] include the specifications of:

a) Model of Types;
b) Model Framework;
c) Framework Application;
d) Interactions Models, represented by sequence diagrams, based on Use Cases Model.

3.5 Consequences:

✓ *Testability:* The clear, non-ambiguous specification of the components, produced when using this pattern, can be used as a basis for later testing their behavior.

✓ *Production of models*: Several models are generated to facilitate the software engineer in the understanding and documentation of the internal characteristics and behavior of the components, and their interrelations with the other components of the problem domain.

✖ *Difficulty of generalization*: In this pattern a Model Framework is generated. This model attempts to generalize the Model of Types, aiming the reuse in a high abstraction level. However, it is difficult to generalize because the software engineering doesn't have mechanisms to help it and so this model is generalized through the experiences of projects accomplished already.

3.6 Related or Interacting Patterns:

- This pattern is a refinement of second step of DCDP, proposed in a previous work [6];
- It should be used together with the previous pattern presented in this paper, **Define Problem**; and
- It produces deliverables that should be used in the next pattern, **Design Components**.

3.7 Known Uses:

- The Laboratory of Software Engineering in Federal University of São Carlos (UFSCar) uses this pattern in their projects.
- This pattern was used in defining the problem of a On-Line Bookstore domain, that was developed in the Computing Department of Federal University of São Carlos (UFSCar). This project generated 17 models, 21 classes and 20 components implemented.

4 Design Components

4.1 Motivation:

When designing components, functional and non-functional requirements must be taken into account. In order to fulfill the functional requirements, the software engineer must define how the components will perform the behavior that was assigned to them. The non-functional requirements, such as distributed architecture, fault tolerance, caching, persistence and load balancing, must also be specified in order to complete the component's functionality.

4.2 Problem:

The software engineer must design the components aiming to fulfill the different requirements. However, it is hard to work with all of them at the same time, since different issues and problems may arise together. The main problem is that the issues related to one requirement may interfere with issues from another requirement, causing a confusion when designing the component.

4.3 Forces:

- Separation of the requirements to isolate each one's concerns makes component development easier;
- Well-defined design models can considerably facilitate the subsequent implementation tasks; and
- *Distributed Adapters Pattern* (DAP) [15] is used to separation of concerns, minimizing then, the impact on business code. It's turning the components independent from a communication API;

4.4 Solution:

The main issue here is "how" the components solve the problem. This is achieved by specifying the functional and non-functional requirements. By using this pattern, this is performed in an incremental way, one requirement at a time. First, the functional requirements are considered, followed by the non-functional requirements (e.g. distribution, persistence and fault tolerance).

In order to deal firstly with the functional requirements, the Classes Models are created, where the classes are modeled with their relationships, taking into consideration the components definitions and their interfaces. Interaction models showing details of the methods behavior are also modeled. The models produced by the **Specify Components** pattern, are used when creating the models in this pattern. Figure 5 shows a portion of the Classes Model of Service Order domain.

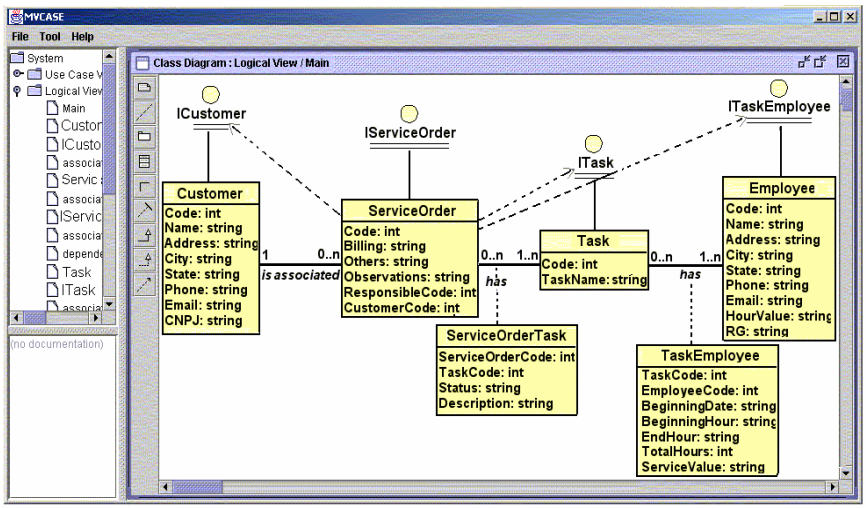


Figure 5. Classes Model obtained from Model of Types.

Next, the non-functional requirements are considered. Starting from *Classes Model*, the *Distributed Adapters Pattern*(DAP), which is a pattern for isolating distribution characteristics from the business rules, is applied to design *Components Models* [7], where the organizations and dependencies between components are shown. The next section presents the application of this pattern.

4.4.1 Applying DAP:

Figure 6 shows the designed Components Model after the application of DAP. The components Source and Target abstract the business rules of the problem domain. The TargetInterface interface abstracts the Target component behavior in distributed scenery. At this interface, the components Source and Target do not have communication code either. These three elements compose a distributed independent layer.

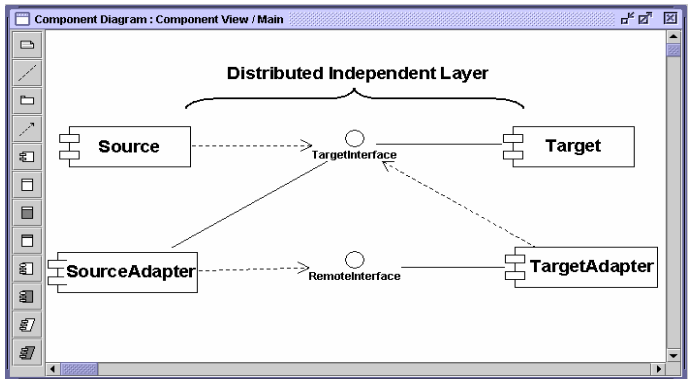


Figure 6. Design Component Model after apply DAP.

The main components are SourceAdapter and TargetAdapter. They are connected to a specific API of distribution and encapsulate the communication details. SourceAdapter is an adapter that isolates the Source component from distributed code. It is located in the same machine that Source and works as a proxy to TargetAdapter. TargetAdapter is located in another machine, isolating the Target component from distributed code. SourceAdapter and TargetAdapter, usually, are located in different machines, and do not directly interact. TargetAdapter implements RemoteInterface used to connect with SourceAdapter.

The presented adapters deal with basic distribution details and hide these details from the business and the user interface code. The adapters may also handle additional non-functional behavior, which also should not affect the business and the user interface code. In this step, we illustrate how the adapters may perform some of this additional behavior, which might be useful for implementing distributed applications.

i. Fault Tolerance. The source adapters presented previously have no fault tolerant behavior. If there is a communication error or if the server is unavailable, they simply raise a communication exception. Nevertheless, source adapters can also implement fault tolerant behavior [2].

If a source adapter receives a remote exception when interacting with the target adapter, it may implement the policy of trying to contact the target adapter again a certain number of times, or trying to contact another target adapter, representing a spare service. This policy, being implemented by the source adapter, is hidden from its client, a GUI for instance [6].

ii. Caching. Some operations may return a considerable amount of data, of which only part is useful at any moment. Sending everything to the client at once is not desirable since it may have a negative impact on network performance. One solution is to send a cache with part of the required data and to transfer more data every time a fault happens [6].

A source adapter can implement this caching behavior. When a querying operation returns many entries, part of them are used to initialize a source adapter. The client of this adapter (a GUI, for instance) retrieves the entries from this adapter. When a fault happens in the source adapter, it contacts the target adapter to retrieve more entries. This caching behavior is implemented in the source adapter and is transparent to the GUI [6].

iii. Data Persistence. To facilitate database access the software engineer can reuse components of Persistence framework [18]. Figure 7 shows these components. The ConnectionPool component, through its IConnectionPool interface, does the management and connection with the database used in the application. The DriversUtil component, based on eXtensible Markup Language (XML), has information from supported database drivers, available through its interface IDriversUtil. The TableManager component manages the mapping of an object into database tables, making their methods available by the ITableManager interface. The persistent component of the FacadePersistent structure, through its IPersistentObject interface, makes the values, which must be added to the database available, passing parameters to the TableManager component.

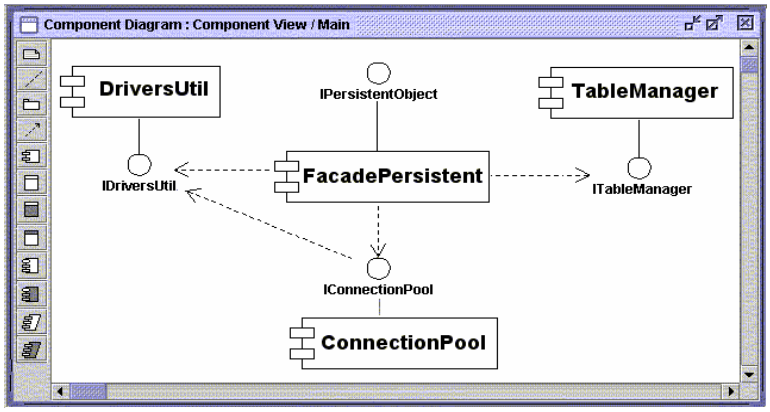


Figure 7. Framework Persistence.

In summary, the main artifacts and the sequence of the design activities of the Design Components pattern, include:

a) Refining Model of Types into Classes Models;
b) Refining the Interactions Models; and
c) Creating the Components Models.

4.5 Consequences:

✓ *Separation of concerns*: To help the software engineer in the understanding of the components, the functional and non-functional requirements are treated one at a time. This helps to avoid the confusion that exists when treating several

requirements at the same time. Other consequence of this separation is that it facilitates the implementation and testing of the components, since each requirement can be tested independently; and

✓ *An incremental project of the requirements:* when it is necessary, the software engineer can add requirements non-functional to the project, as: distribution, persistence, faults tolerance, caching etc.

✖ *Increased classes number* [15]: using the *DAP* pattern, using a pair of adapters, initialization and nomination components are necessary, causing the number of classes to increase, as well as the need for manual effort; however, these structures can be partially generated using the CASE tool, reducing this need;

✖ *Knowledge about other technologies*: using the *Persistence framework*, the software engineer needs to know technologies, like *XML* for definition of information related to database management systems, as connection port, username, password, and others.

4.6 Related or Interacting Patterns:

- *Wrapper-Facade* [17] *and DAP* [15] have the common goal of minimizing platform-specific variation in application code. However, *Wrapper-Facade* encapsulates existing lowerlevel non-object-oriented APIs (such as sockets, and threads), whereas *DAP* encapsulates object-oriented distribution APIs, such as *RMI* and *CORBA* [16].
- *Facade*, *PersistentObject* and *ObjectPool*. *Framework Persistence* is implemented using the *Design Patterns Singleton* and *Facade*, and, patterns for database persistence [18], like *PersistentObject* and *ObjectPool*.
- *Broker* and *Trader*. Well known patterns for structuring distributed systems already exist. The *Broker* [15] and *Trader* [15] patterns are examples. These are architectural patterns and focus mostly on providing fundamental distribution issues, such as marshalling and message protocols. Therefore, they are mostly tailored to the implementation of distributed platforms, such as *CORBA*. *DAP* uses these fundamental patterns and provides a higher level of abstraction: distribution API transparency to both clients and servers [12].
- This pattern is a refinement of DCDP, proposed in a previous work [6];
- It should be used together with the previous pattern presented in this paper, **Specify Components,** using the models generated in this pattern to facilitate the creation of the classes and interaction models;
- It produces deliverables that should be used in the next pattern, **Implement Components**.

4.7 Known Uses:

- The Laboratory of Software Engineering in Federal University of São Carlos (UFSCar) uses this pattern in their projects.
- This pattern was used in defining the problem of a Service Order domain, that was developed in the Computing Department of Federal University of São Carlos (UFSCar). This project generated 28 models, 40 classes and 24 components implemented.

5 Implement Components

5.1 Motivation:

A great effort is necessary in order to implement components. The design models, which specifies how the components must be implemented in order to fulfill the functional and non-functional requirements, must be translated into a low-level executable language, demanding valuable time and money resources.

5.2 Problem:

The most common problems related to the implementation tasks include: time, the unforeseeable cost, maintenance and testing.

5.3 Forces:

- Implementation is not consistent with the design, future maintenance may be prejudiced, as the elements of the design may not be fully present on the final code, and vice-versa; and
- Implementation tasks consists mainly in manual work, since the larger part of thinking was already performed before this phase. Manual work can be optimized through code generators, which speeds these tasks very considerably;

5.4 Solution:

This pattern is based on a code generation approach. A tool is used to generate the components code with basis on their design. After the code is generated, it can be refined to introduce some adjustments.

Initially, the software engineer defines the distribution technology. In the example presented, CORBA[16] was chosen, but other technologies such as RMI [14], JAMP [19] and JINI [14] can be used. In CORBA each component has stubs and skeletons and the interfaces that make its services available.

Next, the software engineer uses a CASE tool with code generation features, to implement the components. In this example, the MVCASE tool [12, 13] was used. However, any other tool that can generate executable code with basis on high-level design specifications, such as classes and components models, such as Rational Rose [22] or Together [21], could be used as well.

The models produced by the **Design Components** pattern are used as an input to the CASE tool. The tool's code generator then generates part of the code that corresponds to the components. In this case, Java was used as the implementation language. The generated code is then customized by software engineer, in order to perform some adjustments and corrections. Next, the implemented components are stored in a repository to be used on applications development in the future.

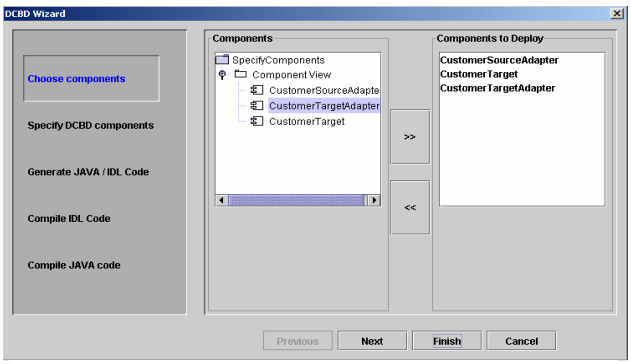


Figure 8. Generate code in MVCase tool.

Figure 8 shows the code generation process in MVCASE.

5.5 Example Implementation:

The software engineer uses the MVCASE code generator and produces customized implementations. Figure 9 shows part of the generated code to CustomerSourceAdapter, of the Service Order example.

```
package dcbd.broker.customer;
import org.omg.CosNaming.*;
import org.omg.CosNaming.NamingContextPackage.*;
import javax.naming.*;
import java.text.*;
import java.io.*;
import java.util.*;

public class CustomerSourceAdapter implements ICustomer{

  private MCustomer.ICustomerTargetInterface customer;

  public CustomerSourceAdapter(String args[]) throws CommunicationException{
    try{

      Properties props = new Properties();
      props.put("vbroker.orb.DefaultInitRef","null");
      props.put("vbroker.orb.InitRef","null");
      props.put("vbroker.agent.addr","200.18.98.65");
      props.put("vbroker.agent.port","14000");
      props.put("SVCnameroot", "NameService");
      org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,props);
      org.omg.CORBA.Object objRef = orb.resolve_initial_references("NameService");
      NamingContext nameService = NamingContextHelper.narrow(objRef);
      NameComponent nc = new NameComponent("Customer Application", "");
      NameComponent [] path = {nc};
      customer = MCustomer.ICustomerTargetInterfaceHelper.narrow(nameService.resolve(path));
    } catch (Exception e){
        System.out.println("ERROR : " + e);
        e.printStackTrace(System.out);
        throw new CommunicationException();
    }
  }
```

Figure 9. Implementation of the CustomerSourceAdapter.

5.6 Consequences:

- ✓ *Reuse*: After using and tested this pattern, implemented distributed components are delivered. These components can be later reused, on applications development. It must be emphasized that not just code is reused, but also the component's design, in a higher abstraction level;
- ✓ *Maintainability:* when using MVCASE, changes can be made directly on the component's design. Because MVCASE has a code generator, changes made on the design are reflected on the generated code. This facilitates the maintenance, since the software engineer can quickly check the effects of the changes, and take decisions more efficiently; and
- ✓ *Better quality documentation*: The generated code always reflects the exact design. This assures that the available documentation are always up-to-date with the code.
- ✘ *Knowledge about distribution technology*: It is necessary the knowledge about some distribution technology. Most of these technologies, such as CORBA, are intrinsically complex and demands great expertise in order to avoid distribution problems, such as performance and security; and

5.7 Related or Interacting Patterns:

- This pattern is a refinement of four step from DCDP, proposed in a previous work [6]; and

- When used together with the previous pattern presented in this paper, **Design Components,** the models generated in this pattern can be directly used to generate the component's code.

5.8 Known Uses:

- The Laboratory of Software Engineering in Federal University of São Carlos (UFSCar) uses this pattern and the MVCase tool in their projects.
- This pattern was used in defining the domain of a Accountancy and Invoice System, that was developed in the Computing Department of Federal University of São Carlos (UFSCar). This project generated 58 models, 50 classes and 30 components implemented.

6 Putting it All Together

Now that you have seen all of the patterns, you might be asking, "how do I put it all together?". All of these patterns collaborate together to provide a mechanism for Distributed Component Development. Figure 10 shows how the patterns interact with each other.

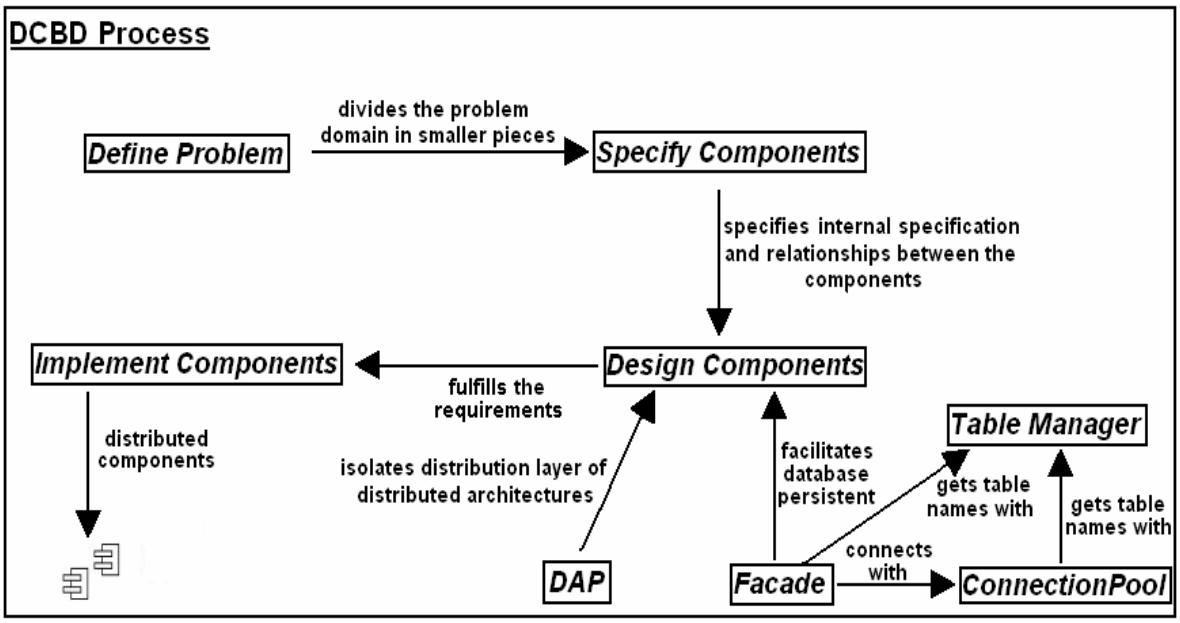


Figure 10. Pattern Interaction Diagram.

Integration of Catalysis CBD method, the principles of middleware [10], components framework (persistence) and the Distributed Adapters Pattern (DAP), a CASE Tool, it was define an process that supports the Distributed Component Development.

The components of a problem domain are built in four patterns: **Define Problem**, **Specify Components**, **Design Components** and **Implement Components**, according to Figure 11. The first three patterns correspond to the three levels of Catalysis, as shown in the right part of Figure. In the last pattern, the physical implementation of the components is done. This Figure presents the levels in waterfall model, but don't represent the process model Waterfall.

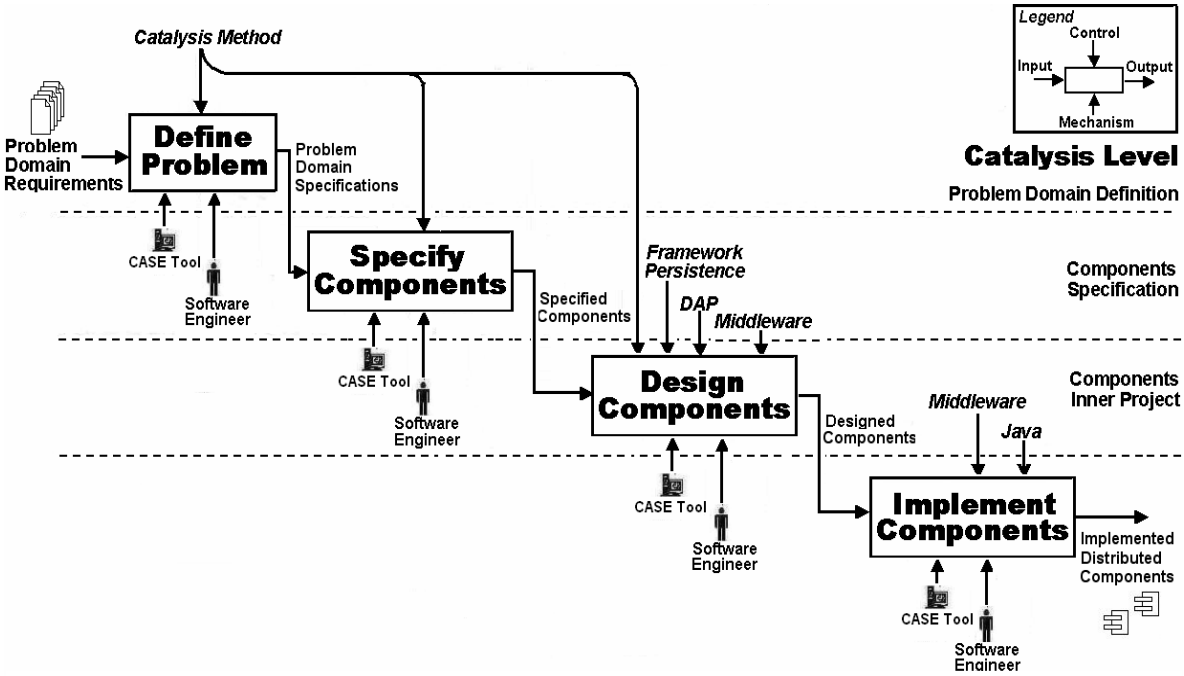


Figure 11. Distributed Components Development Process.

7 Acknowledgements

The authors would like to thank to Shepherd Robert Hanmer for suggestions received during the process. This work was supported by Fundação de Amparo à Pesquisa do Estado da Bahia (Fapesb).

8 References

[1] D'Souza, D., F., Wills, A., C., 1999. **Objects, Components, and Frameworks with UML***, The Catalysis Approach*, Addison-Wesley. USA.

[2] Jacobson, I., Griss, M., Jonsson, P., 1997. **Software Reuse: Architecture, Process and Organization for Business Sucess**, Addison-Wesley. Longman.

[3] Heineman, G., T., Councill, W., T., 2001. **Component-Based Software Engineering, Putting the Pieces Together**, Addison-Wesley. USA.

[4] Szyperski, C., 1998. **Component Software: Beyond Object-Oriented Programming**, Addison-Wesley. USA.

[5] Jacobson, I., et al., 2001. **The Unified Software Development Process**. Addison-Wesley. USA, 4nd edition.

[6] Almeida, E., S., Bianchini, C., P., Prado, A., F., Trevelin, L., C. **DCDP: A Distributed Component Development Pattern**, *In The Second Latin American Conference on Pattern Languages of Programming* (*SugarLoafPlop*), Writers Workshops, 2002, Itaipava/RJ, Brazil.

[7] Rumbaugh, J., et al., 1998. **The Unified Modeling Language Reference Manual**, Addison-Wesley. USA.

[8] Stojanovic, Z., Dahanayake, A., Sol., H, 2001. **A Methodology Framework for Component-Based System Development Support**. In *EMMSAD'2001, Sixth CAiSE/IFIP8.1.*

[9] Boertin, N., Steen, M., Jonkers., H, 2001. **Evaluation of Component-Based Development Methods**. In *EMMSAD'2001, Sixth CAiSE/IFIP8.1.*

[10] Eckerson, W., et al., 1995. **Three Tier Client/Server Architecture: Achieving Scalability, Performance, and Efficiency in Client/Server Applications**. Open Information Systems.

[11] Gamma, E., et al., 1995. **Elements of Design Patterns: Elements of Reusable Object Oriented Software**, Addison-Wesley.

[12] Almeida, E., S., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. **MVCASE: An Integrating Technologies Tool for Distributed Component-Based Software Development**. In *APNOMS'2002, The Asia-Pacific Network Operations and Management Symposium,Poster Session*. Proceedings of IEEE.

[13] Almeida, E., S., Lucrédio, D., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. **MVCASE Tool: An Integrating Technologies Tool for Distributed Component Development** *(in portuguese)*. In *SBES'2002, 16th Brazilian Symposium on Software Engineering, Tools Session*.

[14] Horstmann, C., S., Cornell, G., 2002. **Core Java 2: Volume II***, Advanced Features,* Prentice Hall.

[15] Alves, V., Borba, P., 2001. **Distributed Adapters Pattern (DAP): A Design Pattern for Object-Oriented Distributed Applications**. In *SugarLoafPlop'2001, The First Latin American Conference on Pattern Languages of Programming*.

[16] **The Common Object Request Broker Architecture**, 1996. Object Management Group. Avaliable in 10/04/2002, URL: http:// www.omg.org.

[17] Buschmann, F., et al, 1996. **Pattern Oriented Software Architecture: A System of Patterns**. John Wiley & Sons.

[18] Yoder, J., Johnson, R., E., Wilson, Q., D., 1998. **Connecting Business Objects to Relational Databases.** *In PLoP'1998*, Pattern Language of Progamming.

[19] Guimarães, M., P., Prado, A., F., Trevelin, L., C., 1999. **Development of Object Oriented Distributed Systems (DOODS) using Frameworks of the JAMP plataform**. *In First Workshop on Web Engineering, in conjunction with the 19th International Conference in Software Engineering (ICSE).*

[20] Orfali., R., Harkey, D, 1998. **Client/Server Programming with Java and CORBA**. John Wiley & Sons, Second Edition.

[21] Borland Software Corporation. **Together**. Available at site Borland Software Corporation, URL: http://www.borland.com/together - Consulted in May, 2003.

[22] IBM Rational Software. **Rational Rose® Family**. Available at site IBM Rational Software, URL: http://www.rational.com/products/rose - Consulted in May, 2003.

Switch Strategy Pattern

Luis César Maiarú[1]
Universidad Argentina de la Empresa (UADE)
Lima 77, (073) Buenos Aires, Argentina
luiscesar@argentina.com, lmaiaru@uade.edu.ar

Abstract

The Switch Strategy pattern allows to select different implementations of the same interface, depending on determined condition, without hard-coding a switch statement or a sequence of conditional statements.

1 Intención

Encapsular un conjunto de opciones compartiendo la misma interfaz. Esto permite a los clientes seleccionar una opción sin conocer la política de selección.

2 Otro Nombre

Case Strategy.

3 Motivación

Suponga que se quiera saber el precio que paga cada registro en un congreso. El mismo depende de la categoría del participante, por ejemplo, Profesor, Estudiante, Profesional miembro de una asociación reconocida, o ninguna de las opciones anteriores.

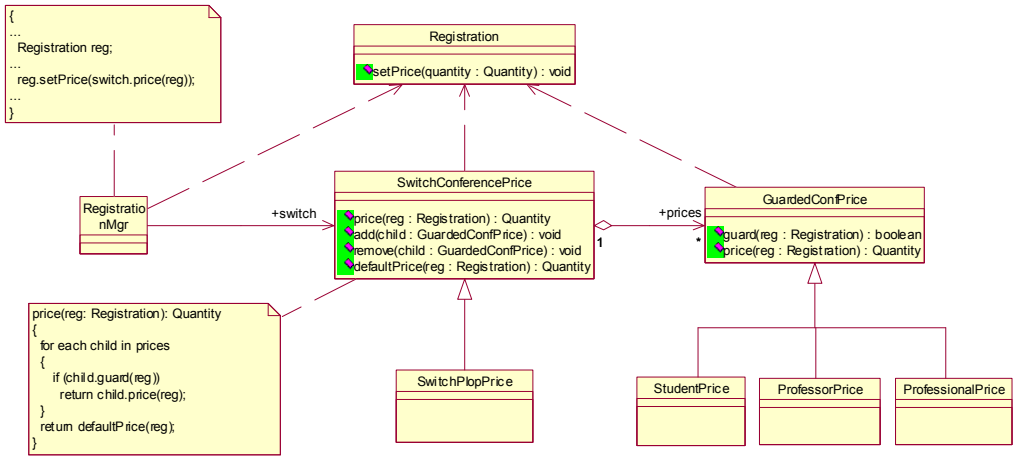
Supongamos que el precio ha ser pagado depende de la categoría del participante y otra información incluida en el registro.

Nos encontramos con la situación de seleccionar entre varias opciones, las cuáles comparten el mismo tipo de entrada y el mismo tipo de salida, pero dependiendo de cada opción una estrategia diferente será usada.

El problema es cómo diseñar una solución que satisfaga los siguientes requerimientos:

- Si las categorías o el porcentaje asociado a las mismas cambian no haya que re-escribir el código del cliente, es decir que se puedan agregar nuevas categorías en modo dinámico.
- Que el cliente no tenga la responsabilidad de tratar con el orden de evaluación si un participante pertenece a mas de una categoría.
- Que el diseño se pueda re-usar en diferentes congresos.

La solución obtenida es manejar estrategias custodiadas (guarded strategies). Un administrador de estrategias custodiadas (SwitchStrategy).) será el responsable de administrar el orden de las estrategias. La primera cuya custodia satisfaga la condición de evaluación será la elegida. El cliente tratará solo con dicho administrador. Además, el administrador tendrá las responsabilidades de agregar o sacar estrategias custodiadas. El administrador poseerá también una estrategia por defecto en caso que de que ninguna de las custodias hayan sido satisfechas.



La clase RegistrationMgr será la responsable de asignarle un precio a cada registro. Para ello se basará en la clase SwitchConferencePrice la cuál tiene una lista de estrategias custodiadas cuyas clases implementan la interfaz GuardedConferencePrice. Estas clases implementan diferentes custodias y estrategias:

GuardedProfessorPrice implementa la custodia para los participantes que son profesores e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

GuardedStudentPrice implementa la custodia para los participantes que son estudiantes e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

GuardedProfessionalPrice implementa la custodia para los participantes que son profesionales acreditados e implementa la estrategia que determina el precio del registro para el mismo dependiendo del descuento.

La clase Quantity ha sido usada para representar el tipo del precio en reemplazo de tipos mas restrictivos como double o Double con el fin de independizarnos de cuestiones implementativas.

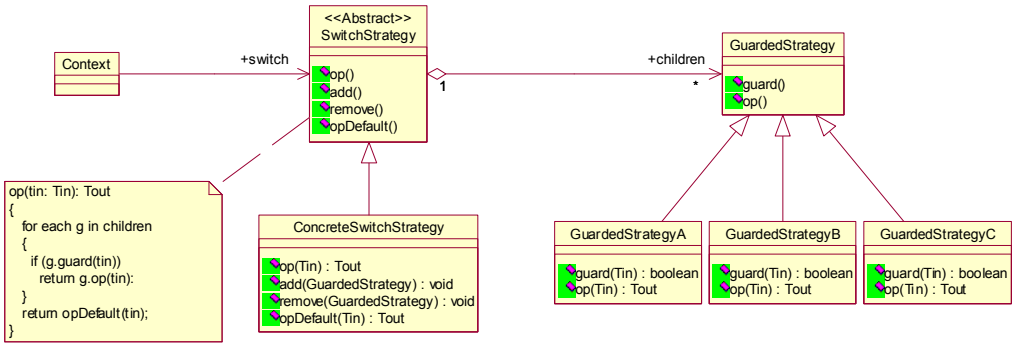
4 Aplicación

Use el patrón Switch Strategy en los siguientes casos:

- Es necesario elegir una opción de un conjunto de bloques de instrucciones dependiendo de una determinada condición. Cada bloque de instrucciones puede ser encapsulado en una operación.

- Es deseable que los clientes no tengan la responsabilidad de preguntar por la condición, sino solo llamar a una operación (interfaz), cuya implementación será seleccionada entre varias dependiendo de una condición.
- Es deseable que los clientes no manejen el orden en que las distintas opciones serán controladas por si aplican o no.
- Es deseable poder agregar o eliminar opciones en modo dinámico.

5 Estructura



6 Participantes

- **SwitchStrategy**

1. Declara la interfaz para la operación que los clientes van a usar.
2. Busca entre todos sus hijos, siguiendo un orden predeterminado, por el primero que satisfacerla condición, en tal caso invoca su estrategia.
3. Almacena los objetos hijos del tipo GuardedStrategy.
4. Declara la interfaz para una operación por defecto en caso en que ninguno objeto hijo satisfaga la condición.

- **ConcreteSwitchStrategy**

1. Puede redefinir el orden de búsqueda entre todos sus hijos.
2. Implementa la operación de defecto.
3. Implementa el agregado y la eliminación de los hijos (opciones).

- **GuardedStrategy**

1. Declara la interfaz de la operación y de la guarda.
2. Representa una opción a ser seleccionada.

- **GuardedStrategyA**

1. Implementa la interfaz de la operación, que es el objetivo del cliente.
2. Implementa la interfaz de la guarda, la cuál es usada para elegir entre las distintas opciones.

- **Context**

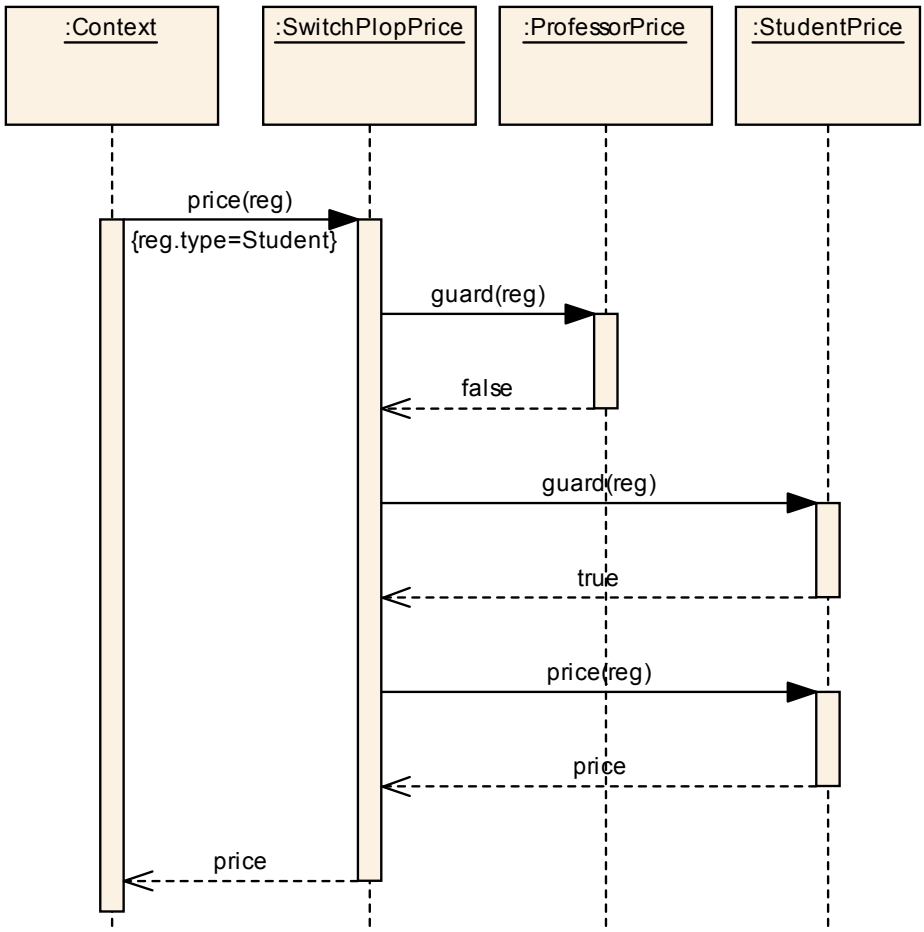
1. Mantiene una referencia a una instancia de SwitchStrategy, donde la selección y las opciones de las posibles implementaciones han sido encapsuladas.
2. Mantiene una referencia a un objeto SwitchStrategy, donde las opciones y la selección están encapsuladas.

- **Tin**

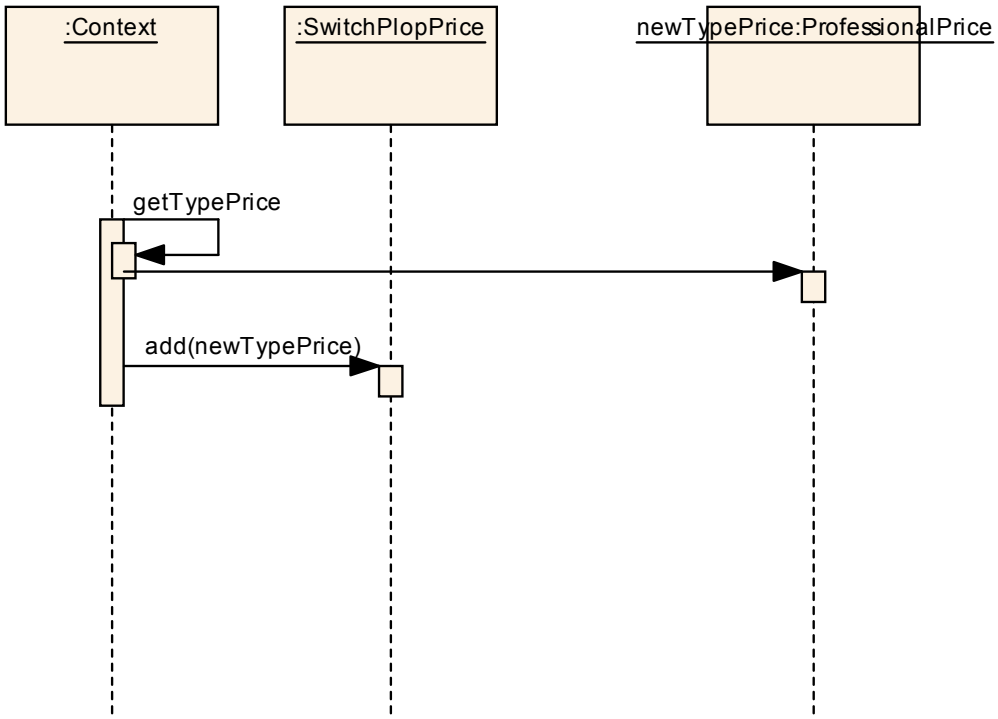
1. Es el tipo del objeto de input usado para verificar la condición de selección (custodia) y para determinar el resultado (objeto de tipo **Tout**).

7 Colaboraciones

- Los clientes usan la clase `SwitchStrategy` para referirse a la estrategia en modo uniforme. La selección de las distintas estrategias está encapsulada en la clase `SwitchStrategy`. La estrategia seleccionada será la *primera* que satisfaga la custodia (guard). Si la condición no es satisfecha por ninguna de las opciones entonces una operación por defecto es llamada.



- Las estrategias *custodiadas* pueden ser agregadas o eliminadas al objeto `SwitchStrategy` en modo dinámico sin tener que modificar el código del cliente que usa la operación.



8 Consecuencias

El patrón Switch Strategy tiene las siguientes ventajas:

1. *Facilidad en el agregado de una nueva opción en modo dinámico.* El requerimiento de considerar una opción de estrategia significaría agregar un apropiado hijo al objeto `SwitchStrategy`.
2. *Flexibilidad para determinar el orden de selección.* El orden de selección de las distintas opciones viene dado por la implementación de la subclase de `SwitchStrategy`, `ConcreteSwitchStrategy`.
3. *Flexibilidad para determinar la custodia.* La custodia (guard) de que cada estrategia es así mismo una estrategia, cuya implementación puede variar libremente respetando su interfaz con el contexto a través de la clase `Tin`.
4. *Los clientes no se deben preocupar por las diferentes estrategias.* A diferencia del patrón Strategy [1] donde el cliente se debe preocupar por cada una de las implementaciones de las estrategias, en este caso, el cliente solo se preocupa por la interfaz dada por la clase `SwitchStrategy`.

El patrón Switch Strategy tiene las siguientes desventajas:

1. *Sobrecarga en la comunicación entre el Contexto y el SwitchStrategy.* Similar al patrón Strategy [1] la interfaz `GuardedStrategy` es compartida por todas las clases `ConcreteGuardedStrategy` sin importar si los algoritmos son simples o complejos. Además en este caso la interfaz está referida a dos operaciones: la custodia y la estrategia. Así, puede ser probable que algunas de las clases `ConcreteGuardedStrategy` no usen toda la información brindada.
2. *Incremento del número de objetos.* También similar al patrón Strategy se incrementa el numero de objetos en las aplicaciones.

3. *Sobrecarga de ejecución en casos de estrategias conocidas.* Cuando las distintas posibles estrategias no cambian muy a menudo, entonces el cliente podría invocar directamente a las respectivas implementaciones. En este caso el patrón Strategy sería usado y se evitaría la búsqueda por la custodia satisfecha.

9 Implementación

1. *Definiendo la interfaz entre Context, SwitchStrategy y GuardedSrategy.* El paso de la información entre el `Context`, la clase que implementa `SwitchStrategy` y las que implementan la interfaz `GuardedStrategy` debería realizarse en un modo eficiente. En el diseño hemos propuesto hacerlo a través de parámetros (la clase `Tin`). Eso tiene la ventaja de hacer la vinculación en tiempo de ejecución. Sin embargo, como la información contenida en la clase `Tin` es la necesaria para cubrir todos las opciones podría haber casos en la evaluación de la custodia o de la operación que no sea utilizada toda esa información. Una posibilidad para evitar estos casos es la utilizar clases filtros en la `ConcreteSwitchStrategy`, para determinadas `GuardedStrategy`.
2. *Haciendo la estrategia por defecto es opcional.* La estrategia por defecto, que viene usada solo en los casos que ninguna custodia ha sido satisfecha es opcional. Podrían haber casos en donde el conjunto de objetos custodiados asegure la completitud de las evaluaciones.

10 Ejemplo de Código

Daremos una presentación de código (nivel medio) Java para el ejemplo presentado en la sección Motivación.

La clase `RegistrationMgr` tendrá la responsabilidad de responder mensajes que hagan referencia a los registros.

```
public class RegistrationMgr
{
      // …
      public Quantity price(Registration reg)
     {
            SwitchConferencePrice switch = new SwitchPlopPrice();
            return switch.price(reg);
     }

} // class RegistrationMgr
```

Cada registro contará con al menos dos atributos: la categoría del participante y el porcentaje de descuento a ser sustraído del precio total. También ha sido incluido el atributo `basePrice` representando el precio base de cada registro.

```
public class Registration
{
      Quantity price;
      String category;
      Quantity basePrice;
```

```
	public void setPrice(Quantity priceC)
	{
		price = priceC;
	} // setPrice(Quantity): void

	public Quantity getBasePrice()
	{
		return basePrice;
	} // getBasePrice(): Quantity

	// …

} // class Registration
```

La clase `Quantity` representa el tipo asociado al precio.

```
public class Quantity
{
	double value;

	public Quantity(double valueC)
	{
		value = valueC;
	} // constructor(double)

	public double getValue()
	{
		return value;
	} // getValue(): double

	// …
} // class Quantity
```

La clase abstract `SwitchConferencePrice` define el tipo de información que será usada por la implementación `SwitchPlopPrice` y por los objetos `GuardedConferencePrice` ligados a la misma.

```
public abstract class SwitchCoferencePrice
{
	Collection children;

	public Quantity price(Registration reg)
	{
		GuardedConferencePrice gCP;
		Iterator it = children.iterator();
		while (it.hasNext())
		{
			gCP= (GuardedConferencePrice) it.next();
			if (gCP.guard(reg))
			{
				return gCP.price(reg);
			}
		}
		return defaultPrice(reg);
```

```
      } // price(Registration): Quantity

      public void add(GuardedConferencePrice) {}
      public void remove(GuardedConferencePrice) {}
      public Quantity defaultPrice() {}

} // class SwitchConferencePrice
```

La clase `SwitchPlopPrice` extiende la clase abstract expuesta más arriba. En este caso, la forma en recorrer la lista de objetos `GuardedConferencePrice` no es considerada en detalle. La primera que valida la custodia es la seleccionada.
Además se implementa `defaultPrice` retornando un valor dependiendo del descuento que posea el registro.

```
public class SwitchPlopPrice extends SwitchConferencePrice
{
      public SwitchPlopPrice()
      {
            children = new Vector();
      }

      public void add(GuardedConferencePrice gCP)
      {
            children.addElement(gCP);
      } // add(GuardedConferencePrice): void

      public void remove(GuardedConferencePrice gCP)
      {
            children.removeElement(gCP);
      } // remove(GuardedConferencePrice): void

      public Quantity defaultPrice(Registration reg)
      {
            // Return a default value
            return new Quantity(5 * reg.getBasePrice.getValue());
      } // price(Registration): Quantity

} // class SwitchPlopPrice
```

La interfaz `GuardedConferencePrice` define la forma en que el pasaje de información es realizado, tanto para la custodia como para la operación precio. En este caso, el mismo tipo de parámetro es usado.

```
public interface GuardedConferencePrice
{
      public boolean guard(Registration reg);
      public Quantity price(Registration reg);
} // interface GuardedConferencePrice
```

Las clases `GuardedProfessorPrice`, `GuardedStudentPrice` y `GuardedProfessionalPrice` implementan la interfaz descripta anteriormente. Cada una de estas clases tiene su propia interpretación de la custodia y del precio del respectivo

registro. El cuál, a igual que en el caso por defecto, depende del registro pasado como parámetro.

```
public class GuardedProfessorPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Professor"));
      }
      public Quantity price(Registration reg)
      {
            return new Quantity(3 * reg.getBasePrice.getValue());
      }
} // class GuardedProfessorPrice

public class GuardedStudentPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Student"));
      }
      public Quantity price (Registration reg)
      {
            return new Quantity(2 * reg.getBasePrice.getValue());
      }
} // class GuardedStudentPrice

public class GuardedProfessionalPrice
      implements GuardedConferencePrice
{
      public boolean guard(Registration reg)
      {
            return reg.category.equals(new String("Professional"));
      }
      public Quantity price(Registration reg)
      [2]{
            return new Quantity(4 * reg.getBasePrice.getValue());
      }
} // class GuardedProfessionalPrice
```

11 Usos Conocidos

- Un particular uso de este patrón es en la validación de reglas las cuáles serán evaluadas dependiendo de determinadas condiciones. Por ejemplo, el Switch Strategy fue usado en el *validador* de ofertas de un Sistema de Mercado Electrónico de Bolsa de Valores[2]. Al ser ingresada una oferta (de compra o de venta) por los operadores, se debía verificar su validez antes de que la misma pudiese ser utilizada para concertar operaciones. La oferta tenía distintos atributos. Verificar la validez de la oferta implicaba evaluar distintas reglas dependiendo de condiciones que hacían referencia a

[2] Electronic Stock Exchange Market System

dichos atributos. Debido a las fuertes restricciones de tiempo en el ingreso de una oferta, verificar todas las reglas aún aquellas que no aplicasen no era deseado. El Switch Strategy permitió agrupar bloques de reglas cuya validación dependían de una misma condición. El *validador* de ofertas solo tenía que interactuar con bloques *custodiados* de reglas.

- Otro uso conocido de este patrón fue en la programación de un *Broker* Java para Base de Datos. El*Broker* hacía la veces de intermediario entre el cliente y la base de datos, abstrayendo al primero de interactuar directamente con los distintos conceptos relacionados con la base de datos. Una de las responsabilidades del Broker era la de *asociar* clases Java con tipos de Base de Datos en modo *completamente arbitrario*. El Switch Strategy permitía que la *estrategia de asociación* tuviera siempre la misma interfaz, haciendo la custodia (*guard*) referencia a las distintas clases y tipos.
- El patrón Switch Strategy fue también utilizado en un Sistema de Categorización de Especies (Stock). Dada una lista de especies, las mismas, de acuerdo a sus atributos, eran agrupadas por categorías. Por cada categoría luego se calculaban distintos tipos de información como aranceles y totales. Las categorías y las condiciones de pertenencia de una especie a una categoría podían variar frecuentemente. El Switch Strategy permitía obtener en todo momento la categoría a la que pertenecía una especie sin modificar el código del cliente. Así mismo permitía asociar y desasociar especies de las categoría en modo dinámico.

12 Patrones Relacionados

Flyweight [1]. El patrón Switch Strategy tiene una directa relación con el patrón Flyweight. La clase Tin podría ser usada como la clave (key) en el patrón Flyweight. La diferencia radica en que en el caso del Switch Strategy no se necesita una relación uno a uno entre las claves y las estrategias. El orden de selección viene determinado por la clase ConcreteSwitchStrategy. Además la creación de la estrategia custodiada es independiente de la clave, cosa que si ocurre en el patrón Flyweight.

Composite [1]. El patrón Switch Strategy se complementa muy bien con el patrón Composite. Usándolos en conjunto es posible anidar bloques custodiados de instrucciones.

Strategy and Abstract Factory [1]. El patrón Switch Strategy puede ser también un buen complemento de los patrones Strategy y Abstract Factory. Cuando el patrón Strategy es usado por un cliente, una estrategia concreta deberia ser usada. Esto puede ser realizado usando el patrón Abstract Factory. El Switch Strategy podría ayudar al Abstract Factory en el proceso de selección si varias estrategias están siendo consideradas.

The Adaptive Object Model Architectural Style [6]. El patrón Switch Strategy puede ser usado con el *Adaptive Object Model Architectural Style.* El Switch Strategy podría ayudarlo en el manejo de grupos de reglas de negocio cuya validación dependa de una determinada condición. También, es posible usar el Switch Strategy en reemplazo del patrón Strategy cuando en tiempo de ejecución es requerido considerar una familia de algoritmos simultáneamente.

13 Agradecimientos

El autor quisiera agradecer al *shepherd* Jorge L. Ortega Arjona por sus valiosos y precisos comentarios. Los mismos constituyeron un importante aporte para el mejoramiento

del presente trabajo. Además quisiera agradecer a los participantes del grupo en el congreso, quienes brindaron positivas sugerencias.

14 Referencias

[1] Gamma, E., Helm, R., Johnson R., and Vlissides, O. (995) *Design Patterns: Elements of Reusable Object-Oriented Software*, Reading, MA: Addison Wesley Longman.
[2] Fred Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal, *Pattern Oriented Software Architecture: A System of Patterns*, John Wiley & Sons, 1996.
[3] Wolfgang Keller, *Mapping Objects to Tables, A Pattern Language.* In Proceedings of the European Pattern Languages of Programming Conference, Irrsee, Germany, 1997. Also at http://www.objectarchitects.de.
[4] Wolfgang Keller, *Object/Relational Access Layers, A Roadmap, Missing Links and More Patterns.* In Proceedings of the 3rd European Conference on Pattern Languages of Programming and Computing, 1998. Also in http://www.objectarchitects.de.
[5] Floyd Marinescu, *EJB Design Patterns, Advanced Patterns, Process and Idioms,* John Wiley and Sons, 2002.
[6] Joseph W. Yoder and Ralf Johnson, *The Adaptive Object Model Architectural Style,* IEEE/IFIP Conference on Software Architecture 2002 (WICSA '02) at the World Computer Congress in Montreal 2002, August 2002.

GIG-Pattern

Maria Lencastre
Felix C. G. Santos
Mardoqueu Souza Vieira
Mechanical Engineering Department
Federal University of Pernambuco
Rua Acadêmico Hélio Ramos, S/N
Recife, PE 50740-530 – Brazil
mlpm@cin.ufpe.br, fcgs@demec.ufpe.br, msv@cin.ufpe.br

Abstract

This paper presents a pattern called GIG, a generic interface graph which deals with definition and control of processes taking into account some specific requirements of simplicity, easiness of definition from algorithmic language and flexibility in the granularity of defined processes. The pattern is intended to help the design and reuse of programs.

1 Introduction

The use of workflow technology helps the development of more flexible and versatile computation strategies. So, workflow management systems are a relevant support for large class of business applications, and many workflow models as well as commercial products are currently available [8]. While the large availability of tools facilitates the development and the fulfilment of customer requirements, workflow applications still require simple, generic and adaptive solutions for the complex task of rapidly producing effective applications, especially when complex domains are involved.

The GIG pattern was developed after we noticed that many numerical algorithms showed the very same organizing structure when trying to achieve process reuse and flexibility for the adaptation to new strategies. Such an organizing structure in turn allowed for an abstraction, which resulted in the GIG, a generic interface graph. As it will be seen, it is possible to devise frameworks to use the GIG pattern in order to implement different processes in a very flexible and automatic way.

The GIG-pattern describes an abstract workflow solution, whose purpose is to provide expressiveness and adaptability through simplified workflow programming, control and use [8]. Other GIG motivation is to maintain predefined algorithmic structure, which means that the translation from algorithmic language representation of the processes into a computer representation must be as direct as possible. This is important because, the achievement of similarity between the way the programmer has its algorithmic code organized and the implementation of it, can bring simplification in further required changes. Also, sometimes, developers need solutions that does not make restrictions on the scale of the process, that is, which need a mixture of small-scale processes (that execute within applications) and large-scale processes (that execute on top of applications), usually this happens when designers are also the programmers.

As a workflow pattern, GIG provides for the separation of process logic from task logic, which is embedded in user applications, allowing the two to be independently modified and the same logic to be reused in different cases. The GIG-pattern considers features related to run-time control functions [7], which manage the workflow processes and sequence the various activities.

This work was devised from the experience obtained during the implementation of several simulators in the FEM context [5]. FEM is a way of implementing an approximate mathematical theory for a physical behaviour. Researchers of the Mechanical Engineering Department – UFPE – Brazil found the need to organize their code in a way that was easier to adapt to new strategies and also to allow process reuse. So they designed and implemented the GIG, a generic interface graph, which provides an interface for process control dealing with the specific requirements mentioned. In this paper the GIG is presented as a pattern.

The pattern's description is organized in the following way. In section 2 the pattern name is identified. Section 3, details the context in which the pattern solution applies. Section 4 presents a motivation example for the GIG-pattern use. Section 5 presents the design challenge through a question. Section 6 shows pattern forces, that is, the patterns design trade-offs, what pulls the problem in different directions, towards different solutions. Section 7 explains how to solve the problem. Section 8 describes the pattern implementation. Section 9 presents some variants that can extend the pattern. Section 10 presents a simple example of use, in order to clarify the pattern use and section 11 presents a more complex one in the FEM simulators context. Section 12 details the resulting context, telling which forces the pattern resolves and which forces remains unresolved by the pattern. Section 13 presents related patterns. Finally, section 14 talks about known uses.

2 Name: GIG-Pattern, Generic Interface Graph for Process Control.

3 Context

Domain specific users, like scientists and engineers, usually program in a procedural style. The explanation for that, in spite of the force of tradition, may be the following. Complex numerical systems usually make use of many different pre-built auxiliary packages (like numerical integrators, solvers for non-linear and linear systems of algebraic equations, and so on) and have their procedures described in algorithmic language. So, the majority of the work is related to making the modules compatible in a monolithic architecture, which resembles the structure of the algorithm. This is a strong force that drives those users towards the procedural style.

During the development of a software system, those developers need functions that help them to organize their logical processes and their involved tasks, in a way that makes easy its future alteration for adapting to new solutions and for the reuse of software routines, avoiding heavy reprogramming. We have repeatedly noticed that many numerical algorithms showed the very same organizing structure. Such an organizing structure comes from the procedural style of the algorithm representation and can be identified to be a Directed Acyclic Graph (DAG) [5].

4 Motivation Example

Consider, for example the case of mesh generation algorithm. A mesh can be described as a partition of a geometric domain into simple geometric entities (triangles, tetrahedra, hexahedra, etc) called geometric finite elements (or simply elements). In Figure 1 we present the algorithm for a particular mesh generation, which, given a plane straight-line graph (PSLG), generates a mesh of triangles.

```
I.  Data input (PSLG)
II. Generate the bounding box for the PSLG
III.Build the initial mesh of the bounding box
IV. For each point in the PSLG do
    IV.I.  Insert point
       IV.I.I.  Find elements affected by the new point
       IV.I.II. Eliminate those elements obtaining the affected region (AF)
       IV.I.III Build new elements from the new point and boundary of AF
V.  Find a line of the PSLG such that it is not an edge of any triangle
                                    (negative line)
VI. While there still is a negative line do
       VI.I Compute the middle point of the line
       VI.II insert middle point (see IV.I)
VII. Eliminate those triangles, which have any point of the bounding box as
     one of their vertices.
VIII.Data output
```

Figure 1 Mesh Generation Algorithm

This algorithm can be represented using the graph structure presented in **Erro! A origem da referência não foi encontrada.**. Observe that there are fifteen sub-routines, including the driver (which executes the procedures I- VIII). This graph structure can be represented in GIG-pattern (see Figure 2). Each one of those processes can be encapsulated in an object of a class, representing a node of the graph. The proposed pattern describes it as a derivation of a base class called *AlgthmNode.*

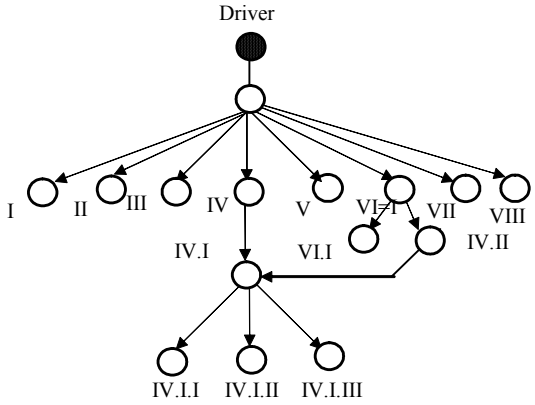


Figure 2 Mesh Generation Graph

Observe that there are many different ways of performing each one of the activities (tasks) described in the above algorithm. For instance, *IV.I.I find elements affected by the new point* concerns a search method in a geometric database of triangles, looking for a triangle whose circumscribed circle contains a given point. There are a lot of search methods available in the specialized literature, each one with its advantages, drawbacks and dependence on special data structures. Replacing the current method by a new one will not affect any other place in the graph.

Entire branches can also be changed as well. For instance, the process *IV.I. insert point*, can be changed by plugging another method to perform that task. That means that all the subsequent processes (children nodes) will be also changed. Besides the severity of the change in the methods needed by the algorithm, all the substitution work can be automatically performed.

On the other hand, the Data Domain of this problem can be decomposed in such a way that all *AlgthmNode* objects (subroutines) will have access only to the data it needs. For instance, the process *III.Build an initial mesh for the bounding box* will need the bounding box and will build the initial mesh, which will be stored in a place in order to be accessed by other nodes. That decomposition will give rise to the classes derived from AlgthmData. The whole set of data pieces depend on the geometric data structure used by the developer. For instance, it can be seen that some structures have to be present: (a) PSLG (accessed by I, II, IV and V); (b) bounding box (accessed by II, III and VII), (c) mesh (accessed by III, IV.I.I, IV.I.II, IV.I.III, V, VII and VIII), (d) auxiliary data (many, it depends on the designer). All those pieces of data will be encapsulated in objects of classes derived from *AlgthmData.*

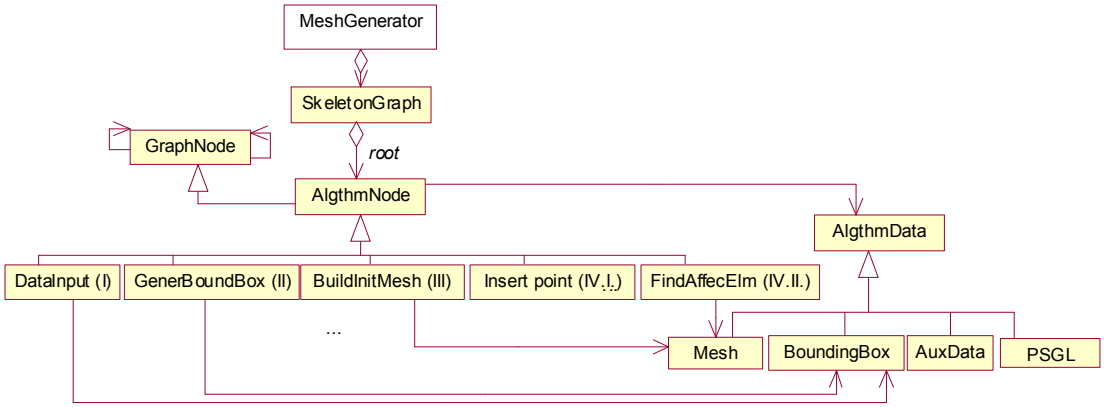


Figure 2 Application of the GIG structure in Mesh generation algorithm

In this example the mesh generation process is the controlled workflow. This process includes information about constituent tasks (represented as the processes (I to VIII). The mesh generation process has requirements related to modularity and exchange of sub-routines, since it has specific parts that have several kinds of implementations, which can be exchangeable.

5 Problem

How to guarantee simplicity in the separation of process logic from task logic, during the development of complex systems, while maintaining solution independence, reuse of processes and the predefined algorithmic structure?

6 Forces

With respect to the defined context, there are different forces, which lead to different solutions. Some of these forces are:

- Maintaining predefined algorithmic structure;
- Simplicity in the process definition;
- Support for different levels of granularity on the defined processes;
- Domain independence;
- Dynamic change of workflow processes;
- Reduction on error occurrences in the coupling of processes;
- Reuse of processes;
- Parallelism and processes synchronization;
- Workflow execution performance;
- Explore existing expertise of domains of knowledge.

The following discussion analyses some of these forces, in order to identify how they are pulling against each other. GIG tries to resolve some opposing forces in the workflow definition context.

When trying to maintain the predefined algorithmic structure, the definition of some sub-process could generate pieces of code that are not easily changeable, because are monolithically defined as a block of code. On the other hand, refined levels on process portioning can provide a process definition at statement level, eliminating existing abstractions (like blocks or modules). Domain independence and dynamic change of process requires abstractions like polymorphism and encapsulation, which are not present in a procedural style (the predefined algorithmic structure).

The guarantee of simplicity in process definition can be one way to avoid errors and stimulate the pattern use. The reuse of already developed and tested processes helps in the simplification of process definition, like the possibility of reusing entire solutions. However, the reuse of processes can also reduces the simplicity due to the need for extensions of classes or configuration. Some other opposing forces to simple process definition are: the guarantee of domain independence, which makes more complex the process definition; also, to allow the

definition of processes parallelism and synchronization the programmer has to deal with extra levels of complexity. The simplification can be compromised when parallelism is required for increasing performance.

Process reuse improves reduction of errors once pre-tested software is incorporated. Refined levels of granularity, in process definition, provide higher level of tangibility in the number of processes to be controlled, increasing the reuse of processes. The guarantee of domain independence also increases the number of reusable process.

Domain independence, avoiding non-monolithic solutions, makes possible the application of the workflow solution to different applications, improving its reuse. However, in these cases the existing expertise of a knowledge domain cannot be appropriately explored to improve the solution. Also, synchronization and parallelism improve in one-way domain independent application supporting the required functionality to existing applications.

The dynamic change of workflow processes improves the solution power, however, gives the programmer the responsibility and the complex task of making a suitable division of code and data, for further exchanging to be pertinent. The reuse of process is fundamental when the user has to change an existing one by another one, which is already tested and classified. Maintaining the pre-defined algorithmic structure does not help domain independence because it does not provide, for example, encapsulation.

Parallelism and processes synchronization are very relevant to allow system optimization and higher levels of control. The refined levels of granularity, in process definition, can allow a more precise level of parallelism definition. The dynamic change of workflow and the reuse of process increase the synchronization power, in process exchanging. Synchronization and parallelism improve in one-way the power of the dynamic change of process (identifying which process are independent or the order of the dependence). On the other hand this can arises more complexity in the changing of processes.

Maintaining the predefined algorithmic structure can sometimes improve performance due to the direct application of some available optimized code; parallelism also improves performance, since allows simultaneous execution of process. On the other hand, simplicity on process definition can decreases the performance, when it eliminates, for example, the possibility of parallelism definition. The guarantee of domain independences can also decrease performance once the existing expertise cannot be appropriately explored. Other forces which compromise the performance, due to the need of extra verification and controls, are: refinement level of the granularity of process definition; dynamic process exchange requires more controls; control of errors, reuse of process reduces the performance, synchronization.

7 Solution

GIG can be described as a workflow solution [7]. GIG follows the object-oriented style for modelling and programming. For simplicity reasons of use and facility in correctness verification, GIG implements a restricted Direct Acyclic Graph (DAG) [5].

7.1 Participants (Structure)

The GIG structure is presented in the UML diagram below (Figure 3).

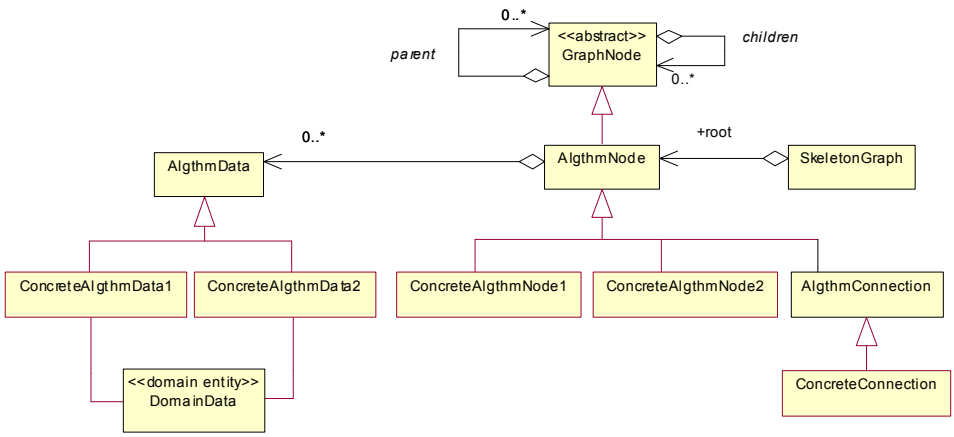


Figure 3 Participants of the GIG-pattern

The GIG pattern is composed of the following participants:

- *GraphNode*: it is an abstract class that implements low level operations related to the interoperability between graph nodes. It controls the relationship between the workflow activities.
- *SkeletonGraph*: it has a reference to the driver of an algorithm graph and encapsulates tools for performing some graph operations. It can be seen as the root of workflow process.
- *AlgthmNode*: represent subroutines that compose the application (workflow tasks). It is used as a base class for all algorithm classes of the application.
- *ConcreteAlgthmNode* Implements a specific subroutine for a task. It invokes other subroutines which can be tasks (defined as its children) or other defined applications.
- *AlgthmData*: represents a data type to be used by an instance of *AlgthmNode.* It is used as a base class for all algorithm data classes of the application.
- *ConcreteAlgthmData* Represents data from the application domain, that is used in *ConcreteAlgthmNode* classes.
- *DomainData*: represents the whole set of types related to the problem domain data.
- *AlgthmConnection*: it is an *AlgthmNode*, which references an algorithm subrotine that was not connected to the graph. This class responsibility is to fetch, and build (like a proxy [10]) the related algorithm and replaces itself with the fetched algorithm. In this way several software processes represented by *SkeletonGraph*s can be assembled producing a complex software system.

7.2 Collaborations

We can identify the following collaborations between GIG participants:

- *GraphNode* encapsulates the responsibility of providing access to other *GraphNodes*, which are its children.
- *ConccreteAlgthmNode* executes the associated process (subroutine) with the help of other processes represented by its children, through calls inserted in its process code. It relies on *GraphNode* to have access to its children *AlgthmNodes*.
- *ConcreteAlgthmData* provides access to workflow data. *AlgthmNode* communicates with Concrete*AlgthmData* objects to have access to its data.
- *AlgthmConnection* provides the dynamic connection for *AlgthmNodes*. The way objects of this class interact with its *SkeletonGraph* or its parents *AlgthmNodes* depends on the implementation. The important thing is that it represents the point where a driver node of a software process/subroutine will be plugged in. It also contains the necessary information about the new *AlgthmNode*.

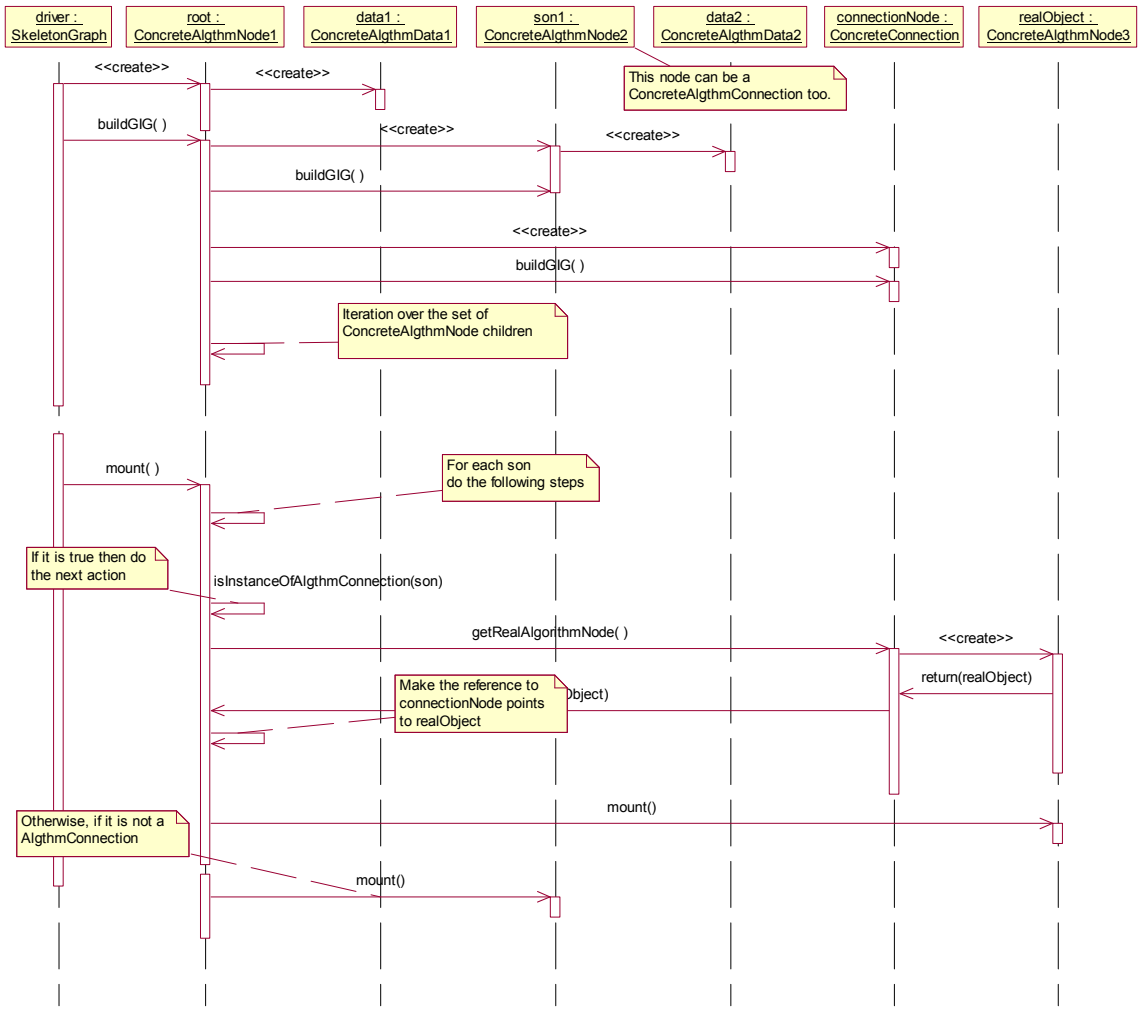


Figure 4 Sequence diagram for GIG building

We can summarize the main part of GIG-pattern interaction, through the UML sequence diagram of Figure 4. The GIG driver, an object of *SkeletonGraph* class, creates the GIG root that is an instance of *AlgthmNode* class. When the *SkeletonGraph* requests the root to build the GIG, it instantiates each child and asks them to build its sub-graphs recursively. After, the graph building the driver waits for requests for GIG mounting, CIG execution, GIG reprogramming, and for other graph manipulation functionalities. The GIG mounting is the act of remove the references to *ConcreteAlgthmConnection* objects and so preparing the GIG for execution. The GIG execution is similar to the GIG building procedure by changing *buildGIG* calls for *execute* and without *<<create>>* calls, during the execution each *ConcreteAlgthmNode* its subroutine accesses the *ConcreteAlgthmData* objects associated with it's node.

8 Implementation

There are some implementation issues associated with the GIG participants, described previously, who need some extra explanation. Other important details about implementation are related to the design steps to be followed by the user for applying the GIG-pattern to a new application. In section 9, and 10 these steps are exemplified.

8.1 Implementation Issues

The *DomainData* is implemented by a set of subclasses of the *AlgthmData*. The subclasses of *AlgthmData* describe the specific domain treated in the problem. An example of the partition of the domain in different levels can be seen in the section 11.

The *AlgthmData* and *AlgthmNode* objects must be materialised for the workflow they are serving. The materialization activities, of *AlgthmData* and *AlgthmNode* objects, can be delegated to object factories that are responsible for accessing the data repository and instantiating the objects, these object factories can have object pools to reuse objects, see section 13 for details about the patterns that can be applied.

The *AlgthmNode* subclasses need to cast the *AlgthmData* objects, associated with each node, to the primitive type.

As was shown each *AlgthmNode* object must have a reference for all its children and data. This reference can be hard coded in *AlgthmNode* subclass, or in a file, or can be handled by another class, wihich has the responsibility to relate each *AlgthmNode* with its children. An example of such a class is *DataAlgthmServer* use in the example in section 10. In this case each *AlgthmNode* can ask to the *DataAlgthmServer* for its children and data or the *DataAlgthmServer* can be active and responsible to build the GIG.

8.2 The Design Steps

The following design steps describe which actions the user needs to perform to apply the GIG-pattern to a problem:

1) Starting from an algorithm in natural language the procedure is first divided into different algorithm sub-routines (algorithm nodes) and then it is organized in the form of a graph.
2) The division of the algorithm into several routines induces a decomposition of the domain data in order to provide them with an appropriate distribution of access to the data. The result of this process gives the *AlgthmData* set.
3) Each *AlgthmNode*, that is an algorithm sub-routine, places calls to its children nodes, which implement processes inside the whole process. The logic is defined inside each *AlgthmNode* subclass and it references the execution of a child algorithm, independently of the routine that is in that child.
4) Each *AlgthmNode* is related to a set of *AlgthmData*, which may be shared with other nodes.
5) The driver of the whole process is identified.

9 Variants

(i) Use of the *TypeObject* pattern [9] to enhance adaptability, producing independence between the software routines and its data components. This is important in situations where the same software component is to be used in different situations and with different pieces of data. The class diagram is like the one in Figure 5. With this extension, AlgthmType provides the AlgthmNode the needed functionality independently of AlgthmData. The relationship between the *AlgthmData* and *DataType* can be done at run time. This extension does not affect the already described interactions of *AlgthmNode* and *AlgthmData* with the other participants.

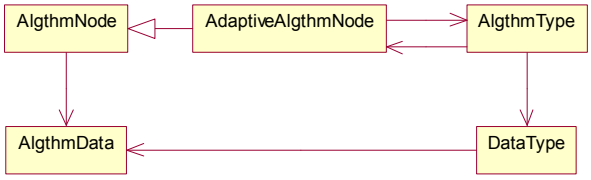


Figure 5 Class diagram for a variant of AlgthmNode

(ii) Hierarchical levels of procedures can be defined, helping in the software management. An application of this extension can be seen in section 11, where three levels of *SkeletonGraphs* were defined. For each one of those levels one may define specific functionalities for all their respective *AlgthmNodes* and *AlgthmData*. Also, at the level of the functionalities of the *SkeletonGraphs* objects specific tools can be defined. These extensions can be oriented for the applications being considered.

(iii) It can be extended to deals with the definition/execution of processes running in a distributed environment. We will not go into further details because this is still under development.

10 First Example of Usage

To exemplify and make clear the use of the GIG-pattern, a very simple application was designed. This application involves the generation of random sequence of items, which are further sorted. A better understanding of the GIG applicability and power, however, can be seen in sections 4 and 11. The proposed application can be subdivided into the following sub-processes (algorithm sub-routines): generation of a random sequence of items; sort of these items; and display of the sorting items. The sort sub-process is implemented here using the Heapsort algorithm derived from [11]. Any one of the sub-processes can be modified afterwards to another one, generating a different solution algorithm.

The Heapsort algorithm written in natural algorithmic language is described in Figure 6. This algorithm is an example solution to a very well known problem, which has many solutions.

In this example the decomposition of the domain data is very simple. It generates only the *HeapSortData* data type, which is used by all Heapsort sub-routines. The created *AlgthNode* classes are derived from the process organization in Figure 7.

Each *AlgthmNode* that is an algorithm subroutine places calls to its children nodes, which implement processes inside the whole process. The logic is defined inside each *AlgthmNode* subclass and it references the execution of a child algorithm, independently of the routine that is in that child.

```
HeapSort (A)
            Buil_Max_Heap (A)

for i ← length[A] down to 2 do
Exchange A[1] ←→ A[i]
heap-size[A] ← heap-size[A] –1
Max_Heapify

Build-Max-Heap(A)
Heap-size[A] ← length[A]
For I ← length[A]/2 downto 1 do
Max-Heapy(A,i)

Max-Heapify(A,i)
l ← LEFT(i)
r ← RIGHT(i)
if l <= heap-size[A] and A[l] > A[I]
   then largest ← l
   else largest ← i
if r <= heap-size[A] and A[r] > A[largest]
   then largest ← r
if largest != I
   then exchange A[I] ←→ A[largest]
        Max-Heapify(A,largest)
```

Figure 6 Heapsort Algorithm

The algorithms organization in the form of a direct acyclic graph can be seen in Figure 7.

```
Main Program

I. Generation of aleatory data
II. HeapSort
 II.I Build-Max-Heapify
 II.I.I Max-Heapify
III.Print
```

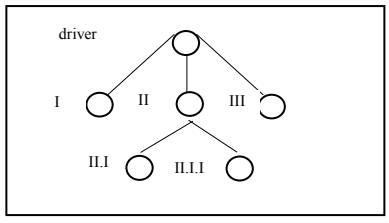


Figure 7 . Main program and correspondent GIG direct acyclic graph

Figure 6 shows the UML class diagram, which was created to implement the application classes in C++ [12], which applies the GIG-pattern. Some classes, described below, were created to implement the GIG pattern and solve the sorting example:

- The *Factory* class follows the GIG implementations suggestions, described in section 9. They define a common interface to materialise *AlgthmData* and *AlgthmNode* objects from a data source. The *FactoryHeapSort* class was created to materialise objects from the classes created to represent the Heapsort algorithm in GIG.
- The classes created to represent the *AlgthmNodes* are: *GenerateRandomAlgthmNode*, *HeapSortAlgthmNode*, *BuildMaxHeapAlgthmNode*, *MaxHeapifyAlgthmNode*, and *PrintAlgthmNode*, each one being related to a procedure described in the HeapSort algorithm, see Figure 7.
- The *HeapSortData* is an *AlgthmData* subclass created to store the sequence of items to be sorted and some control variables.
- The root (driver) of the whole process is here the *GenerateRandomData*

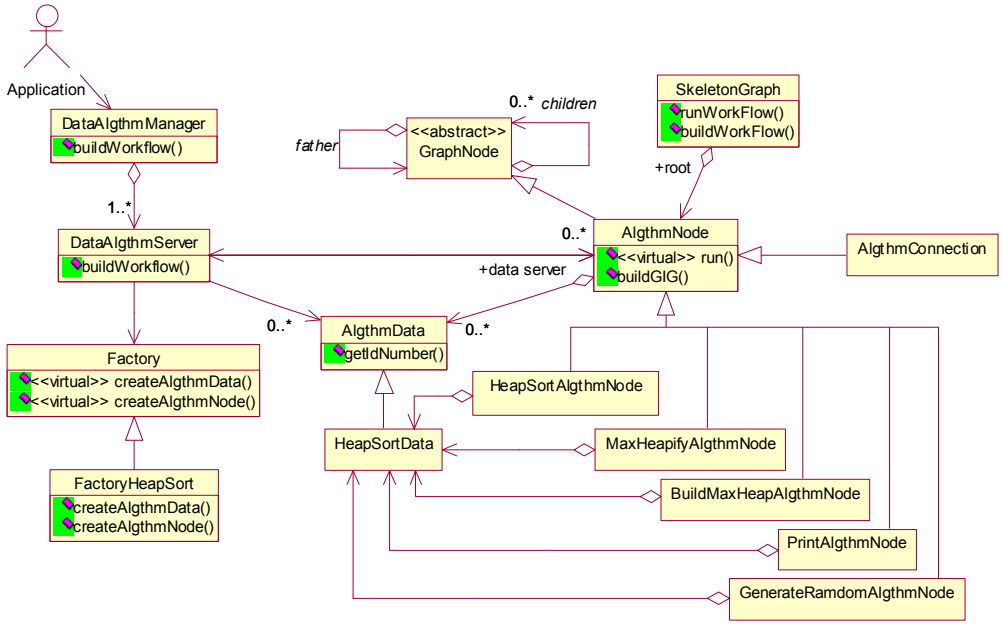


Figure 8 The Class diagram of the GIG implementation and example classes

The sample code presented in Figure 10 is an extract from the implementation of the class *HeapSortAlgthmNode*. With this example, we can understand how the implementation of the *run* method of an *AlgthmNode* class calls the children nodes in its code. In this example the *HeapSortAlgthmNode* class order its first child to run, before some other tasks are performed.

```
void HeapSortAlgthmNode::run() {
    this->runChild(0); //run Build-Max-Heap
    for( int i = heap->size(); i >= 2; i--){
        heap->swap(1,i);
        heap->decrementHeapSize();
        this->runChild(1);//run Max-Heapify
    }
}
```

Figure 9 Implementation of the run method for the class HeapSortAlgthmNode

11 Second Example

In [6], we present an application of GIG in FEM simulators. As usual it is observed that an algorithm defined for the solution of a problem by the FEM has repeated (similar) hierarchical structures. Thus in the pursuing of a high degree of reusability, therefore a framework considering hierarchical levels of processes were used, where each level may have several possibilities of algorithms, and can be easily described by a GIG graph. The whole hierarchy is represented making the connections between the different levels and generating a complete graph. Global Skeleton, the Block Skeletons, the Group Skeletons, and the Phenomena procedures define those levels [2]. These levels satisfy a number of requirements, such as: (i) to separate less reusable modules from reusable ones; (ii) to make it more comprehensible the decomposition of the simulation data among the several processes; (iii) to make it possible the dynamic re-configuration of the simulator through the replacement of reusable modules.

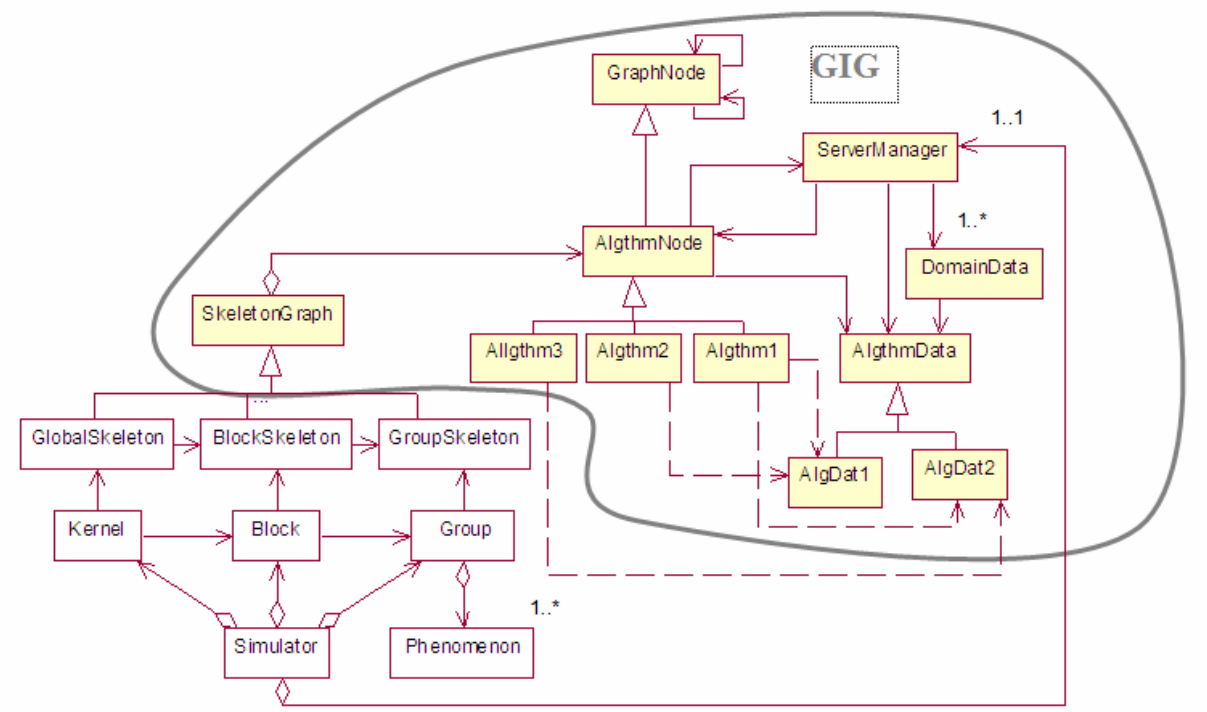


Figure 10 FEM Simulator and GIG classes

For instance, the global Skeleton articulates time loop (if present), adaptation iterations and defines processes involving the call of Block Skeletons. Block Skeletons may define different solution strategies for different Groups, thus, articulating Group processes. Group Skeletons articulate their phenomena procedures in very specific less reusable ways. It is in this level that solvers for algebraic systems are applied. Phenomena are the abstraction of the entities being simulated. All those skeletons can be implemented as objects from classes following the GIG pattern (see Figure 10). Therefore, the GIG would allow for the realization of the interoperability of the different levels of computation (by automatically plugging the lower level skeletons in the higher ones).

In the example described below, we consider a FEM simulator specification. This kind of simulator is capable of solving, for example, problems involving transient phenomena, where the phenomena context includes linear temperature-dependent elasticity, rigid body motion and linear heat transfer [2,6]. Only two blocks are needed in the present case. The number of Groups depends on the phenomena types present in a specific simulation. The number and type of phenomena depends as well on the simulation being carried out. In the ith-Block Skeleton Nig is its number of groups.

```
I.From Blocks i = 1 until 2
  I.0)Retrieve initial state for Block i
  I.I)Compute initial time step Δt_i for Block i
  I.II)Compute initial auxiliary data for Block i
II.Compute initial Δt = min_{1≤i≤2}{Δt_i}, set time instant t_1=0
III.While t_1 ≤ T_max do:
  III.0)Set t_0 = t_1 and t_1 = t_0 + Δ t
  III.I)For Block i = 1 until 2
    III.I.0)Solve for Block i
    III.I.I)Compute next time Δt_i for Block i
  III.II)Compute next time step Δ t = min_{1≤i≤n} {Δ t_i }
  III.III)Continue with time iteration
IV.End of the simulation
```

Figure 11 Global Algorim Skeleton

Figure 11 shows the Global Skeleton, while Figure 12 shows two Block Skeletons. Figure 14 and Figure 15 present the GIG direct acyclic graph to implement Global and Block Algorithm skeletons.

```
Is-B^i)Retrieve Initial State for Block i (see(I.0)):
Is-B^i.0)For r = 1 until Nig
 Is-B^i.0.0)Group r, compute phenomena initial states
It-B^i)Compute initial time step for Block i (see(I.I)):
It-B^i.0)For r = 1 until Nig
  It-B^i.0.0)Group r, compute Initial time step Δr
It-B^i.I)Set Δt^i = min_{1≤r≤Nig} {Δ_r }
```

Figure 12 Block Algorithm Skeletons

As it was already said, there should be *AlgthmData* objects, which will contain the needed problem and process data needed by each *AlgthmNode* object. A specialization of *AlgthmNode* is *AlgthmConnection*, which is defined whenever a lower level process is to be called up. Its *AlgthmData* object includes pieces of information needed in the identification of the lower level skeleton that will be plugged in the Algorithm Skeleton Graph. This identification concerns a driver *AlgthmNode* object (from another graph, integrating in this way the graphs presented in Figure 14 and Figure 15), which will substitute the related *AlgthmConnection* object.

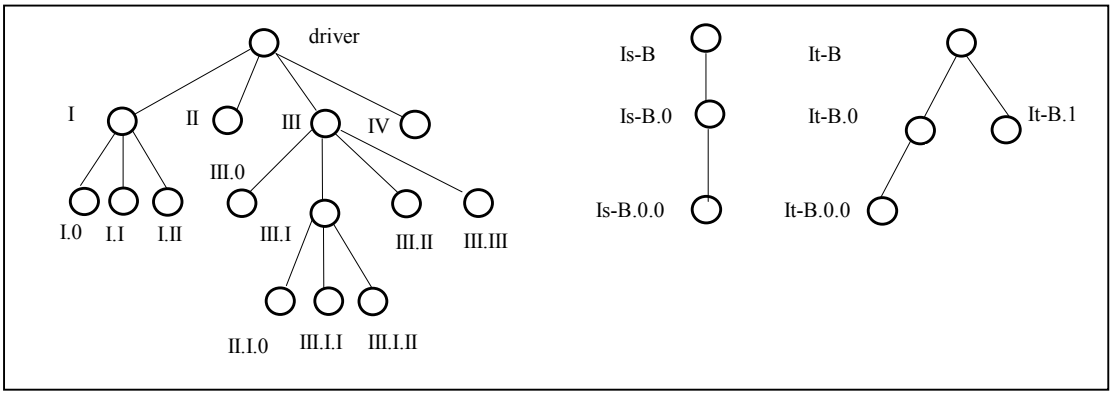


Figure 14. Global Alg.Skeleton graph

Figure.15 Block Alg.Skeletons graphs

12 Consequences

In what follows we make some considerations about the forces treated by the proposed pattern.
We can observe **positive forces** for the use of the GIG-pattern:

- Easiness of translating from algorithmic language into computer processes and simplicity in the process definition. It supports an organisation in a graph level, providing the distribution of code in a very flexible way, not compelling a rigid division of code. To improve simplicity in process definition we try to avoid unnecessary levels of details and to maintain similarity to the predefined algorithmic structure.
- Different users have evaluated this pattern with success, in applications with different levels of complexity. A simple example can be seen in section 10, and a more complex one in [6].
- Support for different levels of granularity of the defined processes. It allows a flexible representation for a mixture of scales, since it does not restrict the levels of programming into which the code is defined. Differently, in [1] the workflow must be defined in terms of a set of node types that are already been coded in the programming language level.
- It can be applied to any domain solution, through the definition of specific domain data classes and algorithms, as it can be seen from the pattern participants, in section 6.1,
- It allows the test of individual parts of the process independently, reducing the error occurrences in the coupling of processes.
- It allows the reuse of entire solutions, making changes in specific points. In GIG, it is easy to change parts of the graph, maintaining the other ones intact.
- It allows the graph change (that is, the process change) at run-time. This is achieved through the GIG intrinsic dynamic structure, as was shown in section 7.2. Data and process can be defined at run-time, depending on GIG implementation, once the pattern can be easily extended to incorporate design patterns like [9], as presented in section 9.

Some **negative forces**, or restrictions, can also be identified:

- The pattern makes severe restrictions on the graph structure, requiring it to be an acyclic graph. The designer is not allowed to define neither recursive iterations nor loops out of the node code.
- GIG-pattern makes no explicit reference or imposition for the use of a specific set of process types, differently from [1]. We can consider that this may cause loss of workflow-refined control. It is the programmer responsibility to define and manage this organization, if required by the application.
- Flow control is inside each node code. This can bring difficulties to some part of the process adaptation and control.
- Synchronization is not GIG-pattern responsibility. GIG does not define a specific structure to deal with process parallelism and processes synchronization. To allow the definition of processes parallelism, the programmer has to deal with extra complexity. GIG-pattern requires unnecessary levels of repetition, that is, the replication of whole process graph branches.

We may summarize saying that this pattern it is not so appropriate for applications that are simple and do not require exchangeable processes, modularity or articulation of sub-routines. Also it is inappropriate for applications where there is a need of a high level of refinement in the programs code, or if they need to process synchronization and parallelism; in these cases, an alternative is the use of the Micro-workflow proposal, described in [1]. However, through the use of the Micro-workflow alternative one of the worthy things you loss is simplicity and the level of granularity; the translation from algorithmic language is not such a direct mapping, losing in this way some levels of abstraction. The application of the GIG pattern to simplify application can be more expensive then a simple solution. On the other hand, it provides extra facilities like reuse, flexibility for new solutions, domain independence, etc.

13 Related patterns

The following patterns, can be used together with the GIG-pattern:

- *Factory Method* [10], which can be used to materialize objects for workflow management;
- *Template Method* [10], used to define skeletons of algorithms in *DataAlgthmServer* class;
- *Composite* [10], used to implement the *AlgthmNode* class functionality in the framework.
- *Proxy* [10], used in *AlgthmConnection* class;
- *Strategy* [10], *used in AlgorithmNode* and *AlgorithmData* classes

- Adaptive object-model patterns, such as *TypeObject* [9], shown in variants section.
- *FEM-SimulatorSkeleton* [2] achieves great benefit from the GIG approach.

14 Known uses

Many variations of numerical algorithms show the very same organizing structure, which was abstracted by the GIG-pattern. Some of these numerical algorithms are: mesh generation procedures, geometric reconstruction from planar slices and integration of geometric reconstruction procedures, and so on. Despite of being a generic solution that can be applied elsewhere, the users of this pattern have been scientists and engineers. The GIG-pattern has been applied with success in the development of different FEM simulator applications, and in a variety of other numerical methods in computational Mechanics. Other known uses, which applies GIG pattern, is a specific environment called Plexus, whose objective is the construction of FEM simulators for treating problems involving coupled multi-physic phenomena [2,6]. This environment applies GIG as a general solution for the numerical methods and articulation strategies for solving groups of phenomena [15]. Section 11 has given some details.

Acknowledgments

Thanks to Joe Yoder who served as the SugarLoaf PLoP shepherd.

References

[1] Manolescu D.A., "Micro-Workflow: A Workflow Architecture Supporting Compositional Object Oriented Software Development", Ph.D, Depart. of Computer Science University of Illinois at Urbana-Champaign, 2001**.**

[2] Lencastre M., Santos F., Rodrigues I., "FEM Simulator based on Skeletons for Coupled Phenomena", SugarloafPLoP'2002, Brazil.

[3] Lencastre M., Santos F., "FEM Simulation Environment for Coupled Multi-physics Phenomena". Simulation and Planning In High Autonomy Systems, AIS02, Portugal, 2002.

[4] Lencastre M., Santos F., Araújo J, "A Process Model for FEM Simulation Support Development" SCSC2002', Summer Computer Simulation Conference, California, 2002

[5] Lencastre, M., Santos F., Vieira M. "Workflow for Simulators based on Finite Element Method", Internat.Conference on Computational Science 2003 (ICCS 2003), Saint Petersburg, Russian, 2003.

[6] Lencastre M., Santos F., Vieira M., "A Case Study using GIG", submitted 2003.

[7] Workflow Management Coalition, "The Workflow Reference Model, Workflow Management Coalition Specification", - Winchester, Hampshire - UK, 95.

[8] Casati F., Fugini M.G, Mirbel, I. and Pernici, B., "WIRES: A methodology for developing Workflow Applications", Requirements Engineering (2002), volume 7, number 2, Editors P. Loucopoulos and J. Mylopoulos ISSN:0973602.

[9] Yoder J., Johnson R. "The Adaptive Object Model Architectural Style", Proceeding of The Working IEEE/IFIP Conference on Software Architecture (WICSA3'02), World Computer Congress in Montreal, 2002.

[10] Gamma E., Helm R., Johnson R., Vlissides J., "Design Patterns: Elements of Reusable Object-Oriented Software" Addison-Wesley, 1995.

[11] Cormen T., Leiserson C., Rivest R., Stein C., "Introduction to Algorithms" second edition, MIT Press, 2001.

[12] Stroustrup, B. "The C++ Programming Language", third edition, Addison-Wesley, 1997.

[13] Fayad M., Douglas S. Johnson R., "Building Application Frameworks: Object-Oriented Foundations of Framework Design", Wiley Computer Publishing, 1999.

[14] "Software Architecture System Design, Development and Maintenance" Edited by Jan Bosch, Morven Gentleman, Christine Hofmeister, and Juha Kuusela; Kluwer Academic Publishers 2002.

[15] Lencastre M, "PLEXUS - A domain specific approach for FEM simulators development", a PhD thesis being developed in Federal University of Pernambuco, Brazil, 2003.

Adaptability Aspects: An Architectural Pattern for Structuring Adaptive Applications with Aspects

Ayla Dantas* and Paulo Borba†
Informatics Center – Federal University of Pernambuco
Po Box 7851 - 50.732-970 Recife, PE - Brazil
{add,phmb}@cin.ufpe.br

Abstract

This paper presents an architectural pattern for structuring adaptive applications using aspect-oriented programming in order to obtain separation of concerns. It is composed of known and novel patterns organized so as to provide good maintainability and modularity. [1]

1 Intent

This architectural pattern is intended to show how to use aspects [14] in order to better structure adaptive applications, which are able to change their behavior in response to context changes [11], such as the device new localization or its resources state.

2 Context

Adaptability has became a common requirement [9] and its implementation usually affects many parts of the code. Most implementations of this requirement lead to tangled code, mixing different concerns such as application business rules, GUI code, and adaptive behavior implementation. It is sometimes hard to include the adaptability concern in new and existing applications in easily maintainable way. Besides that, the mechanisms for accessing contextual information, such as environment sensors or analyzers, change frequently, which usually demands regular modifications to the application that can be hard to do. This is a problem because contextual information obtained by new sources may lead to different application behaviors. For example, we can have a device that becomes able of accessing its temperature, and according to it changes its processing mode.

*Supported by CNPq.

†Partially supported by CNPq, grant 521994/96–9.

[1] We followed the POSA (Pattern-Oriented Software Architecture) structure to present our pattern, including an Intent section and distributing the contents of sections to be called *Variants* and *Resolved Example* throughout the pattern.

3 Problem

Avoid lack of flexibility and code tangling mixing adaptive functionality implementation with different concerns such as application business rules and GUI code.

4 Forces

In order to solve the problem, *Adaptability Aspects* balances the following forces:

- Separate adaptability concerns from other concerns, in order to improve reuse and maintainability.
- The adaptability functionality might be either plugged in/out and also turned on/off, because the ways to acess the context and the behaviors triggered by environment state may change a lot.
- Not all the developers need to know a particular Aspect-Oriented Programming (AOP) language, because each team may be focused on a different pattern module, that does not necessarily need AOP.
- The application should be easy to maintain.
- The application can be implemented in any platform, from embedded devices, such as cellular phones, to enterprise applications.
- The kind of contextual information might change and this should not cause a significant impact on the system.

5 Solution

Use aspects [14] to make the application adaptive in a modular and not invasive way. With Aspect-Oriented Programming we can cleanly capture some implementation aspects that affect many parts of a system, as it happens with adaptability, providing appropriate isolation, composition and reuse of the code used to implement those aspects [8].

By using aspects in Adaptability Aspects we define how the behavior of the application functionalities should be changed in order to support adaptability. These aspects are able to crosscut some application execution points and change their normal execution. They should interact with a module for monitoring the context in order to identify when a change must be triggered and they use auxiliary classes to perform the application changes. For a better flexibility, the auxiliary classes should interact with a module responsible for obtaining dynamic data specifying how the application should adapt in a specific situation. Auxiliary classes are used to avoid requiring all developers to know an AOP language and to avoid code tangling in the adaptive behavior implementation.

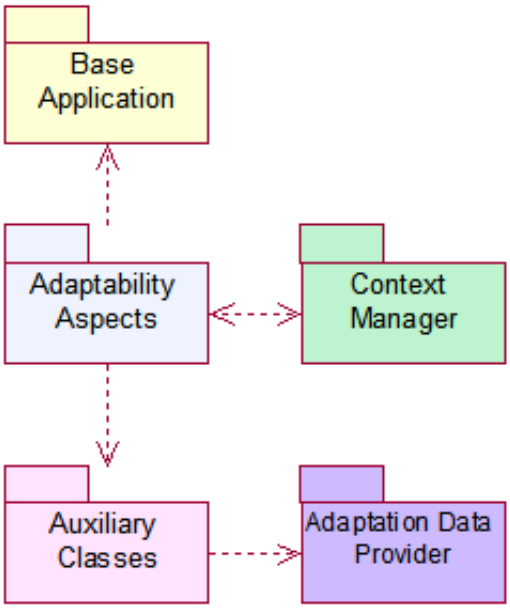


Figure 1: *Adaptability Aspects* elements.

6 Structure

The *Adaptability Aspects* architectural pattern presents five elements or modules:

- **Base Application**: The core application functionalities, such as business and GUI code, and possibly persistence and distribution code, but no adaptability code.
- **Adaptability Aspects**: The aspects implementing the adaptability concern. They specify how the behavior of the base application functionalities should be changed to adapt to contextual changes. This element delegates several tasks to the auxiliary classes.
- **Auxiliary Classes**: Classes used by the aspects to provide the adaptive behavior. Its isolation from the aspect is intended to improve reuse. They communicate with the adaptation data provider module in order to obtain dynamic data for the adaptation. Besides that, for developing the auxiliary classes, the developers of this module do not necessarily need to know an AOP language. The Adaptability Aspects developer or the system architect may simply specify the interfaces of these classes and what they should do. Then, from these specifications, these classes can be built and have their methods invoked by the aspects.
- **Context Manager**: Module responsible for analyzing context changes and triggering adaptive actions implemented by the aspects. It can also be called by the aspects to obtain information about the context. Its implementation can be based on a variation of the Observer pattern [4], or on its implementation with aspects [6]. In this way, new mechanisms for accessing the context can be easily supported without significant impact on the application (see the Example section).

- **Adaptation Data Provider**: Classes responsible for providing data for dynamic adaptations according to context changes. This means that the same context change can lead to different behaviors in different moments according to the data provided by this module. These classes can be organized as an Adaptive Object-Model (AOM) [16].

These elements and their inter-relation are shown in Figure 1. They are represented there using the UML package notation. Each package represents a logical part of the code, but each of these parts can be implemented using several programming language-specific packages. The arrows represent the dependency between the packages. It is important to notice that the relation between *Context Manager* and *Adaptability Aspects* is double-way but this does not imply in internal packages double-way relations. Sometimes a *Context Manager* package uses an *Adaptability Aspects* package to notify about context changes or an *Adaptability Aspects* package may use a *Context Manager* to request some information about the context in a certain execution point. Another important observation is that we have a different kind of dependency between *Base Application* and *Adaptability Aspects* because the latter is capable of crosscuting the former.

7 Dynamics

The following scenarios depict the dynamic behavior of the *Adaptability Aspects* pattern.

Scenario I, which is illustrated by Figure 2, presents the application behavior when the aspects request information about the context at specific points in the execution flow and the application changes its behavior at those points according to the response obtained:

- The application starts executing one of its core functions.
- The adaptability aspects detect points in the execution flow (join points) of the core function where an adaptation might be performed (secure points). This is implemented using AOP constructs such as pointcuts, which are a way of identifying join points. Then, these aspects lead to behavior changes in the application before, after or around those points using advice.
- Those aspects then query the context manager to verify whether any adaptation should be performed, which will depend on the environment state.
- The actions implementing the adaptation are then delegated to the auxiliary classes. These actions are performed before, after, or around the join points.
- In order to access dynamic data specifying how an adaptation should be performed, the auxiliary classes access the objects of the adaptation data provider, which can be based on the Metadata and Active Object-Model pattern language [3], also called Adaptive Object-Model Architecture [15].
- The application is then changed according to this dynamic data.

The sequence diagram shown in Figure 2 shows the interaction between the pattern elements. Instead of representing objects, each diagram box represents a collection of objects from the correspondent pattern module.

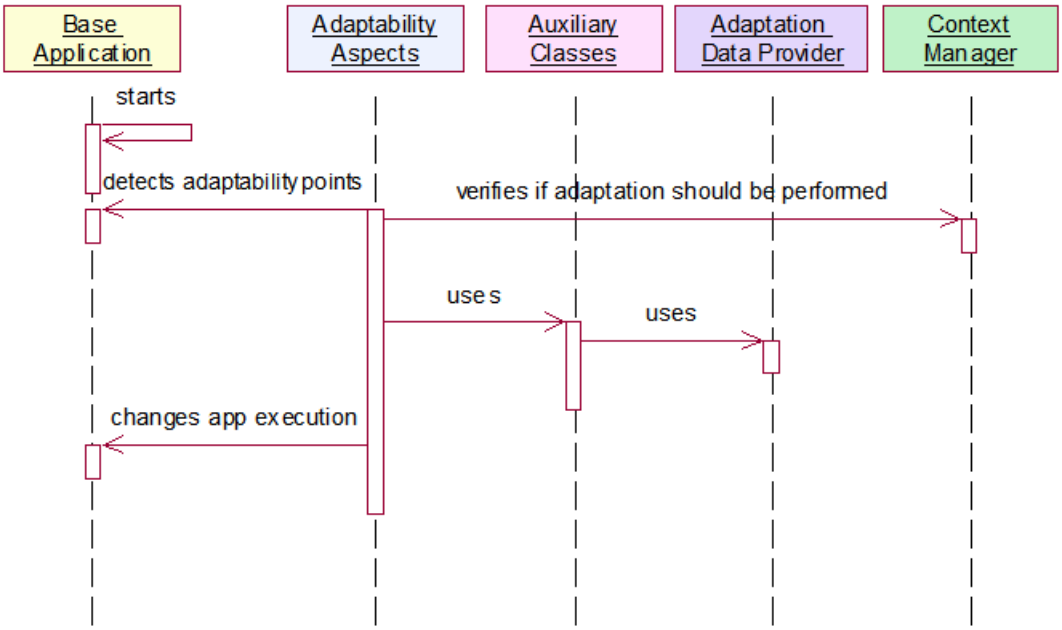


Figure 2: Dynamics of *Adaptability Aspects* pattern (scenario I)

Scenario II, illustrated by Figure 3, shows how the elements of the *Adaptability Aspects* pattern interact when a context change triggers an adaptation on the application behavior.

- The application starts execution.
- The context manager begins to monitor the context continuously.
- When a registered environment change is detected, the adaptability aspects are notified.
- The adaptability aspects detect a point in the execution flow and then use auxiliary classes to perform the changes in the application.
- The auxiliary classes use the adaptation data provider in order to access dynamic information on how the adaptation should be performed.
- The aspects change the application behavior.

When the adaptability aspects are notified about a context change, the action to be performed can execute either immediately, as for example, an invocation to the garbage collector or not, as it is more usual. In the latter case, a field representing the adaptation state can be directly set, but the adaptive behavior is introduced on the base application just when a specific join point is reached.

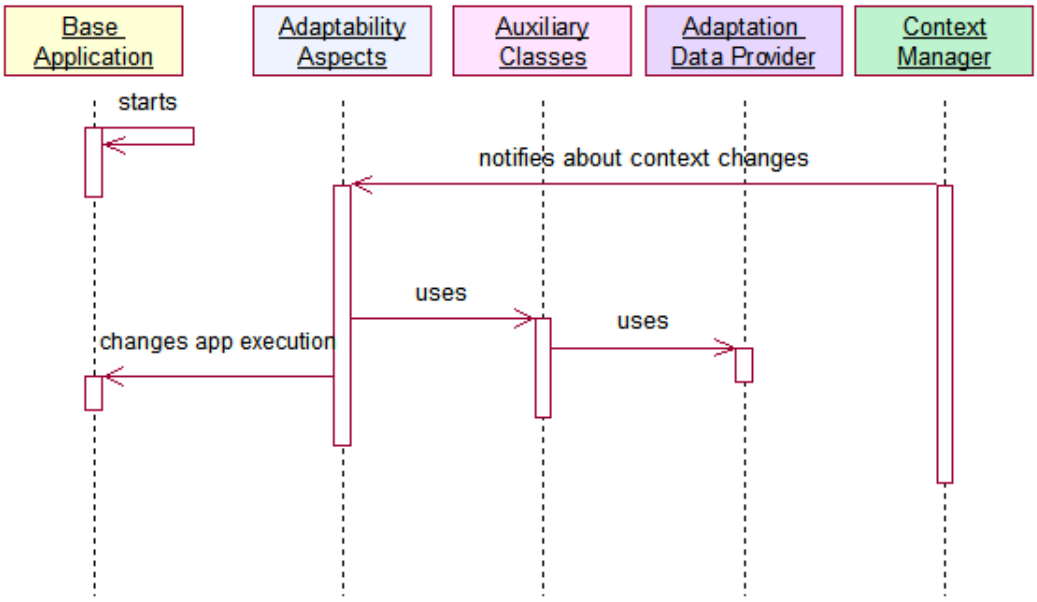


Figure 3: Dynamics of the *Adaptability Aspects* pattern (scenario II)

8 Consequences

The *Adaptability Aspects* pattern provides the following benefits:

- *Modularity.* The base application is isolated from the adaptability aspects and from the classes that actually execute actions at the identified adaptation points.
- *Reuse.* The use of auxiliary classes improves reuse, since these classes can be used by many aspects. Besides that, as will be seen on the implementation section, the Adaptation Data Provider and Context Manager modules should also be organized in order to improve reuse by other adaptive applications.
- *Extensibility.* As the aspects code is isolated from the core and auxiliary classes code, it becomes easier to maintain each part of the application and also to extend the system. The context manager should also be internally organized to promote extensibility, making it simpler to achieve and minimizing the impacts caused by the inclusion of a new context element. Nevertheless, the use of AOP in some modules must be done carefully and the use of visual tools is strongly recommended, because AOP languages are considerably powerful, and a change in one part of its code can affect the entire system.
- *Platform Independence.* Its general structure can be applied to a wide range of systems, from embedded systems (such as the example we will present shortly) to enterprise applications. This follows from it not imposing a burden on efficiency and not requiring a significant amount of resources.

- *Dynamic changes.* With the adaptation data provider module, dynamic changes in the application behavior can be performed, and those changes do not need to be expressed by new code; they may be expressed in metadata (XML files, for example).

The *Adaptability Aspects* architectural pattern also has some liabilities:

- *Code size.* Implementations of the pattern are usually bigger than alternative non-modular solutions that tangle adaptation code with base application code, as we can observe from design and based on our implementations. The implementations proposed on the Implementation section for this architectural pattern can also increase the code size if more flexibility, dynamicity, and reuse are required. This liability can be a problem for embedded systems, in which resources are restricted, but for enterprise applications it would not be an issue.

- *Efficiency.* Due to the number of elements required by the architecture, the proposed pattern imposes a burden on efficiency when compared to other non-modular solutions. Nevertheless, such burden would be a minor problem for an enterprise architecture. Besides that, some efficiency problems may be related to the used AOP language.

- *Dynamic loading.* An important requirement related to adaptability nowadays is dynamicity (the extent to which the adaptation should be dynamic). Our architectural pattern provides dynamic changes using the adaptation data provider element, which presents new application behaviors in metadata. However, it may not be possible to dynamically load code in order to include new kinds of adaptation mechanisms (ways of triggering new adaptations) without stopping the application, which will depend on the application platform and on the AOP language being used. AspectJ [7], an AOP language in widespread use, does not allow dynamic loading of new aspects in its latest stable version.

- *New programming paradigm.* As we have at least one module in the pattern that uses aspects, some developers of the team need to know Aspect-Oriented Programming. Learning this new programming paradigm may lead to more development time, initially. However, for other modules, or even for some parts of a module, knowing a particular AOP language is not required.

9 Implementation

The following guidelines help implementing an adaptive system using the *Adaptability Aspects* pattern.

1. *Develop the base application.* The application is developed with its main functionalities, preferably organized in a way to help its evolution. If the application has already been developed, it is occasionally suggested to do some refactoring in order to provide internal modularity to this module.

2. *Develop the context management module.* This module must present ways of detecting which elements from the context should change, which elements should be

informed when these changes occur, and which actions should take place at those moments. We propose the implementation of this module using the *Observer* pattern, and specially its implementation using aspects, proposed in another work [6], and adapted to the observation of the context, which increases the flexibility of this module in relation to new environment elements to be observed. This is illustrated by Figure 4. Although this figure illustrates the Context Manager module, the `SpecificAdaptabilityAspect` is part of the Adaptability Aspects part and is shown here just to illustrate the interaction between these pattern elements. The `SpecificAdaptabilityAspect` and `SpecificAdaptationProtocol` aspects and the `SpecificContextElementVerifier` class of this figure are used to represent the elements used to manage a specific context change. For example, if we want a different application behavior, such as the changing of languages used for translation in a dictionary, when the device localization changes, we would replace these classes by the following ones respectively: `DictionaryCustomization`, `LanguageAdaptationProtocol` and `LocalizationVerifier`. Figure 7, in the next section, better illustrates this example and the inter-relationship among the pattern elements in this situation.

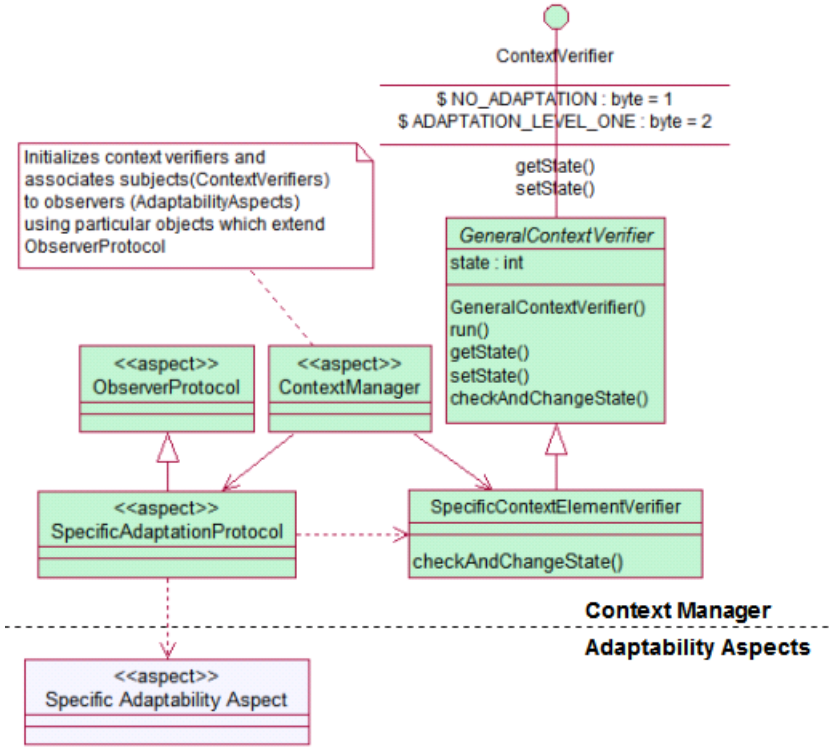


Figure 4: Context Manager Possible Implementation

3. *Develop the adaptation data provider module.* Dynamic adaptation is generally desirable. It is not good practice to fix the adaptations in code. One option is to have these changes in metadata such as XML files that can be obtained locally or remotely (the Bridge [4] pattern is indicated for providing this flexibility). Such an idea is adopted by the Adaptive Object-Model architecture [16]. The use of AOM

is not mandatory for the development of this module, but presents the advantage that, once implemented, one of its parts can be reused by many systems; only the interpreter of this AOM remains to be implemented according to the wanted changes, which is a task that can be performed by auxiliary classes. An AOM partial implementation that can be reused by many kinds of adaptive applications is shown in Figure 5. The application elements can be represented as `EntityType` elements, which present `Entities` with their `Properties` or `StrategyObjects` to represent different system behaviors. The interpreter of the AOM will translate these kinds of elements into application properties or algorithms. The auxiliary classes used by the adaptability aspects will have access to the adaptation data provider module by the `AppAOMManager` class, using its `getEntityType` method. An alternative implementation for the adaptation data provider module may explore the idea presented in another work [2], using an application-specific object responsible for obtaining dynamic data for adaptations and making them available through its operations. However, with this approach, besides losing in reuse, for each new kind of data, we would have to modify this object hierarchy, and the number of operations could increase a lot.

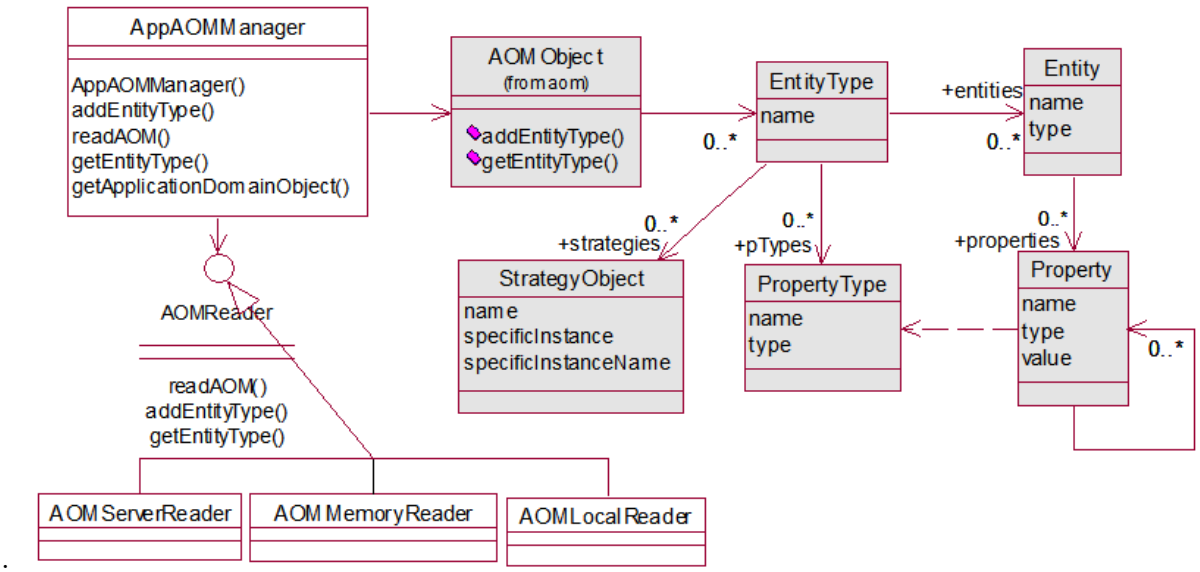


Figure 5: **Adaptation Data Provider** using AOM

4. *Identify which kind of adaptations will be included.* It is necessary to analyze what will really constitute an adaptation (an application change caused by the environment) and which context changes would lead to those adaptations. Example adaptations are changes in the application language due to the detection of a device location modification, memory management, changes and inclusion of screens in the application due to user inputs or server responses, for example.

5. *Transform each kind of adaptation into an aspect.* Use AOP constructions such as pointcuts and advice to access the application components that can change and the execution points (join points) where the adaptations should be performed. These

aspects should contain the least amount of non-aspect code as possible, delegating necessary actions to auxiliary classes that will be developed in a later step. Besides that, these classes should be able to access the context manager module in order to verify whether an adaptation should be done. Each aspect can also be a `SpecificAdaptabilityAspect` in the situation illustrated by Figure 4.

6. *Develop the auxiliary classes.* These classes will actually present the code to be executed at the points selected in the aspects module. Their design should promote reuse by many aspects. In order to perform dynamic adaptations, this module should communicate with the adaptation data provider module. One suggestion is to include in these classes interpreters of AOM for the application being developed.

10 Example

To exemplify the *Adaptability Aspects* pattern, and specially its second scenario, we will now consider a dictionary application developed in Java [5], for the J2ME (Java 2 Micro Edition) platform. This platform accommodates consumer electronics and embedded devices [12]. The AOP language used is AspectJ [7], a general purpose aspect-oriented extension to Java in widespread use.

Our **Base Application** will be composed by this dictionary, which is a simple MIDP (Mobile Information Device Profile)-based application (also known as MIDlet [12]), without any adaptations. Its basic functionality is to translate a given word from English into Portuguese. It follows the MVC [1] architectural pattern and its main classes are illustrated by Figure 6: the controller part is being represented by the `DictionaryMIDlet` and the `DictionaryController`; the view part is represented by the `DictionaryScreen`; the model is represented by the `InputSearchData` class, which stores some information about the dictionary, such as the word being translated and the source and destination languages of the translation.

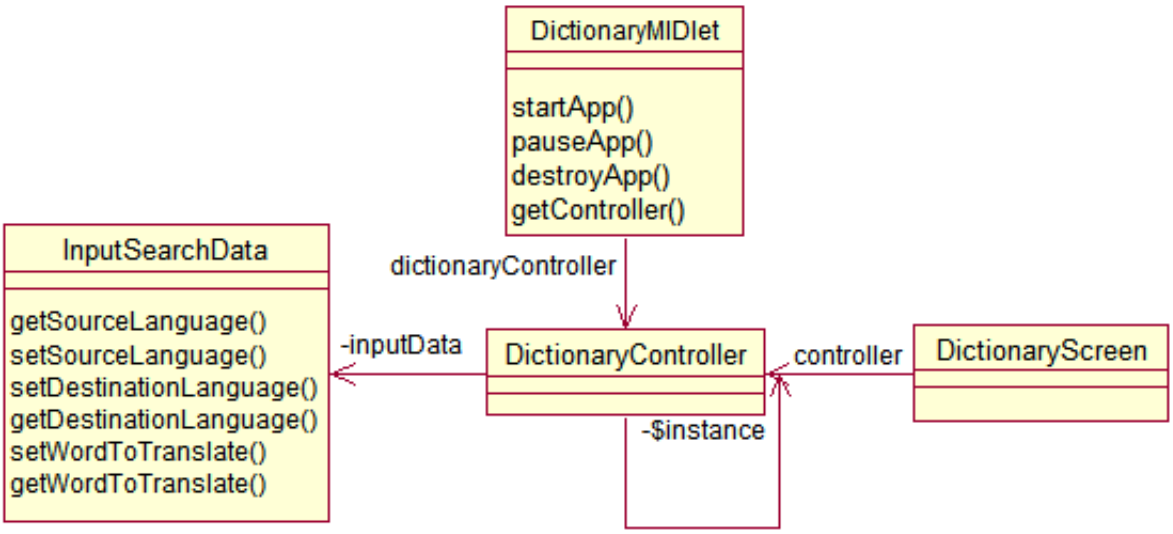


Figure 6: A **Base Application** example

In order to make the example easier to understand, the adaptation we will include will be the change of the application translation language when the device location changes. The main classes and aspects used for this adaptation are illustrated by Figure 7, which is a UML diagram that uses the << `aspect` >> stereotype to identify aspects. Other stereotypes are also used in some dependency relationships to represent classes whose behavior is monitored or changed by an aspect (<< `affects` >> stereotype), or those which are just used as auxiliary classes (<< `uses` >> stereotype) by an aspect.

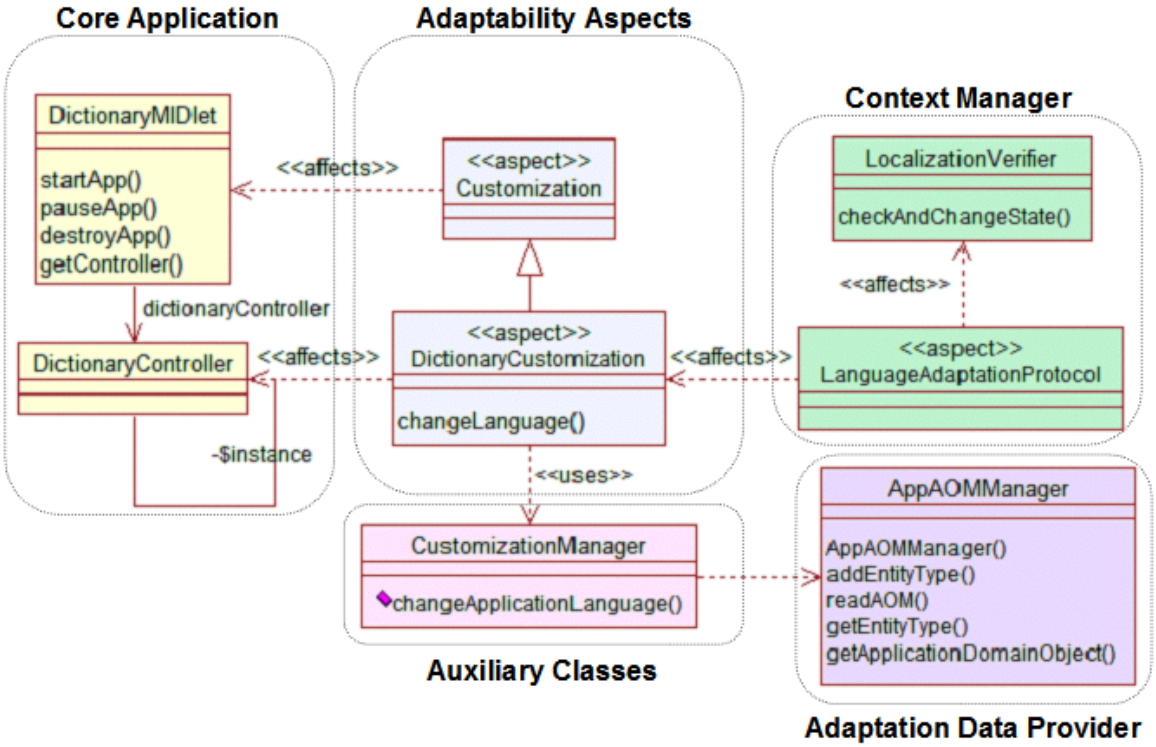


Figure 7: An **Adaptability Aspects** pattern example

The aspects which will compose the **Adaptability Aspects** element of the pattern are organized in a small framework, composed of two aspects: the `Customization` aspect (which can be reused by any MIDP-based J2ME application) and the `DictionaryCustomization` aspect (which is application specific and extends the former). They are responsible for changing application properties, using a captured `MIDlet` instance at the moment the `startApp` method is executed. The following code from the `Customization` aspect illustrates that.

```
pointcut MIDletStart(MIDlet midlet):
  execution(void startApp()) && target(midlet);
before(MIDlet midlet): MIDletStart(midlet) {
  adaptBefore(midlet);
}
```

The `Customization` aspect is general because any MIDP-based J2ME application should present a class implementing `MIDlet` with a `startApp` method. Its `before` advice invokes the `adaptBefore` abstract method, which is defined in the `DictionaryCustomization` aspect. At its invocation, this aspect stores the `MIDlet` instance, which makes it able of changing application properties in its methods. The interaction between these aspects and the application classes is illustrated by Figure 7.

The **Context Manager** module will follow the structure shown in Figure 4. The code for the `ObserverProtocol` is shown elsewhere [6]. Our `SpecificAdaptationProtocol` aspect will be the `LanguageAdaptationProtocol`, which is shown below:

```
 1: public aspect LanguageAdaptationProtocol extends ObserverProtocol {
 2:   declare parents: LocalizationVerifier implements Subject;
 3:   declare parents: DictionaryCustomization implements
 4:     Observer;
 5:   protected pointcut subjectChange(Subject s):
 6:     call(void LocalizationVerifier.setState(int)) && target(s);
 7:   protected void updateObserver(Subject s, Observer o) {
 8:     LocalizationVerifier lv = (LocalizationVerifier) s;
 9:     if (lv.getState()!=ContextVerifier.NO_ADAPTATION) {
10:       ((DictionaryCustomization) o).changeLanguage();
11:       ((LocalizationVerifier)
12:         s).setState(ContextVerifier.NO_ADAPTATION);
13:     }
14:   }
15:}
```

In this aspect we identify the `Subject` (`LocalizationVerifier`) and the `Observer` (`DictionaryCustomization` aspect) of the pattern using AspectJ introduction (lines 2 and 3). Then, we define the pointcut `subjectChange`, inherited from `ObserverProtocol`, identifying the execution points that characterize the `Subject` change (lines 5 and 6), which is the call of the `setState` method on a `LocalizationVerifier` object in this example. Finally, we define the also inherited method, `updateObserver`, indicating what should happen to the `Subject` and the `Observer` when there is a change (lines 7-14). In this case, a method from the `DictionaryCustomization` aspect is invoked to change the application language. We will talk about this method later, while describing our **Auxiliary Classes** module use.

The `GeneralContexVerifier` from Figure 4 extends the `Thread` class and invokes an abstract method called `checkAndChangeState` at regular intervals. This class can be reused by other applications. The `LocalizationContextVerifier` extends the `GeneralContextVerifier` and should only specify its abstract method, verifying the device localization as the following code illustrates.

```
private LocalizationObject lo = new LocalizationObject();
public void checkAndChangeState() {
  if (lo.hasChanged())
    this.setState(ContextVerifier.ADAPTATION_LEVEL_ONE);
}
```

We use here an abstraction of a `LocalizationObject`, which is responsible for verifying if the device localization has changed. If this happens, the `setState` method is called, the `LanguageAdaptationProtocol` aspect identifies that the `Observer` should be notified and executes the `updateObserver` method, which will call the `changeApplicationLanguage` method from the `DictionaryCustomization` aspect.

As Figure 4 illustrates, the `ContextManager` initializes the context verifiers and associates subjects to observers. The following code extracted from it shows how it works for this example.

```
public aspect ContextManager {
  pointcut MIDletStart(MIDlet midlet): execution(void startApp()) &&
    target(midlet);
  after(MIDlet midlet): MIDletStart(midlet){
    LocalizationVerifier lv = new LocalizationVerifier();
    lv.start();
    LanguageAdaptationProtocol.aspectOf().addObserver(lv,
      DictionaryCustomization.aspectOf());
...
  }
}
```

In order to avoid non-aspect code in aspects, the `changeLanguage` method from the `DictionaryCustomization` aspect delegates the application language change to an auxiliary class, the `CustomizationManager`, which will be part of the **Auxiliary Classes** element of the *Adaptability Aspects* pattern. Nevertheless, this element is not mandatory and another possible technique would be to use non-aspect code in methods of the aspect and not in the advice body when extensive code is necessary to provide the change. However, to illustrate the **Auxiliary Classes** module use and its interaction with **Adaptation Data Provider** module (see Figure 5), we will show one of the `CustomizationManager` methods implementation, the `changeApplicationLanguage`:

```
public void changeApplicationLanguage(){
  EntityType et = aomManager.getEntityType("DictionaryProperties");
  Entity e = et.getEntity("InputSearchData");
  Property sourceLang = e.getProperty("sourceLanguage");
  Property destLang   = e.getProperty("destinationLanguage");
  InputSearchData isd = this.midlet.getController().getInputData();
  isd.setDestinationLanguage(destLang.getValue().toString());
  isd.setSourceLanguage(sourceLang.getValue().toString());
}
```

In this code we can notice the relation of the **Auxiliary Classes** component with the **Adaptation Data Provider** module. Here, this module uses the *Adaptive Object-Model* architectural style. As we can see, the application properties and behavior, described in metadata (in our case an XML file), are translated into the object structure illustrated by Figure 5. An UML object diagram of a possible configuration for this code is illustrated by Figure 8. The `changeApplicationLanguage` method then interprets this object model

and changes the application using the `MIDlet` object and modifying the `InputSearchData` instance. This is just a simple example of AOM's use. By using it, better exploring the patterns it is related to, we can do much more things that would lead to more flexibility.

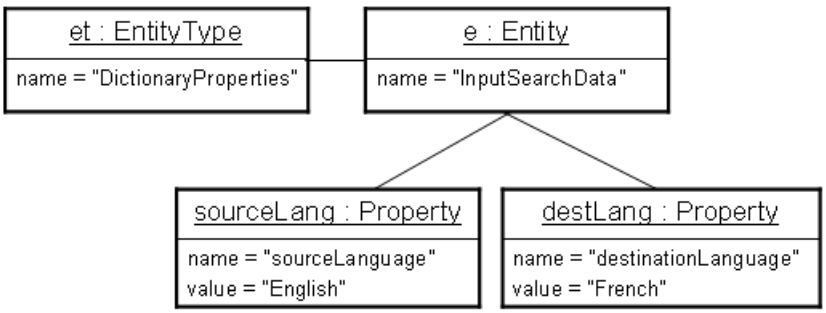


Figure 8: Entity configuration example

11 Known Uses

The *Adaptability Aspects* architectural pattern has been used in two experiments we have developed with J2ME: a Dictionary application for a cellular phone and an album for pictures, also for small devices. Besides that, some parts of the patterns have already been referred to. An example of this is a work [10] presenting some experiments using aspects. One of the conclusions is that it is an interesting architecture to separate an AOP system in three parts: the base code, the aspects and the auxiliary code. In this structure, the aspects part functions as glue between the other two. Another example is the implementation of the Observer aspect using aspects proposed by Hanneman [6] which we modify here in the **Context Manager** module.

12 See Also

- The *Reflection* architectural pattern [1], which provides a mechanism for changing structure and behavior of software systems dynamically, is related to our pattern. It is intended to make applications adaptable, that means, able to easily evolve due to requirement changes, which is also one of our requirements. Although it provides a lot of flexibility, it seems to increase the complexity of the system more than our solution and even to be less efficient.

- *Adaptive Object-Model* systems represent their attributes, classes, and relationships as metadata [15]. AOMs can be used to represent all the system structure or just what changes in the system. In our pattern, we took the latter approach and explored AOP as a non-invasive way to include AOMs. We have used AOMs in the **Adaptation Data Provider** module as the provider of the adaptive behavior data.

- The *PADA(Pattern for Distribution Aspects)* [13], even dealing with Distribution, is also related to this work as it provides a solution to the lack of modularity and

maintainability using AOP.

- The *Observer* pattern [4] is very useful in the implementation of the **Context Manager** component (see Figure 4), which actually characterizes the adaptive behavior. Its implementation with aspects [6] is interesting as a dynamic way (using pointcuts) of identifying subject changes.

- Another work [2] describes some practices incorporated to this pattern and presents the implementation of some adaptive concerns into J2ME applications using AspectJ, comparing them to pure Object-Oriented solutions using GoF patterns. The implementation shown there does not exactly follow this pattern and does not use Adaptive Object-Models. Instead of that, it uses a less flexible approach for obtaining dynamic data for adaptations.

13 Acknowledgments

We would like to give special thanks to Paulo Masiero, our shepherd, for his important comments and suggestions, helping us to improve our pattern. Thanks to Rossana Andrade for her work as mediator and her attention, and also to Vander Alves, for his valuable suggestions. Besides them, we also want to thank the conference participants for the suggestions made at the Writers' Workshop, and specially Joseph Yoder, for his comments and ideas even after the SugarLoafPLoP.

References

[1] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *A System of Patterns: Pattern-Oriented Software Architecture*. John Wiley & Sons, 1996.

[2] Ayla Dantas and Paulo Borba. Developing adaptive J2ME Applications Using AspectJ. In *Proceedings of the 7th Brazilian Symposium on Programming Languages*, pages 226–242, May 2003.

[3] Brian Foote and Joseph Yoder. Metadata and Active Object-Models. Collected papers from the PLoP '98 and EuroPLoP '98 Conference Technical Report wucs-98-25, Dept. of Computer Science, Washington University, September 1998.

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[5] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, second edition, 2000.

[6] Jan Hannemann and Gregor Kiczales. Design Pattern Implementation in Java and AspectJ. In *Proceedings of the 17th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 161–173. ACM Press, 2002.

[7] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. Getting Started with AspectJ. *Communications of the ACM*, 44(10):59–65, 2001.

[8] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect–Oriented Programming. In *European Conference on Object–Oriented Programming, ECOOP'97*, LNCS 1241, pages 220–242, Finland, June 1997. Springer–Verlag.

[9] Kalle Lyytinen and Youngjin Yoo. Issues and Challenges in Ubiquitous Computing: Introduction. *Communications of the ACM*, 45(12):62–65, 2002.

[10] Gail C. Murphy, Robert J. Walker, Elisa L. A. Baniassad, Martin P. Robillard, Albert Lai, and Mik A. Kersten Kersten. Does Aspect-Oriented Programming Work? *Communications of the ACM*, 44(10):75–77, 2001.

[11] Peyman Oreizy, Michael M. Gorlick, Richard N. Taylor, Dennis Heimbigner, Gregory Johnson, Nenad Medvidovic, Alex Quilici, David S. Rosenblum, and Alexander L. Wolf. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14(3):54–62, May 1999.

[12] Vartan Piroumian. *Wireless J2ME Platform Programming*. Sun Microsystems Press, 2002.

[13] Sérgio Soares and Paulo Borba. PaDA: A Pattern for Distribution Aspects. In *Second Latin American Conference on Pattern Languages of Programming, SugarLoafPLoP'2002*, pages 87–100, Itaipava, Brazil, 5th–7th August 2002.

[14] Robert J. Walker, Elisa L. A. Baniassad, and Gail C. Murphy. An Initial Assessment of Aspect-Oriented Programming. In *Proceedings of the 21st International Conference on Software Engineering*, pages 120–130. IEEE Computer Society Press, 1999.

[15] Joseph W. Yoder, Federico Balaguer, and Ralph Johnson. Architecture and Design of Adaptive Object-Models. *ACM SIGPLAN Notices*, 36(12):50–60, 2001.

[16] Joseph W. Yoder and Ralph Johnson. The Adaptive Object-Model Architectural Style. In *Working IEEE/IFIP Conference on Software Architecture 2002(WICSA)*, Montréal, Québec, Canada, August 25-31 2002.

Un patrón para lexicones de patrones

Alan Calderón Castro
Escuela de Computación e Informática – Universidad de Costa Rica (UCR)
Fax 506-207-5527 – San José – Costa Rica
calderon@ecci.ucr.ac.cr

Resumen

A la fecha se han publicado muchos catálogos de patrones asociados con el desarrollo de software (patrones de diseño, patrones de diseño arquitectónico de sistemas, patrones de modelado de requerimientos, modismos o estilos de programación—conocidos como "idioms" en inglés—y otros). El conocimiento de patrones crece día a día y es esencial su incorporación a la enseñanza de la ingeniería de software y a la industria del software. Se requiere un sistema que permita organizar el conocimiento de patrones de manera natural que facilite las búsquedas, la organización y el intercambio de conocimiento de patrones, de variantes de patrones y combinaciones de patrones, así como la continua aparición de patrones para los dominios de experiencia existentes y otros nuevos. Visto desde otro ángulo, se requiere un sistema que soporte el desarrollo evolutivo de lenguajes de patrones y mejore su expresividad. En este artículo se propone un patrón para construir sistemas que soporten la evolución de lenguajes de patrones. Este patrón intenta proveer principios para construir sistemas de software que categoricen y sistematicen el conocimiento de diseño de la industria del software y de las organizaciones dedicadas a la enseñanza de la ingeniería de software. En [5] se ejemplifica la aplicación del patrón propuesto.

Abstract

Presently many pattern catalogs for software development have been published (design patterns, architecture patterns, analysis patterns, idioms and other). The pattern knowledge keeps growing day by day and it is essential its incorporation to software engineering teaching. On the other hand, the software industry also requires to systematize pattern knowledge. A system is required that allows to organize the pattern knowledge in a natural way that facilitates searches, the organization of pattern, pattern variants and pattern combinations, as well as the continuous appearance of new patterns for existent domains of experience and new ones. Seen from another angle, a system to support pattern languages evolution and to improve their expressiveness is required. To be consequent with the heuristic focus accustomed by those who specify patterns, this paper describes a pattern aimed at building systems to support pattern language evolution. Some principles to construct these type of systems are organized in a pattern form. An example of this pattern is described in [5].

1 Nombre del Patrón: Lexicón de Patrones

2 Contexto

A la fecha se han publicado muchos catálogos de patrones asociados con el desarrollo de software. Podría comenzarse citando "Design Patterns" [13], denominado catálogo de GoF (por "Gang of Four" como se referencia a sus cuatro autores), que organiza el conocimiento heurístico a nivel de diseño detallado de objetos.

Seguidamente podrían mencionarse las ediciones 1 y 2 de “Pattern-Oriented Software Architecture”, de Buschmann y otros la primera, y de Schmidt y otros la segunda (ver [4] y [19]), que enfatizan en la organización del conocimiento heurístico a nivel de diseño arquitectónico. Luego, se debería agregar “Data Model Patterns” [14], “Object Models” [7] que organizan conocimiento heurístico a nivel de modelado de requerimientos en torno a distintas familias de aplicaciones. Finalmente, no pueden dejar de mencionarse “Core J2EE Patterns” [2], "Advanced C++" [8] y otros, que organizan conocimiento heurístico a nivel de programación y por ende se refieren al uso óptimo de tecnologías de programación específicas. Esto solo para mencionar algunos de los catálogos más conocidos, porque el estudio [10] identifica tres grandes categorías de documentos publicados que incluyen patrones: una que abarca las actividades típicas del proceso de desarrollo de software (modelado de requerimientos, diseño, programación, prueba, etc.), otra que abarca a patrones del proceso de desarrollo de software y a patrones de organizaciones de desarrollo de software y una tercera categoría con otros tipos de patrones.

Se puede observar en algunos de estos catálogos una estructura de categorización recurrente orientada a facilitar la resolución de problemas de diseño. Los catálogos más reconocidos incluyen mapas de patrones que muestran cómo los patrones se complementan entre sí, y también recomendaciones generales para la guía del lector. Por estas características, el catálogo de GoF ha sido considerado por muchos autores como un ejemplo prototípico . Sus características esenciales son:

1. Categorización de patrones en: patrones de estructuración, patrones de comportamiento y patrones de construcción de objetos o patrones construccionistas .
2. A cada patrón se le asigna un nombre significativo.
3. El uso de una plantilla derivada de la tripleta “Contexto”, “Problema” y “Solución” propuesta en [1], para la especificación de cada patrón.
4. Como parte de la especificación de cada patrón se incluyen secciones para analizar las consecuencias de la aplicación de la solución y para hacer explícitas las relaciones con los demás patrones del catálogo.
5. Mapa de conexiones entre patrones.

Estas características organizativas esenciales también están presentes en otros catálogos, como por ejemplo [4] conocido como POSA 1, [19] conocido como POSA 2 y [2] . La categorización es distinta a la de GoF, y la plantilla también difiere, pero los principios organizativos se mantienen, como cuando se aplica un patrón a un contexto específico. En otros casos, como [12] y [14] se puede observar claramente la primera característica pero no las otras dos. Otros catálogos sin embargo no coinciden del todo con ninguna de las características citadas (por ejemplo, [7] y [17]). Es posible construir toda una explicación teórica para la coincidencia entre GoF, POSA 1, POSA 2 y [2]. A partir de esta explicación se puede derivar una estructura más amplia y adecuada que la que muestran estos catálogos para organizar el conocimiento de patrones.

Otro aspecto importante de destacar es que los autores de GoF señalan que el catálogo provee un vocabulario de diseño, con lo que establecen el vínculo con la noción de lenguaje de patrones:

"Los científicos de la computación nombramos y catalogamos algoritmos y estructuras de datos pero no nombramos con frecuencia otros tipos de patrones. Los patrones de diseño proveen un vocabulario para que los diseñadores lo usen al comunicarse, al documentar y al explorar alternativas de diseño. Los patrones de diseño hacen que un sistema se vea menos complejo al permitirle a usted hablar a un nivel de abstracción más alto que el que provee la notación de diseño o el lenguaje de programación. Los patrones de diseño elevan el nivel al cual usted diseña y discute sobre diseño con sus colegas" (ver pág. 352 de GoF).

Es bueno aclarar que los autores de GoF no consideran que su libro sea un lenguaje de patrones y admiten ciertas limitantes en este sentido. La noción de lenguaje de patrones también se ha asociado con un esquema para organizar el cuerpo de conocimientos de patrones que evoluciona muy rápidamente. De hecho en POSA 2 se afirma que la mejor forma de organizar patrones es a través de lenguajes de patrones. La rápida evolución del conocimiento de patrones se puede percibir en la frecuente aparición de variantes de patrones y combinaciones de patrones que eventualmente pueden llegar a considerarse patrones en sí mismas, así como en

la identificación de nuevas conexiones con otros patrones del mismo dominio y con dominios de experiencia complementarios.

Por otro lado, en contraste con los catálogos que organizan pequeñas colecciones de patrones, existen otros trabajos relevantes que están orientados a organizar grandes cantidades de documentos de patrones y a incorporar fácilmente documentos de las más variadas procedencias y estilos. Estos trabajos pueden categorizarse bajo el concepto de "repositorios de patrones". Algunos ejemplos son [20], [9] y [21] . En el caso de [20] y [9] se trata de una colección de categorías de patrones, una estructura de categorización de un solo nivel. En el caso de [21] no se vislumbra ninguna categorización específica. Más bien, este sitio presenta varias categorías de criterios que han sido considerados útiles para buscar un patrón y un esquema de búsqueda basado en estos criterios que es más complejo que el provisto por [20] y [9]. Si bien hay coincidencia en ciertas categorías, son mas las diferencias. Esto quizás sugiere que, por el momento, no es viable llegar a un acuerdo sobre una categorización específica que satisfaga las expectativas de por lo menos una mayoría de autores y usuarios de patrones.

En este contexto, se pueden plantear dos preguntas relevantes: ¿es la estructura de los mejores catálogos y repositorios publicados hasta ahora adecuada para soportar la evolución natural del conocimiento de patrones? y ¿es posible identificar principios de categorización generales (no una categorización específica) que sean bien recibidos tanto por quienes elaboran catálogos y repositorios, como por quienes los usan?. La situación actual plantea la necesidad de la aplicación intensiva, en contextos industriales y de procesos de enseñanza y aprendizaje, de este inmenso y cambiante cuerpo de conocimientos, para lo que se requiere de una estructura organizativa tal que, no solamente se facilite su consulta para asistir la resolución de problemas, sino también su reorganización continua dada su rápida evolución natural. Esta evolución se caracteriza por el surgimiento continuo de nuevos patrones, variantes, combinaciones y conexiones entre patrones y lenguajes de patrones afines o complementarios. En este trabajo se responden afirmativamente ambas preguntas y se esquematiza la solución en forma de patrón.

El enfoque de este trabajo consiste en ampliar la estructura recurrente de GoF, POSA 1, POSA 2 y [2] para representar adecuadamente una colección creciente y cambiante de lenguajes de patrones, así como las conexiones entre estos. Para esto se usan principios de categorización natural derivados de los lenguajes naturales que nos permiten a los seres humanos organizar grandes cantidades de experiencias provenientes de los más diversos dominios. Desde este enfoque, la función general de un lexicón de patrones (a diferencia de un catálogo, un sistema o un repositorio) es la de soportar la evolución natural de lenguajes de patrones afines y facilitar su uso. En este artículo se especifica un patrón para lexicones de patrones que se basa no solamente en el patrón recurrente observado en los catálogos mencionados, sino también en la siguiente caracterización de lenguaje de patrones:

Un lenguaje de patrones puede caracterizarse como una especialización de un lenguaje natural (es decir un subsistema simbólico empotrado en un sistema más amplio que es un lenguaje natural) cuyas categorías naturales han sido construidas para describir y organizar planes para resolver problemas de diseño de software, en forma heurística y en un dominio de diseño específico. Los planes son los patrones y constan a su vez de cinco aspectos básicos,"El Contexto" (C), "El Problema" (P), "La Solución" (S) y "Las Asociaciones con otros patrones" (A)—de acuerdo con [1]. Finalmente "Las consecuencias de aplicar la solución" (CS) que han agregado los ingenieros de software. Toda tupla (C, P, S, A, CS) tiene un nombre significativo que constituye una palabra de la especialización y por ende refleja las escogencias gramaticales preferidas por la comunidad de usuarios y autores de patrones.

Como subconjunto de un lenguaje natural, un lenguaje de patrones no se puede reducir a un conjunto (catálogo o repositorio) de patrones interrelacionados, ni a un sistema de software que organice las especificaciones de los patrones y sus interrelaciones. El lenguaje como tal emerge en la intersubjetividad, en el uso cotidiano. Un lenguaje de patrones evoluciona como un lenguaje natural, solo que especializándose en un dominio muy limitado—de fronteras difusas—de experiencias de diseño, al que ha sido enfocado por la comunidad de sus gestores y usuarios cotidianos. Esto significa que continuamente van a emerger nuevas categorías de patrones y nuevos patrones producto de la combinación y variación del conjunto original, así como

de la construcción de otros nuevos por parte de sus gestores y usuarios. Por el mismo proceso, un lenguaje de patrones entrelaza asociaciones con otros especializados en dominios de experiencia afines. De la misma forma en que existen diccionarios para los lenguajes naturales, es posible construir uno para un lenguaje de patrones. Un lexicón de patrones es básicamente un diccionario "hipervinculado" que sistematiza el conocimiento sobre los términos de un lenguaje de patrones. Sobre esta base, se ha preferido usar el término lexicón de patrones en lugar de sistema de patrones.

No se discutirá en este documento la justificación teórica de esta caracterización; más bien, siguiendo al tradición heurística del movimiento de patrones, se usará una estructura inspirada en los aspectos "Contexto", "Problema", "Solución" y "Consecuencias" para describir un patrón para lexicones de patrones. En lugar de la sección de "Ejemplo" a lo largo de todo el trabajo se mencionan catálogos que pueden considerarse intanciaciones parciales del patrón propuesto.

3 Problema

Si la función general de un lexicón de patrones es dar soporte al proceso natural de evolución de lenguajes de patrones afines, se puede derivar que el problema central en el diseño de un lexicón de patrones consiste en idear una estructura, por un lado, suficientemente bien definida y natural como para facilitar la búsqueda de patrones y la resolución de problemas de diseño, y por otro lado, suficientemente flexible como para acomodarse a la evolución natural de los lenguajes afines, lo cual conlleva la continua aparición de patrones nuevos, combinaciones y variaciones típicas, y dominios de experiencia de diseño nuevos pero afines. Cabe enfatizar que el problema no consiste en definir una categorización de patrones específica, sino más bien un modelo de categorización de patrones que cada grupo de ingenieros de software debe adaptar a sus propias necesidades. Este patrón sí pretende en cambio presentar ciertas heurísticas generales para construir categorizaciones de patrones. Se parte del supuesto de que aunque se puede llegar a cierto grado de coincidencia en cuanto a la categorización de patrones, la evolución acelerada del conocimiento sobre patrones todavía hace muy difícil un acuerdo suficientemente sólido que sustente una categorización óptima.

4 Fuerzas

Para lograr la estabilidad y adaptabilidad requeridas se deben considerar varias tendencias o fuerzas que se analizan a continuación:

4.1 Cada dominio y equipo de ingenieros de software puede configurar su categorización idiosincrática

De cada dominio de experiencias de diseño emerge un sistema de categorización idiosincrático. Cada sistema de categorías va a tener distinta anchura y niveles de subcategorización. Por ejemplo, la cantidad de categorías de patrones en GoF, POSA 1, POSA 2 y [2] es distinta y resulta de la construcción particular que cada grupo de autores ha hecho del dominio de experiencias que ha trabajado, aun cuando hayan trabajado a veces el mismo dominio. Más aún, en la actualidad el conocimiento sobre patrones se sigue incrementando día a día muy rápidamente, en estas circunstancias es difícil poder establecer una categorización de patrones universal. Por tanto, es relevante facilitar a cada equipo de ingenierios de software la definición de su propia categorización.

4.2 Las fronteras entre dominios de experiencia de diseño son difusas

Los dominios de experiencia de diseño de software tienden a mezclarse en su evolución natural, debido a que las fronteras son difusas; por ejemplo POSA 1 propone un esquema matricial—es decir, categorías con subcategorías—de categorización. Las columnas de tal esquema pueden considerarse de hecho dominios de experiencia distinguibles. Existen patrones como "Singleton" cuya categorización es ambigua, de hecho ha sido incluido tanto en "Patrones de Diseño" como en "Patrones de Programación" (o "idioms"). El patrón

"Composite Entity" de [2] se puede considerar una variante de "Whole-Part" de POSA 1, pero a la vez es de hecho un "Patrón de Programación". Todo esto contribuye a que las fronteras entre dominios sean difusas. Ésta es una característica muy poderosa de los lenguajes naturales que debe aprovecharse en un lenguaje de patrones y por tanto debe ser representada en un lexicón de patrones. Desde este punto de vista, no se trata de eliminar este tipo de ambigüedades, sino más bien de sacar provecho de estas, tal como sucede en un lenguaje natural. Por ejemplo, la categorización de un patrón en distintos dominios puede facilitar su búsqueda para distintos usuarios que de previo suponen que se ubica en distintas categorías.

4.3 Las categorías y dominios son parte esencial de los vocabularios de diseño

Los nombres de las categorías de patrones y de los dominios de experiencia tienden a usarse en las referencias a los catálogos y en las búsquedas que hacen los ingenierios de software como parte de la resolución de problemas de diseño. De hecho estos nombres también son parte esencial de los vocabularios organizados por los catálogos publicados hasta ahora. La cantidad de categorías de patrones también se incrementa constantemente, aunque no al mismo ritmo que la de los patrones.

4.4 De cada patrón tienden a emerger variantes típicas

De cada patrón propuesto van a emerger muchas variantes con el tiempo. Algunas de éstas se van a repetir al igual que los patrones que las originaron. Tanto las variantes típicas como la virtualmente infinita cantidad de aplicaciones va a estar implicada de alguna forma en el uso del nombre del patrón, de manera análoga a como usamos cualquier palabra de un lenguaje natural. Por ejemplo, en el contexto de una conversación entre diseñadores de software que están considerando la posibilidad de aplicar el patrón "EJB de Entidad Compuesta" ("Composite Entity" de [2]) van a estar implicadas las variantes originadas por las distintas estrategias de programación que de hecho se documentan en [2]. En un momento dado, la necesidad de mantener un cierto nivel de abstracción evitará que los actores discutan sobre las variantes, pero en otro momento, cuando se haya llegado a un acuerdo sobre la conveniencia de aplicar el patrón por ejemplo, definitivamente las variantes se harán explícitas.

En este contexto son relevantes los casos en que las variaciones son típicas y por lo tanto pueden ser consideradas en sí mismas patrones. El surgimiento de variaciones típicas podría basarse por ejemplo en la adaptación de patrones a tecnologías de programación específicas. Este es el caso de "Composite Entity" de [2] que puede considerarse una variación típica de "Composite" de GoF adaptada a la tecnología J2EE . Como todo patrón, una variante que llegue a ser típica tendrá su propio nombre. Los nombres de las variantes típicas incluirán elementos de los nombres de los patrones base.

4.5 De cada conjunto de patrones tienden a emerger combinaciones típicas

Los patrones siempre son usados en combinación con otros. Algunas de estas combinaciones se van a repetir y podrían llegar a ser consideradas patrones en sí mismas. Se puede citar como ejemplo "Controlador de Fachada" que se basa en "Controller" de [15] y "Façade" de GoF. En términos generales, a un objeto "Controlador de Fachada" se le asigna la función de resolver los comandos solicitados por el usuario distribuyendo mensajes apropiados entre un subsistema de objetos subsidiarios. A la vez, el "Controlador de Fachada" provee una interfaz simplificada para el sistema de objetos subsidiarios, una interfaz de alto nivel que oculta la complejidad del conjunto de interfaces provistas por cada uno de los objetos subsidiarios. Combinaciones típicas como ésta, derivan su nombre de la combinación de los nombres de los patrones originales; típicamente el nombre de uno de los patrones funcionará como adjetivo del otro.

4.6 Cada dominio de experiencias configura sus propias plantillas

De cada dominio de experiencias de diseño emerge una plantilla propia para la especificación de patrones. De hecho se conocen a la fecha varias plantillas . No parece conveniente estandarizar un esquema, más bien una parte importante de la comunidad de autores y usuarios de patrones asociados a la construcción de software ha optado por especificar con base en cinco aspectos primarios, cuatro tomados de [1] y otro que se ha agregado en catálogos como GoF y POSA 1, y que ha sido aceptado por la comunidad en general:

· Contexto (C).
· Problema (P).
· Solución (S).
· Consecuencias de aplicar la Solución (CS).
· Asociaciones con otros patrones (A).

4.7 Las categorías de asociaciones relevantes están cambiando

"En esta red, las asociaciones entre patrones son parte del lenguaje tanto como los patrones mismos." (ver pág. 314 de [1]).

En la comunidad de usuarios y autores de patrones se está haciendo un gran esfuerzo por identificar asociaciones relevantes. Se considera que esto es esencial para lograr mayores grados de expresividad en los lenguajes de patrones. Hasta ahora, se ha puesto mucho énfasis en las asociaciones de la categoría de complementariedad, como cuando un patrón X se puede usar para aplicar otro patrón Y.

Se pueden considerar otras categorías de asociaciones relevantes en un sistema o catálogo de patrones. Por ejemplo, en [18] se consideran algunas como las siguientes:

· "Un patrón contiene o generaliza a otro de escala más pequeña.
· Dos patrones se complementan mutuamente.
· Dos patrones resuelven problemas distintos que se traslapan y coexisten en el mismo nivel.
· Dos patrones resuelven el mismo problema en formas igualmente válidas y —por ende—alternativas.
· Patrones distintos comparten una estructura de solución similar lo que implica una conexión a un nivel más alto." (ver páginas 153 y 154 de [18]).

Lo cierto es que la red de asociaciones relevantes entre patrones es muy dinámica y entrelaza a patrones de un mismo dominio, de diferentes dominios y de diferentes niveles de abstracción, esto en virtud de que "... la cualidad innombrada aparece, no cuando un patrón vive aislado, sino cuando un sistema completo de patrones, interdependientes a varios niveles, deviene completamente estable y vivo" (ver página 135 de [1]).

Más aún, desde un enfoque teórico derivado de Lakoff [16], se puede argumentar que existen otras categorías de asociaciones relevantes, por ejemplo, asociaciones de complementariedad entre categorías de patrones. Considérese la asociación de complementariedad entre "patrones de estructuración" y "patrones de comportamiento" de GoF. Ésta deriva, entre otras razones, de la complementariedad entre "Compuesto" e "Iterador". En términos más generales, este enfoque lleva a considerar otras categorías de asociaciones tales como:

· Extensión, denominada literalmente en [16] “instanciación”. En el contexto de patrones se puede interpretar como que un patrón tiene asociado un vector de atributos Y que está contenido en el vector X de un segundo patrón, el énfasis se pone en que X agrega otros atributos a Y, o "X extiende a Y".
· Similitud, denominada literalmente en [16] “similarity”. Los vectores de atributos de X y Y correspondientes a dos patrones, tienen algunos atributos en común, pero también hay otros en que difieren. El énfasis se pone en el subconjunto de atributos en común.
· Transformacional, denominada literalmente en [16] “transformational”. Los vectores de atributos de X y Y de dos patrones también tienen en este caso un subconjunto en común. Tienen atributos en los que difieren, pero aquí el énfasis se pone en el tipo de relación específica que existe entre los atributos en que difieren. Este

tipo de asociación existe entre un patrón X y una de sus variantes Xv. Si X resulta de la combinación de Y y Z, también existe una asociación de tipo transformacional entre Y y X, y Z y X.

Finalmente cabe apuntar que identificar y caracterizar apropiadamente cualquier categoría de asociación que facilite la búsqueda de patrones, su aplicación en contextos específicos, la resolución de problemas de diseño, el aprendizaje o la enseñanza del diseño de software con base en patrones, va a mejorar la expresividad de los lenguajes de patrones. La sistematización de tales asociaciones es por lo tanto esencial.

4.8 La red de dominios de diseño es muy inestable

El concepto de dominio de diseño es una adaptación de lo que en [16] se denomina el principio del dominio de experiencia:

"Si existe un dominio básico de experiencia asociado con A, entonces es natural que todas las entidades de este dominio estén en la misma categoría que A" (página 93 de [16]).

Hasta la fecha, con base en los catálogos publicados, se pueden considerar cuatro dominios generales, es decir, independientes del dominio de aplicación:

4.8.1 Patrones de modelado de requerimientos ([14], [12] y [7]).
4.8.2 Patrones arquitectónicos (POSA 1 y POSA 2, aunque también contienen patrones de diseño).
4.8.3 Patrones de diseño (GoF) y [2].
4.8.4 Patrones de programación, estilos o "idioms" en inglés ([8] y [2]).

La principal asociación entre estos dominios es de complementariedad. De acuerdo con el dominio de la aplicación se debe escoger un patrón arquitectónico. Luego los patrones de diseño complementan la aplicación del patrón arquitectónico escogido y a su vez, los modismos o patrones de programación complementan la aplicación de los patrones de diseño escogidos.

A estos dominios genéricos se deben agregar los dominios de aplicación.Un dominio de aplicación constituye una categoría difusa y compleja de aplicaciones, relacionadas entre sí por el tipo de problemas y contextos que direccionan, así como por los requerimientos funcionales genéricos y las restricciones de desempeño o de plataforma tecnológica que normalmente les acompañan.

Por ejemplo un dominio de aplicación o dominio de experiencias podría ser lo que los ingenierios de software denominan "sistemas de información empresariales" o "sistemas de información para los negocios", y un buen ejemplo de aplicación o sistema de software propio de este dominio es un sistema de contabilidad. El siguiente diagrama presenta un panorama amplio de los distintos dominios de diseño de software que podrían ser relevantes (cada rectángulo representa un posible dominio).

En la figura #1 se intenta presentar un continuo de dominios de diseño vinculados a lo que podría denominarse en términos generales el dominio de diseño de software. En el extremo superior del eje vertical del mapa se sitúan todos aquellos conocimientos que son suficientemente generales como para ser útiles en cualquier dominio de aplicación. En el extremo inferior del mismo eje, en cambio, se sitúan todos aquellos esfuerzos tendientes a la producción de conocimientos orientados a un dominio de aplicación específico. Entonces, la franja superior del mapa representa dominios genéricos de patrones, independientes de cualquier dominio de aplicación. La franja inferior representa, por el contrario, dominios de diseño orientados a un dominio de aplicación específico; del lado izquierdo aparecen los conocimientos asociados a la programación y del lado opuesto los asociados al modelado. En el eje horizontal se han representado tres de las áreas típicas del proceso de desarrollo de software: las tres columnas representan al análisis de requerimientos, diseño y programación, actividades del proceso de desarrollo de software. La columna de diseño incluye tanto el diseño arquitectónico como el diseño detallado de objetos.

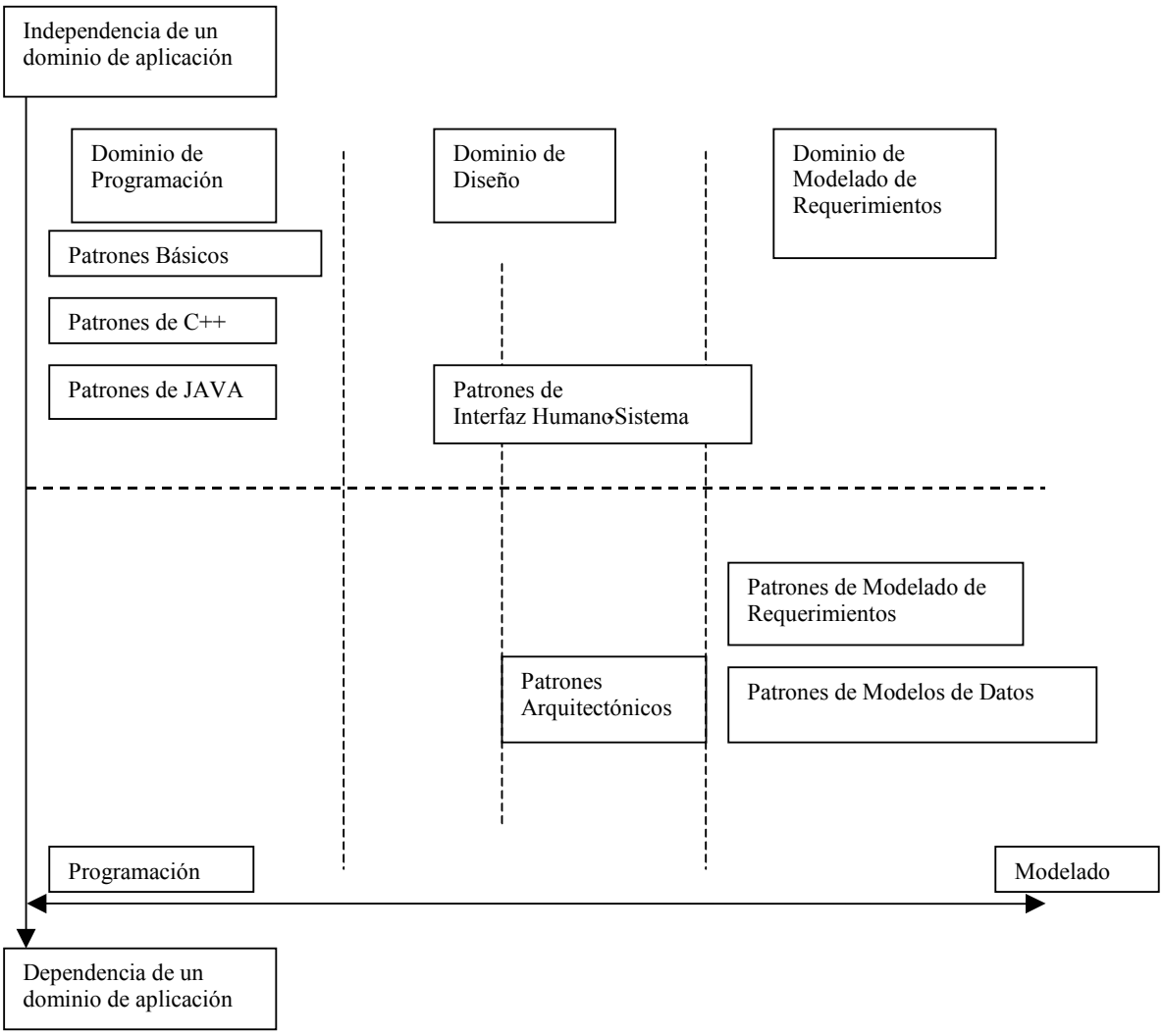


Figura 1: Mapa de dominios de patrones

Todas las fronteras son tenues y difusas, dependen, entre otras cosas, del grado de abstracción de cada patrón propuesto. Por ejemplo, [14] propone patrones de modelos de datos orientados a dominios de aplicación específicos, pero también patrones de modelos de datos presumiblemente independientes de cualquier dominio de aplicación. "Singleton" de GoF podría verse como un patrón de programación independiente del lenguaje. Los patrones arquitectónicos están fuertemente orientados a dominios de aplicación ("Layers" de POSA 1 a aplicaciones empresariales distribuidas, "Pipes and Filtres" a la construcción de intérpretes o compiladores). Los patrones de interfaz humano-sistema como "Composite View" de [2], tienden a ser independientes del dominio de aplicación, pero pueden ser muy útiles en la actividad de modelado de requerimientos para la elaboración de prototipos, y a la vez tienen implicaciones arquitectónicas que llegan a influir sobre el diseño de objetos.

Desde esta perspectiva se puede comprender que la red de asociaciones entre todos estos dominios es muy profusa y dinámica. Sin embargo su sistematización es esencial para facilitar la resolución de problemas de diseño basada en patrones. A la fecha no es posible anticipar la evolución que va a tener esta red de dominios. Un lexicón de patrones debería ser suficientemente flexible para adaptarse a la evolución natural de esta red de

dominios sin prescribir un conjunto específico. Todos los dominios citados tan solo son ejemplos de los que podrían incluirse en un lexicón específico.

5 Solución

La solución abarca cuatro aspectos:

· Modelo de funciones o casos de uso: es un modelo de casos de uso que se propone como esquema genérico de las funciones que se puede considerar al construir un lexicón de patrones y de las conexiones entre funciones.
· Modelo de categorización de patrones: es un modelo conceptual de objetos que se propone como representación de la estructura de categorización de un lenguaje de patrones.
· Modelo de categorización de asociaciones: es un modelo conceptual de objetos que representa las principales relaciones entre categorías de asociaciones que se pueden considerar al construir un lexicón de patrones.
· Clases de plantillas de especificación: es un modelo conceptual de objetos que representa las distintas clases de plantillas para especificar los distintos tipos de elementos que conforman un lexicón de patrones.

Esta solución se sustenta en la perspectiva de los lenguajes de patrones como subconjuntos de los lenguajes naturales y en la teoría semántica cognoscitivista de Lakoff [16]. La teoría de los Modelos Cognoscitivos Idealizados (MCIs) de Lakoff propone que cada categoría natural se puede modelar como un MCI, propone varias categorías de MCI y principios estructurantes, así como una organización de los MCI en tres niveles (supra-ordinario, ordinario y sub-ordinario) dado un dominio de experiencias. La teoría de Lakoff es relevante por cuanto los patrones pueden verse como MCI, lo que posibilita definir un modelo de categorización que permite construir lexicones de patrones estables y adaptables a la evolución natural de los lenguajes de patrones.

Para representar los distintos modelos que conforman la solución se ha optado por el UML ("Unified Modeling Language"), dado el uso tan extendido de este lenguaje gráfico entre ingenieros de software. Para una explicación completa de UML se refiere al lector a [3].

5.1 Modelo de funciones o casos de uso

Aunque este patrón para lexicones de patrones también se puede aplicar a la construcción de catálogos de patrones, la estructura de funciones no se puede usar en tal caso. Las dos principales funciones de un lexicón de patrones están representadas por los casos de uso "Editar el Lexicón" y "Navegar por el Lexicón".

a. “Editar lexicón”: Como caso de uso abstracto, representa la función general de actualizar las estructuras de categorización, de asociaciones y de especificación de los patrones y a la vez es un punto de acceso a otros casos de uso de un lexicón de patrones (en particular a "Navegar por el Lexicón", ver sección 5.1.2):

a.1 "Editar Dominios de Experiencias de Diseño (DED)": Permite editar la lista de términos usados para denominar los distintos dominios de experiencia de diseño (DED) abarcados por un lexicón de patrones. Esto significa que permite editar sus especificaciones y sus relaciones.
a.2 "Editar Categorías de Patrones (CP)": Permite editar la lista de términos usados para denominar las distintas categorías de patrones (CP) de un DED. Esto significa que permite editar sus especificaciones y sus relaciones.
a.3 "Editar Patrones (P)": Permite editar la lista de patrones (P) de una CP de un DED. Esto significa que permite editar sus especificaciones y sus relaciones. Esta funcionalidad se aplica igual sobre patrones base, así como sobre patrones de variantes de patrones y patrones de combinaciones de patrones.
a.4 "Editar Tipos de Asociaciones": Permite editar la lista de términos con que se conocen los tipos o categorías de asociaciones (TA). Esto significa que permite editar sus especificaciones.

a.5 "Editar Repositorio de Instancias": Permite editar el conjunto de instancias de patrones disponibles en un lexicón. Cada patrón puede tener asociadas varias instancias. Una de éstas puede ser declarada instancia prototípica (asociación "prototipo de"). Esta función se aplica igual sobre patrones base, como sobre patrones de variantes de patrones y patrones de combinaciones de patrones.

a.6 "Editar Repositorio de Plantillas": Permite editar el conjunto de plantillas disponibles en un lexicón. Cada elemento de un lexicón de patrones tiene asociada una especificación que se basa en una plantilla. Esto significa que permite editar las plantillas mismas y sus especificaciones.

a.7 "Actualizar Mapas": Algunas de las plantillas para especificar elementos de un lexicón de patrones convenientemente incluirán mapas de asociaciones entre los elementos subsidiarios. Por ejemplo, una categoría de patrones incluye un mapa que muestra los patrones de la categoría y sus asociaciones. Estos mapas se desactualizarán fácilmente con la realización de operaciones de edición sobre un lexicón. Por esta razón es útil automatizar la actualización de mapas a partir de los cambios recientes en las especificaciones de los elementos de un lexicón de patrones.

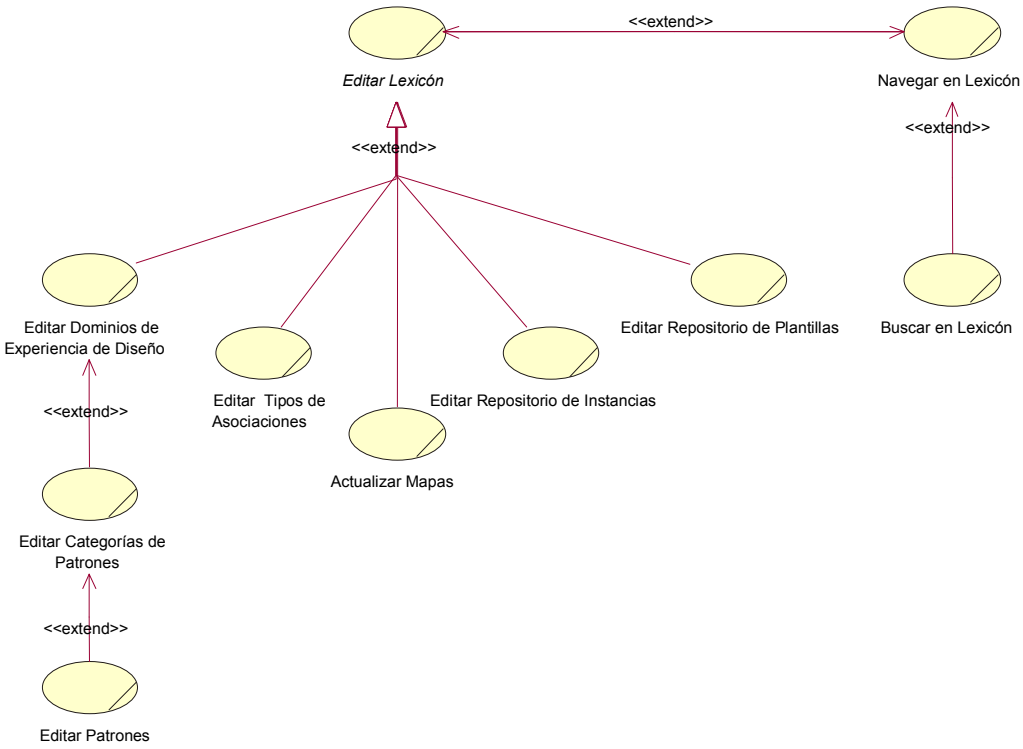


Figura 2: Modelo de Navegación de funciones

b. "Navegar por Lexicón"

Es un caso de uso concreto y representa la función general que permite recorrer los documentos de un lexicón de patrones, a través de los hipervínculos gráficos de los mapas y los hipervínculos textuales. A la vez es un punto de acceso a los demás casos de uso de un lexicón de patrones (en particular a "Editar el Lexicón", ver sección anterior):

b.1 "Buscar en Lexicón": Permite hacer búsquedas de documentos basadas en texto libre, o en descriptores. Estas búsquedas se pueden aplicar a un dominio, a una categoría o a un patrón y pueden reducirse a ciertas secciones de las especificaciones o pueden abarcar los documentos completos.

5.2 Modelo de categorización de patrones

La categorización de un lexicón de patrones se basará en la estructura de categorización típica de los lenguajes naturales, dado que un lenguaje de patrones es básicamente una especialización de un lenguaje natural. Por tanto, dado un dominio de experiencias de diseño, una de categorización de patrones se basará en tres niveles: básico u ordinario, supra-ordinario (más abstracto y general que el básico) y sub-ordinario (más concreto y específico que el básico) . El nivel básico u ordinario es el más usado, el más fácil de aprender, el más fácil de recordar, el que corresponde con la percepción gestáltica, el que corresponde con la mayor capacidad de formar imágenes (ver páginas de 31 a 56 de [16]).

La figura #3 muestra los elementos esenciales del modelo de categorización para un lexicón de patrones, el nivel de categorización al que pertenecen y sus asociaciones. Para cada elemento se introduce un estereotipo, con el fin de construir representaciones en UML del diseño conceptual de un lexicón de patrones:

a. Dominio de Experiencias de Diseño (<<DED>>): Corresponde con el principio de dominio de experiencia descrito en [16]. Ejemplos de dominios de experiencia que han sido elaborados en catálogos publicados son: "diseño detallado de objetos con J2EE " en [2], “diseño detallado de objetos independiente de la tecnología de desarrollo” en GoF y parcialmente en POSA 1, "diseño arquitectónico" en [2], POSA 1 y POSA 2, "estilos de programación" (o "idioms" en inglés) en [2] y [8]. Existen principalmente asociaciones de complementariedad entre los DED, por ejemplo entre "diseño arquitectónico" y "diseño detallado de objetos".

b. Categoría de Patrones (<<CP>>): Estos elementos pertenecen al nivel supra-ordinario descrito en [16]. Todo DED se dividirá naturalmente en varias categorías de patrones (CPs). Por ejemplo, en GoF existen tres categorías básicas de patrones: "patrones de construcción", "patrones de estructuración" y "patrones de comportamiento" las cuales a su vez se subdividen en dos sub-categorías: "patrones de objetos" y "patrones de clases". En cada categoría se identificará el patrón prototípico, es decir el patrón que mejor representa a la categoría. Esto proveerá a la categoría una estructura más natural que facilitará procesos de aprendizaje. Según [16] las categorías naturales se estructuran en muchos casos alrededor de elementos prototípicos. Estos elementos prototípicos no solo son usados como representantes de su categoría en procesos de razonamiento y resolución de problemas, sino que también facilitan el aprendizaje de la categoría misma.

c. Patrón (<<P>>): Estos elementos pertenecen al nivel de categorización básico u ordinario descrito en [16], pues son de hecho los que más se usan al resolver problemas de diseño. Un patrón se focaliza en una categoría de problemas y sus soluciones, por ende es una categoría. En GoF se presentan 23 patrones que muestran distintos tipos de asociaciones. Algunas de estas asociaciones se muestran en el mapa (ver solapa posterior de GoF), otras se analizan en el texto especificatorio de cada patrón.

d. Patrón de Variante de Patrón (<<P-VP>>): Estos elementos pertenecen al nivel sub-ordinario de categorización descrito en [16]. Son mucho más específicos que los del nivel básico u ordinario (los patrones base). Un buen ejemplo es el patrón "EJB de Entidad Compuesta" de [2] que resulta ser una adaptación de "Compuesto" de GoF a la tecnología JAVA .

e. Patrón de Combinación de Patrones (<<P-CP>>): Estos elementos también pertenecen al nivel sub-ordinario de categorización descrito en [16]. Al resultar de la combinación de patrones base, tienen asociados contextos de uso mucho más reducidos o específicos. Un buen ejemplo de patrón de combinación de patrones es "Controlador de Fachada" que se propone en [15] y "Fachada de Envoltura" ("Wrapper Façade") propuesto en POSA 2.

El principio del continuo entre gramática y lexicón descrito en [11], opera naturalmente al darle nombre a las <<P-VP>> y <<P-CP>>. Por esta razón, se deberá poner especial atención en la función gramatical que se asigna a los nombres que componen el nombre de una variante o combinación. Por ejemplo, los nombres citados: "EJB de Entidad Compuesta" y "Controlador de Fachada", no podrían invertirse sin correr el riesgo de causar confusión en el lector de un catálogo o usuario de un lexicón de patrones, respecto de la intencionalidad de estos patrones de nivel sub-ordinario.

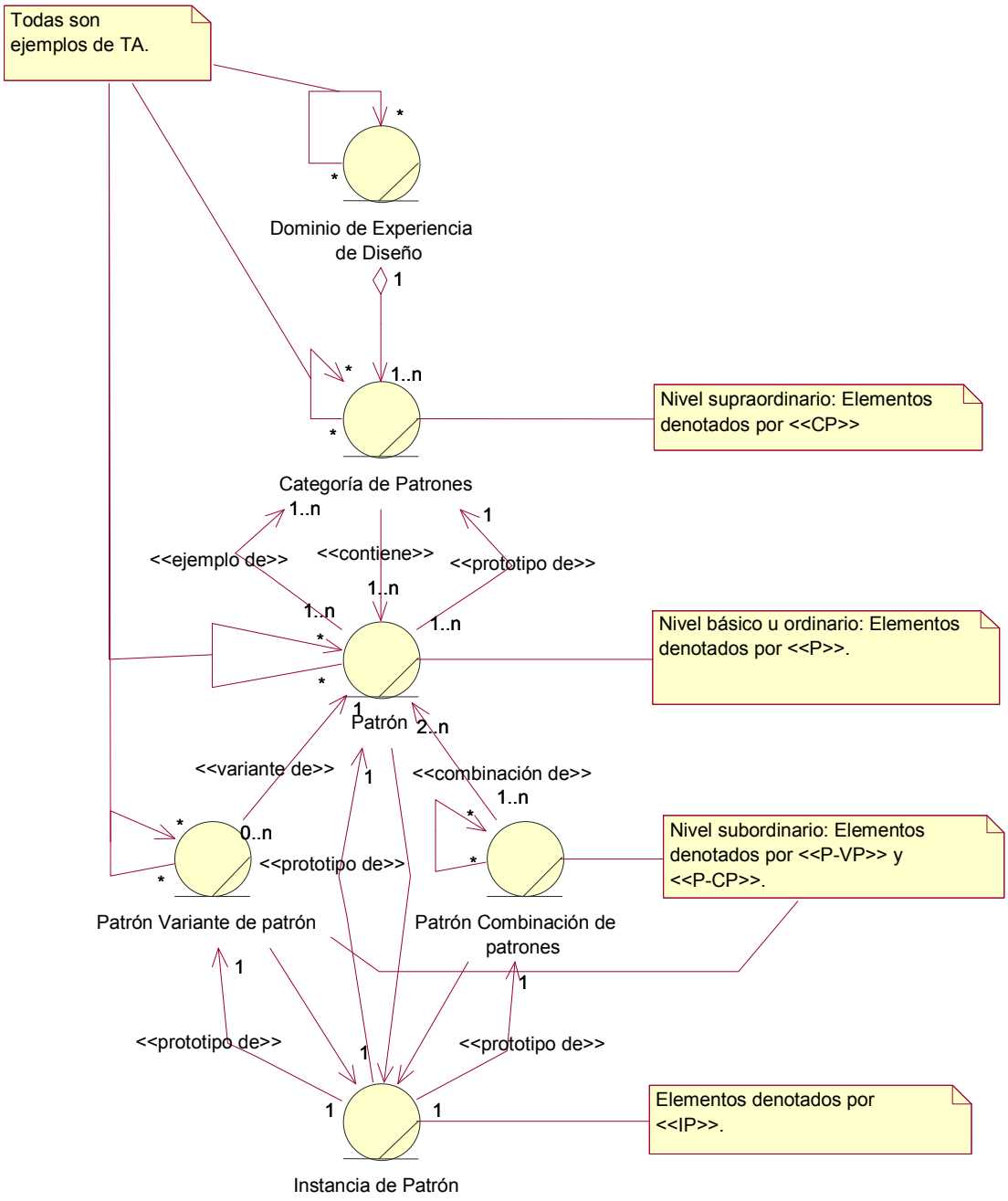


Figura 3: Modelo de Categorización de Patrones

Los principios más importantes que subyacen este modelo de categorización con tres niveles de abstracción dado un dominio son, por un lado, estabilidad y flexibilidad ante cambios en la cantidad de patrones. Por otro lado adaptabilidad a las necesidades y preferencias de cada grupo de ingenieros de software específico. En última instancia se busca que los usuarios de un lexicón puedan organizar los patrones que consideren

relevantes en la forma en que mejor les convenga, estableciendo sus propios dominios, categorías de patrones, patrones, variantes y combinaciones. Sin embargo, el presente patrón sí ofrece una serie de principios que buscan optimizar cada categorización idiosincrática. A través del modelo de categorización se establece que, dado un dominio de experiencias, debe incluirse tres niveles de abstracción (no cinco ni dos) y también se establece la funcionalidad de cada nivel. El sustento para estos principios deriva teóricamente de la organización de los lenguajes naturales según la teoría de los modelos cognoscitivos idealizados propuesta en [16].

Otro principio que cabe destacar es que no se busca definir fronteras precisas entre las categorías de los distintos niveles. Los estudios empíricos que reporta Lakoff sustentan el hecho de que los MCI no introducen fronteras precisas entre las categorías naturales. Esta característica de los lenguajes naturales no solo ha sido corroborada empíricamente sino que, como principio de diseño de lexicones de patrones, además es útil entre otras razones porque:

a. Facilita las búsquedas de patrones, pues se generan más caminos para llegar a cada patrón, categoría de patrones o dominio de experiencias de diseño.
b. Implica que se deben hacer explícitas relaciones de similitud y contraste entre patrones, lo cual facilita las búsquedas.
c. Implica que se deben hacer explícitas relaciones de complementariedad entre patrones, lo cual facilita el diseño basado en patrones.

5.3 Modelo de categorización de asociaciones

Cada categoría de asociaciones puede ser ejemplo de otra y a la vez pueden existir varios ejemplos de la misma. Idealmente el modelo de categorización de asociaciones no tendrá más de tres niveles. Para cada categoría de asociación se elaborará una especificación con base en una plantilla (ver sección de plantillas para los elementos de un lexicón de patrones).

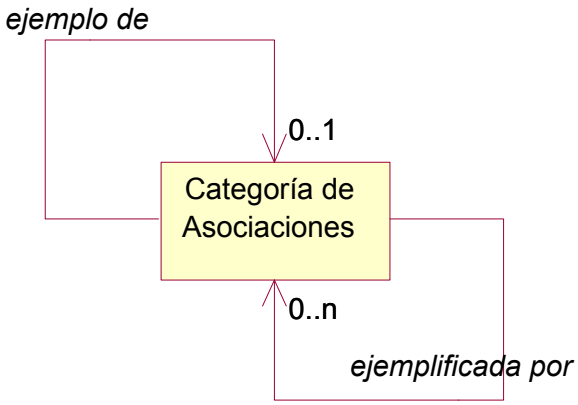


Figura 4: Modelo de Categorización de Asociaciones

La enumeración de las distintas categorías de asociaciones en un lexicón de patrones no es un objetivo de este trabajo (como tampoco la definición de una categorización específica de patrones); para efectos de esta propuesta lo importante es que debe considerarse a las asociaciones como elementos esenciales de un lenguaje de patrones y por ende deben poder categorizarse y sistematizarse, al igual que los demás elementos analizados anteriormente. Esto implica que:

a. debe darse nombre a los tipos de asociaciones entre elementos de un lexicón de patrones, y

b. que los tipos o categorías naturales de asociaciones entre elementos de un lexicón de patrones deben ser visualizados como patrones de asociaciones. Por tanto debe existir una plantilla para especificar categorías de asociaciones.

5.4 Clases de Plantillas de especificación

En un lexicón de patrones, todas las instancias de los tipos de elementos analizados hasta ahora (los tipos de elementos estructurales de un lenguaje de patrones, a saber: <<DED>>, <<CP>>, <<P>> y <<TA>>, así como <<P-VP>> y <<P-CP>>) deben ser especificados con base en una plantilla. Una plantilla estructura los atributos relevantes de una instancia de un tipo de elemento.

Aunque tienen muchas características en común, los catálogos que han servido de referencia a este trabajo (GoF, POSA 1, POSA 2 y [2]) no muestran un acuerdo en torno a la plantilla para especificar patrones (con excepción de POSA 1 y POSA 2). Por otro lado, a manera de ejemplo, los patrones del dominio de modelado de requerimientos, requieren una sección de "estructura" o "solución" más compleja que la que usan los catálogos referidos . Inclusive la especificación de patrones atípica observada en [12], [7], [8], [14] y [17] dudosamente podría atribuirse a una decisión arbitraria de sus autores. Es factible que los mismos dominios de experiencia tratados por ellos hayan inducido otro tipo de estructuraciones. Estos autores se aproximan más a la conocida “Forma de Portland” que a la más rígida propuesta por los catálogos mencionados.

La figura #5 muestra las distintas clases de plantillas que se deben incluir en un lexicón de patrones. La “PE de P” aparece como una clase abstracta para representar el hecho de que existen diferentes plantillas para patrones. En [5] se amplía el diagrama incluyendo los esquemas propuestos por GoF, POSA 1 y POSA 2 como especializaciones de la clase genérica de plantillas para patrones. Las plantillas para variantes de patrones (“PE de VP”) y para combinaciones de patrones (“PE de CP”) aparecen como ejemplos de esta misma por cuanto básicamente se orientan a especificar patrones más especializados. Al aplicar el patrón de lexicones de patrones habrá que escoger cuál de las plantillas para especificar patrones se usará para especificar variantes y combinaciones de patrones.

Tal como se aprecia en la figura #5, se proponen atributos específicos para los demás elementos que tradicionalmente no son considerados en forma explícita en los catálogos de patrones (dominios de diseño, categorías de patrones y categorías de asociaciones). Con base en la práctica de cada grupo de ingenieros de software, estas plantillas podrán adaptarse a sus propias necesidades, el principio que subyace es que se debe hacer explícito cómo es que el grupo de trabajo está organizando sus patrones, cuáles son los dominios, las categorías de patrones, las variantes y combinaciones, así como las categorías de asociaciones. Este también es conocimiento esencial de un lenguaje de patrones que no puede seguirse dejando implícito porque de hecho evoluciona constantemente con base en la experiencia de los diseñadores.

Se ha incluido una plantilla para especificar aplicaciones de patrones (“PE de IP”). Todos los catálogos mencionados incluyen al menos un ejemplo de aplicación de cada patrón. Esta sección típicamente no muestra una estructuración muy rígida, por esta razón solo se han incluido tres atributos que se han considerado esenciales con base en lo observado de los catálogos estudiados.

Al igual que en las secciones anteriores de la solución, no se trata de proveer plantillas específicas, sino más bien clases de plantillas que deben considerarse, así como la estructura básica de cada plantilla que deberá adaptarse. Por tanto, esta propuesta debe acomodarse a la evolución natural de un lexicón de patrones específico.

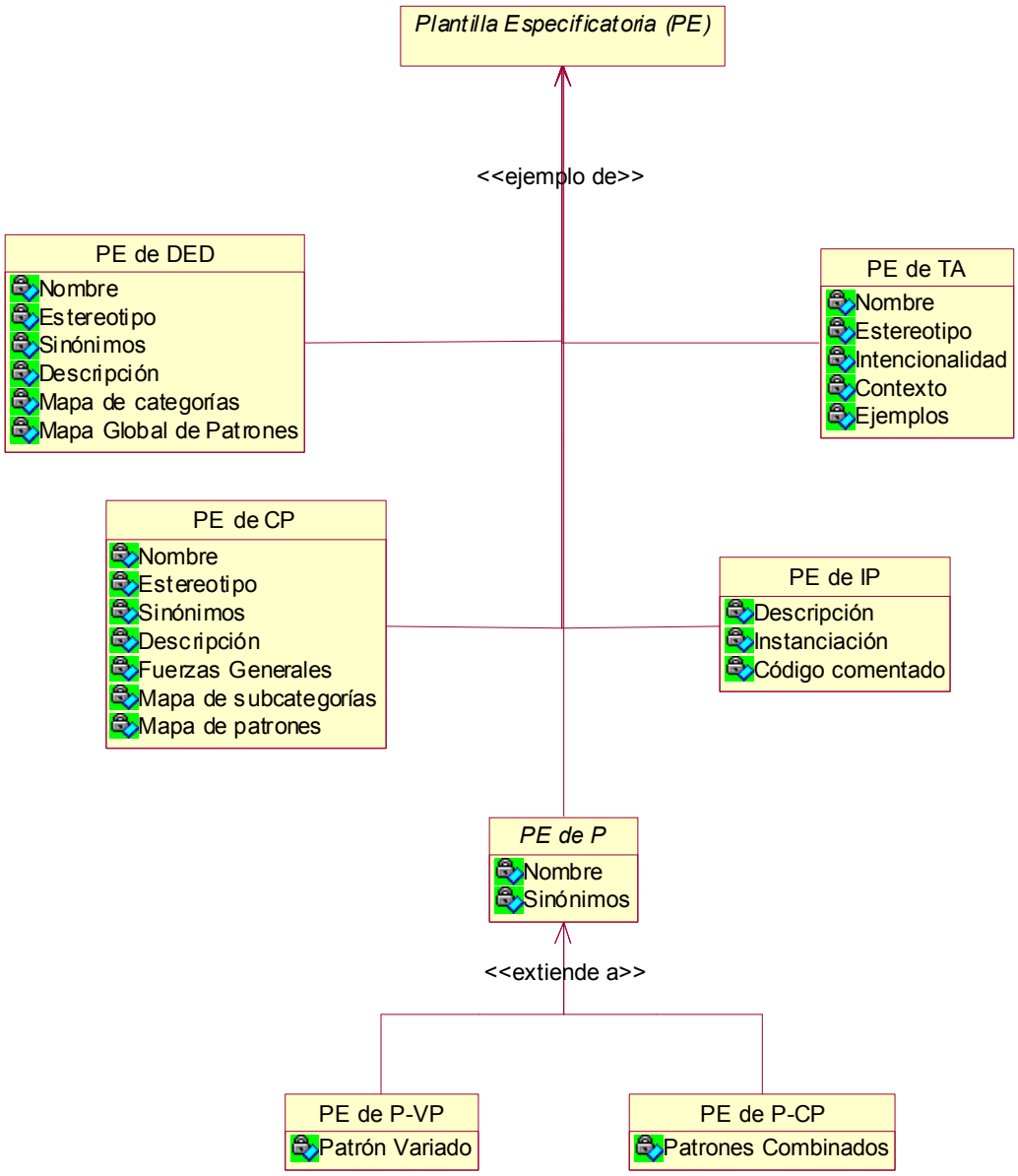


Figura 5: Clases de Plantillas de Especificación

6 Ejemplos

En la medida en que catálogos como GoF, POSA 1, POSA 2 y [2] muestran categorizaciones que coinciden parcialmente con el modelo de categorización propuesto en la sección 5.2, puede afirmarse que son ejemplos parciales de este patrón de lexicones de patrones. Todos estos catálogos, como se ha visto, incluyen al menos el nivel supra-ordinario y básico u ordinario en sus clasificaciones. En algunos casos se hace referencia a variantes. Particularmente se puede citar la referencia a las variantes de "Proxy" en POSA 1 y las denominadas "estrategias" ("strategies") en [2]. En estos casos se empieza a vislumbrar un nivel sub-ordinario.

Estos catálogos sin embargo no incluyen explícitamente una especificación del dominio o dominios que abarcan, ni de las categorías de asociaciones, ni de distintas clases de plantillas. Por esta razón sólo pueden considerarse ejemplos parciales de este patrón.

En [5] se describe en detalle cómo se podrían integrar los patrones de GoF y POSA 1 utilizando la solución propuesta en este patrón. En [5] aparece de hecho el ejemplo más completo de este patrón que por razones de espacio se elaboró en un documento aparte.

7 Consecuencias

Ventajas de usar el patrón propuesto para construir lexicones de patrones:

1. Se facilita la integración de lenguajes de patrones afines, pues la cantidad de dominios de diseño (<<DED>>) queda abierta, se pueden trazar asociaciones entre éstos y además es posible incorporar nuevas plantillas especificatorias de patrones que se adapten a los nuevos dominios.
2. Se facilita la reorganización continua de un lenguaje de patrones pues la red de categorías de patrones (<<CP>>) para cada <<DED>> también queda abierta y se pueden trazar asociaciones entre éstas.
3. Se facilita la integración del conocimiento a todos los niveles de categorización y dominios de diseño, pues la cantidad de tipos de asociaciones (<<TA>>) queda abierta para que se adapte fácilmente a la evolución natural de los lenguajes de patrones afines.
4. Se facilita la sistematización del conocimiento sobre variantes y combinaciones de patrones puesto que es posible integrar las que surjan, trazar asociaciones entre ellas, además de categorizarlas y representarlas en mapas.
5. Se facilita la sistematización del conocimiento sobre la aplicación de patrones, pues el repositorio de instancias o aplicaciones de patrones se puede incrementar y reorganizar continuamente.
6. Se facilita la búsqueda de patrones, pues se representan esquemas de categorización propios del lenguaje natural y se pueden introducir nuevos tipos de asociaciones. Además es posible introducir asociaciones útiles específicamente para la búsqueda de patrones como "similar a" o "contrasta con" (se ha elaborado con detalle en [5]).
7. Se facilita la resolución de problemas de diseño basada en patrones, pues es posible recorrer secuencias de patrones pertenecientes a distintos dominios complementarios. La resolución de problemas no solo está asociada con la identificación oportuna de patrones complementarios, sino también alternativos. Esto se puede lograr introduciendo categorías de asociaciones adecuadas (esto se muestra en [5]).
8. Se facilita el aprendizaje de diseño, lo cual puede ser relevante no solo para aprendices, sino también para los expertos que están incursionando en nuevos dominios de diseño. Esto por cuanto se propone una estructura de categorización cercana a la natural, se identifican prototipos de patrones en las categorías de patrones y se pueden introducir además categorías de asociaciones que faciliten la identificación de patrones similares y contrastantes.
9. Como representación de la estructura de categorización de un lenguaje de patrones, el esquema propuesto enriquece la representación de los lenguajes de patrones en comparación con las propuestas actuales que no toman en cuenta los distintos niveles de categorización propios del lenguaje natural, ni la estructura interna de las categorías, ni hacen explícito el uso de las distintas categorías de asociaciones.

La principal desventaja de usar el esquema propuesto es que probablemente implica mucho esfuerzo de actualización continua. Aunque un lexicón basado en este patrón tendrá una operación automática para facilitar la tarea de actualización de los mapas a partir de la inserción de nuevos patrones y asociaciones ("Actualizar Mapas"), el uso de un sistema basado en este patrón en un ambiente real (sea industrial o académico) indudablemente permitirá identificar variantes de este patrón que incorporen nuevas funciones orientadas a disminuir el esfuerzo de actualización. Otras desventajas son:

1. Los lexicones basados en este patrón no pretenden identificar una categorización de patrones estándar y aceptable para toda la comunidad de autores y usuarios de patrones, catálogos, repositorios y lenguajes de patrones. En el largo plazo, esto podría verse como desventaja, pues sí parece útil buscar, poco a poco, una unificación de criterios. Sin embargo, al hacerse explícitos los criterios de una organización específica de patrones (en términos de dominios, categorías de patrones y categorías de asociaciones) es posible que se simplifiquen las discusiones tendientes a la definición de una categorización universal, pues cada grupo de ingenieros de software podrá comparar su esquema de categorización con el de otros grupos más fácilmente.
2. Una categorización de patrones basada en tres niveles de abstracción para cada dominio de experiencias, como la que resultaría de aplicar este patrón, siempre va a ser más compleja que cualquier otra basada en menos niveles, sin embargo, el costo en términos de complejidad se compensa con el beneficio en términos de estabilidad, flexibilidad y adaptabilidad.
3. La construcción de herramientas para administrar el conocimiento de patrones es probablemente más difícil si se adopta este patrón, pero se reitera la ventaja en términos de mayor estabilidad, flexibilidad y adaptabilidad.

Se concluye que si lo que se quiere es representar una colección pequeña y relativamente estable de patrones de patrones no conviene usar este patrón, mientras que si se trata de representar una colección inestable de lenguajes de patrones entonces este patrón es idóneo.

Finalmente, cabe destacar que el sistema propuesto permite imaginar el desarrollo ulterior de una nueva generación de asistentes inteligentes de diseño de software basados en patrones. Una implantación adecuada de la solución propuesta, serviría como base de conocimiento para un sistema que, utilizando técnicas apropiadas de inteligencia artificial, podría asistir al usuario de un lexicón en la búsqueda de patrones útiles para la resolución de su problema de diseño.

8 Agradecimientos

Un agradecimiento muy especial a Jorge L. Ortega Arjona, Ph.D. quien como “shepherd” me ayudó a mejorar la versión original de este artículo. Agradezco también a Manuel Arce Arenales, Ph.D. por toda su ayuda a lo largo del proceso de investigación que ha dado como resultado éste y otros trabajos.

9 REFERENCIAS

[1] Alexander, Christopher, 1979. *The Timeless Way of Building*. Oxford University Press, New York.

[2] Alur, D., Crupi, J., Malks, D., 2001. *Core J2EE Patterns (Best Practices and Design Strategies)*. Sun Microsystems Press, E.E.U.U.

[3] Booch, Grady; Rumbaugh, J.; Jacobson, Ian, 1999. *El Lenguaje Unificado de Modelado*. Addison-Wesley, Madrid.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P., Stal, M., 1996. *Pattern-Oriented Software Architecture (A System of Patterns)*. John Wiley & Sons, West Sussex, Inglaterra. Conocido como POSA 1.

[5] Calderón, Alan, 2003. Unifying pattern catalogs (towards a pattern for pattern lexicons). No publicado todavía.

[6] Coad, Peter, 1992. Object-Oriented Patterns. *Communications of the ACM*, vol. 35(9), setiembre 1992.

[7] Coad, Peter, 1997. *Object Models (Strategies, Patterns and applications)*. Yourdon Press, New Jersey.

[8] Coplien, James, 1992. *Advanced C++: Programming styles and idioms*. Addison Wesley, E.E.U.U.

[9] Cunningham, Ward, 2003. *Category Pattern*. Disponible en: <http://c2.com/cgi-bin/wiki?CategoryPattern>. Accedido por última vez el 2 de setiembre del 2003.

[10] Czichy, Thoralf, 2001. *Pattern-based Software Development* (An Empirical Study). Recibido directamente del autor cuyo correo electrónico es: thoralf@czichy.org.

[11] Ellis, John M., 1993. *Language, Thought and Logic*. Northwestern University Press, Evanston, Illinois.

[12] Fowler, Martin, 1997. *Analysis Patterns: Reusable Object Models*. Addison-Wesley, E.E.U.U.

[13] Gamma, E., Helm R., Johnson, R. y Vlissides J., 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Company, E.E.U.U. Conocido como GoF.

[14] Hay, David C., 1996. *Data Model Patterns (Conventions of Thought)*. Dorset House Pub. New York.

[15] Larman, Craig, 1998. *Applying UML and Patterns (An Introduction to Object-Oriented Analysis and Design)*. Prentice Hall, New Jersey.

[16] Lakoff, George, 1987. *Women, Fire, and Dangerous Things (What categories reveal about the mind)*. The University of Chicago Press, Chicago.

[17] Pree, Wolfgang, 1995. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley Pub., E.E.U.U.

[18] Salingaros, Nikos A, 2000. The Structure of Patterns Languages. *Architectural Research Quarterly*, vol. 4, 2000. Cambridge University Press.

[19] Schmidt, D., Stal, M., Rohnert, H., Buschmann, F, 2000. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons. West Sussex, Inglaterra. Conocido como POSA 2.

[20] Tichy, Walter F., 2003. *Essential Software Design Patterns*. Disponible en: <http://www.ipd.uka.de/~tichy/patterns/overview.html>. Accedido por última vez el 2 de setiembre del 2003.

[21] Vinko Vrsalovic, 2003. *The Software Pattern Classification Website*. Disponible en: <http://patterns.ing.puc.cl/>. Accedido por última vez el 8 de setiembre del 2003.

A Pattern System to Supervisory Control of Automated Manufacturing System

Paulo Cézar Stadzisz[1], Jean Marcelo Simão[1,2] & Marcos Antonio Quináia[1,3]

[1] Federal Center of Technological Education of Paraná
Post-Graduation Program in Electric Engineering and Industrial Computer Science
Av. Sete de Setembro, 3165 - CEP 80.230-901 - Curitiba-PR – Brasil
http://www.cpgei.cefetpr.br
{simao, quinaia}@cpgei.cfetpr.br
stadzisz@lit.cpdtt.cefetpr.br

[2] Université Henri Poincaré (UHP)
Centre de Recherche en Automatique de Nancy (CRAN)
Présidence - 24-30, rue Lionnois BP 60120 - 54003 Nancy Cedex
http://www.cran.uhp-nancy.fr
simao@cran.uhp-nancy.fr

[3] State University of Center-West
Rua Presidente Zacarias, 875 - CEP 85015-430 - Guarapuava - PR
http://www.unicentro.br/
quinaia@unicentro.br

Abstract

Software patterns represent a promising research area in reason of the benefits happened of its application, mainly in terms of productivity reached with the reutilization. In automatics, patterns can be applied to recurring problems involving many types of computational systems. A complex domain of application, for which patterns can bring great contribution, is the Supervisory Control of Automated Manufacturing Systems (SC-AMS). This article proposes a system of patterns that aim to be applied in SC-AMS domain. The system is composed by an architectural pattern and three design patterns.

1. Introduction

Nowadays, a useful technique to compose computational systems is the *architectural pattern*. It expresses an organization or structural scheme, foreseeing a set of predefined subsystems, specifying its responsibilities and including rules and general principles to their organizations and relationships [6]. In fact, as general principle, the proposition of an architectural patterns is not a simple task, once a trade-off between efficiency in the performance of the instances and generality of the solution is needed.

To obtain a better organization and reusability degree in architectural patterns, a good practice is to define its subsystems in terms of design patterns, once these last ones are already well specified and possibly tested.

Architectural patterns based on the design patterns, can be applied in many application domains, as in telecommunications and automatics. In automatics, patterns are applicable, for

example, on the development of Supervisory Control of Automated Manufacturing Systems (SC-AMS). In fact, considering the typical complexity and dimension of SC-AMS, the development and use of architectural patterns can bring an important contribution to the developers.

Despite the numerous studies evolving SC-AMS [8][10][18][19], a lack of specific researches to the development of architectural patterns to these computational systems is noted [22]. This lack is especially related to aspects of the composition and execution of the control decision and consequent co-ordination of elements in the factory [24].

The conceiving process of an architectural pattern to SC-AMS is not a simple task because besides conceiving a strategy of factory control, it is necessary to generalize it in a set of situations of similar factory control.

Some approaches have been proposed in the literature as computational architectures or same as patterns to compose (in a certain way) SC-AMS [5][11][16][22][23], but none as architectural pattern composed by design patterns, regarding and solving the decision and co-ordination issue.

In this paper it is proposed an architectural pattern to this important area in computation and automatics called as Supervisory Control of Automated Manufacturing System (SC-AMS). The architectural pattern is based on design patterns, which are improvements of a computational architecture, which proposes strategies to effectively solve issues pertinent to SC-AMS, as the Monitoring & Command and the Regency (including the Decision and Co-ordination) [24][25].

The solution is agent based, where the agent classes specify a Generic Rules Based System (GRBS) [25]. Each instance of the architectural pattern is an Expert System (ES) with an advanced inference process, reached by the agent collaboration that results in incremental time growth in relation to the number of rules.

The proposed patterns are conceived from the analysis of supervisory controls of factories, including the simulated factory, modeled in the ANALYTICE II simulation tool [24]. ANALYTICE II allows expressing the fundamental characteristics of real industrial systems [14][23].

The architectural patterns is described following the POSA [6] format, whereas design patterns are presented as a mix of the two approaches very used called Alexandrian from [1] and GOF from [13].

The organization of this article is the following: section 2 is an overview about SC-AMS and its context; in section 3 there is an explanation over the design pattern Monitoring and in section 4 another explanation over the design pattern Command, while in section 5 presents the design pattern Regency and, finally, the section 6 presents the architectural pattern of SC-AMS in function of the presented design patterns.

2. An overview of SC-AMS

Before propose design patterns and an architectural pattern to SC-AMS composed by them, would be interesting a contextualization more detailed about Automated Manufacturing Systems (AMS), as well as about the Supervisory Control to AMS. In this sense, as example, this section presents simulated manufacture cell, its features and the related computational decisional system (specially the Supervisory Control).

The presented manufacture cell in Figure 1 is a system simulated in ANALYTICE II tool [14][23]. This manufacture cell is composed of various machines and their function is to produce fictitious parts of the types A and B.

Each processed part in this AMS has a *process plan* generated in another decision system called *Planning*. The plan specifies which machines the part must visit and which operation must

be carried on through it [5][15]. The *process plan* for A part is {<Store> <Table 1> <Machine-Tool> <Table 2>} and for B parts is {<Store> <Table 1> <Table 3> <Lathe> <Table 3>}. There could still be an alternative of manufacture in the process plan, in case of an existing *Dynamic Scheduler* (with a *dispatcher*) to carry out the elections in execution time [23].

The *Supervisory Control* software role is to make the constituent elements of AMS (e.g. lathes and robots) work in a harmonic way to carry out the manufacture of the parts following the *process plans* [18][19]. In a general manner, the elements of an AMS can be classified in equipment, hierarchical elements and process elements.

A common division of the equipment is to classify them as execution (carry out operations over parts), transport (carry out the transport of parts) and storage (carry out storing parts). In the proposed example both Lathe and Machine-Tool are classified as execution equipment, while Puma, Kuka 386 and ER III as transport and, finally, Store and Tables as of storage.

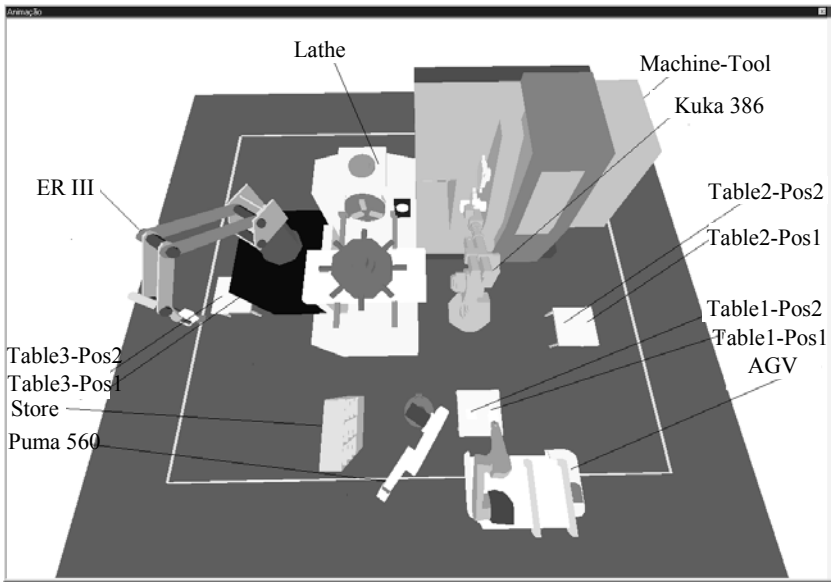


Figure 1 - Manufacture cell simulated in ANALYTICE II

The hierarchical elements are subsystems of an industrial plant, as the workstation (i.e. an equipment set), the manufacture cell (i.e. equipment set and workstations) and the plant (i.e. equipment set, workstations and cells). As example, the AMS illustrated in Figure 1, could have three workstations {<Lathe> <Table3> <ER III>}, {<Machine-Tool> <Table2> <Kuka 386>} and {<Store> <Table1> <Puma>}. The AMS as a whole could be considered as a composite cell by the three stations and the equipment of transport < AGV > (i.e. auto-guided vehicle).

This hierarchical division provides the SC-AMS development in several levels, known as "Hierarchical Supervisory Control" [15]. For example, a Hierarchical SC can determine that some parts go to a cell and not to another one. Once the parts are in the cell, another coordination level of this SC-AMS will determine which elements of that cell will process the parts.

The last type is the process element, which includes the parts (or products), the lot of parts and the pallets. One lot of parts consists of a parts group of the same type that advances in conjunction in the manufacture system. One lot has a processing priority and a production plan (to know which lot must visit which cell), allowing extending the scopes of supervisory control. Finally, one pallet is an element on which one or more parts (depending on the model) are placed for the purpose of protection and standardization in the transport. The pallets are limited

resources in the AMS. Depending on the morphology of the parts, the AMS may not use pallets, as occurred in the studied example.

3. Design Pattern: Monitor

3.1 Intent

The intent is to propose a design pattern, called Monitor, as a generic solution to facilitate the creation of monitoring module in the design of the Supervisory Control of AMS.

3.2 Context

In the scope of Supervisory Control of AMS there exist the *monitoring*, which consist in to observe the discrete states of factory elements. The context of this design pattern Monitor is proposed a generic solution (regarding the reusability) to monitoring problem in SC-AMS. The idea is generically represent and specify the monitoring of the factory elements, in terms of their attributes.

To solve the question of monitoring it is needed to monitor the discrete states of the factory elements (e.g. equipment, work-stations and manufacturing-cells) and notify these states to interested elements (e.g. specially the Regency). In fact, to know these states it is fundamental to allow carry out the Regency and consequently the Command [5][24] as is argue after.

More detailed examples of AMS elements are equipment (e.g. robot, lathe and auto-guided vehicle), hierarchical elements (e.g. station-works, manufacturing cells and plants) and process elements (e.g. parts, lot of parts and pallets).

Each AMS element has attributes that specify its characteristics. As an instance, the robot can have an attribute to specify its state of work (i.e. free or busy) or another to specify the state of operation (e.g. turned in, turned off or out of order). All these states must be monitored in the SC-AMS and the Regency keep track of it.

3.3 Problem

The AMS elements, in the line of time, can assume different discrete states (e.g. robot moving, robot stopped, lathe free and lathe processing) to each attribute. These states can have strong influence over the process of decision (inside of the Regency), and then it is fundamental to monitor them. The monitoring problem consists in observing the most diverse equipment discrete states (that can be viewed as facts) and informing them to other elements of the Supervisory Control, specifically to the decision elements, in a standardized way [2][3] [5][7].

In a more detailed way, the main forces founded in monitoring problem are:

- Interface with a lot of different kind of elements (e.g. production cells, equipments and products) to know its discrete states.
- Deduce some discrete state when the monitored element does not have a direct feedback.
- Standardize the discrete states in a way that other elements (e.g. Regency) can understand and work with them in an easy way and in a high level.

- For each element, separate the standardized discrete states (that are correlated) in little sets (that can be called "attributes"). As example, in the case of a robot, the attribute "general state" can assume the states "busy or free" and the attribute "gripper" can assume the state "open" or "closed".
- Quickly inform (notify) the interested elements (and only the interested ones) about the discrete states (or facts) of elements attributes, having as objective to allow the system to be more reactive.

In terms of pattern, the problem is to find a generic way (respecting a trade-of with the applicability) to carry out the monitoring in agree with these cited forces.

3.4 Solution

To expose the solution, it is proposed the use of computational (classes of) agents. These agents are weak-deliberative, cognitive, reactive and cooperative. The solution comes from agents responsible by monitor each viewed element and directly notify the interested ones, for example other agents from the Regency.

In the sense of determining the meaning of the agent in this work, a computational agent can be defined as a software module, with high degree of cohesion, with well-defined scope, with autonomy and taking part in a certain context whose changes are perceived by the agent. These perceptions may change the agent behavior and it may promote other changes in the context [12] [20][21][26].

The referred agents are cohesive objects instanced from a hierarchy of classes created to treat classes of factory's elements. In fact, in the pattern instance, the instantiated agents (from low levels of the hierarchy of classes) permit to better specify specific characteristics, whereas the higher level of classes of agent gives the generic behavior of them. Each agent captures the states of the monitored elements by interfacing with feedback elements (e.g. sensors, hardware and software) or by deduction of states using determined artifice (e.g. watchdogs or information correlation) [24][25].

3.5 Structure

The agents responsible for monitoring are divided into two main classes (of a hierarchy of classes) entitled as *FBA* (from *Fact Base Agent*) and *AT* (from *Attribute Agent*), which the instances are respectively called ***fba*** and ***at***.

Each type of feature observed regarding an element is kept by an ***at***, e.g. the state of work from a robot (free or transporting) or the general state of this same robot (active or out of order). While the whole element (e.g. a robot) is managed by a ***fba*** (which computationally represents the element) that aggregates the concerned ***at***s, monitors the information from the element, standardizes the information (e.g. in a predefined set of symbols) and sends the standardized information to the interested and aggregated ***at***s.

The name ***fba*** was chosen considering that each discrete state observed by the aggregated ***at*** is, also, fact. Then, the set of ***fba*** with its ***at***s is considered as base of facts, like those from Expert System (ES).

In the diagram of Figure 2, the FBA is specialized in equipment-oriented, hierarchy-oriented and process element-oriented agents. And each level can be specialized in more specific levels, as the case of the Equipment_FBA a possible derivation is the classes to treat equipment to store, process and transport the parts.

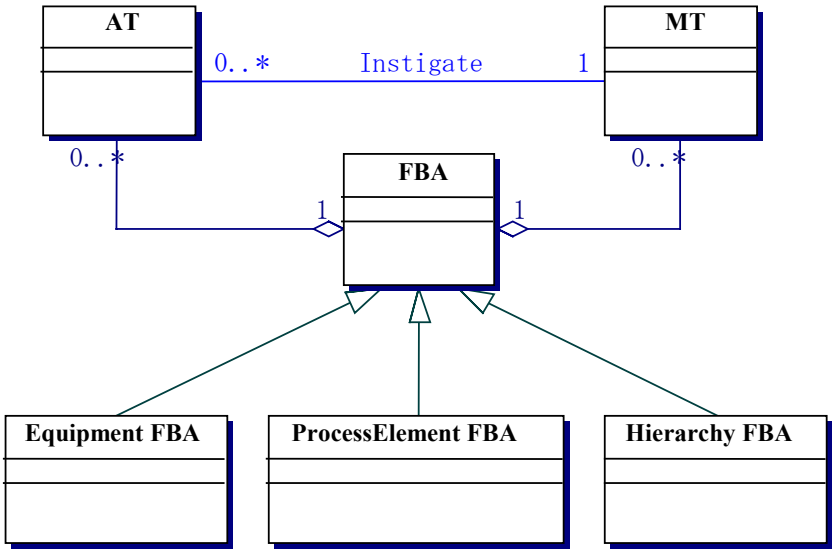


Figure 2 - Class Diagram to Monitoring.

3.6 Dynamics

A scenario for the execution (in a generic way) of the structure previously exposed could be the following:

- The ***fba*** monitors the characteristics of the elements of AMS and standardizes the information.
- The ***fba*** notifies the interested ***at***s responsible for maintaining, one by one the state of an attribute of the monitored elements.
- The ***at***s notify the pertinent Decision Elements and wait by a confirmation about the information treatment from these Decision Elements.

3.7 Consequences

The adoption of the Monitoring design pattern brings the following benefits:

- It "makes easier" to rewrite the monitoring component to work in a new SC-AMS. All the code that has to deal with specific characteristics of the environment elements is concentrated in the more specific levels of the hierarchy that has a standard interface (i.e. ***at***s). If it is required a specific change in the environment elements, it is expected that the changes in the code will be restricted to the code of the low levels. This allows a "quicker" adaptation (in terms of project) to the new environment with the largest reuse of existing code.
- In sense of the instanced solution:
  - Monitoring is encapsulated in well-contained elements, with functional independence in relation to the other SC-AMS elements.
  - Monitored information is mapped as a set of symbols common to all the other SC-AMS elements, transforming the heterogeneous ones in homogeneous ones.

  - The use of ***at***s brings a special advantage, the notification mechanism that permits notifying the changes happened to the Decision Elements, avoiding traditional searches looking for states or facts.
- As liability:
  - To a simpler AMS this solution can be so robust and maybe could be not compensatory use it. Therefore, the solution is indicated to complex AMS where there exists a great number of information to be processed (monitored).
  - To compose the Monitoring agent-classes in lower level demand expert people and high level of technology integration, being imperative (to real case) study the solution applicability with the actual technology.

4. Design Pattern: *Command*

4.1 Intent

The intent is propose design pattern, called Command, as a generic solution to facilitate the composition of command module in the design of the Supervisory Control of AMS.

4.2 Context

The context of the design pattern Command consists in to specify (in a general way) the send of commands to some kinds of elements (e.g. cells, workstations or equipments), using appropriate protocols and information (e.g. process parameters). It is also part of the context some synchronization of commands given by the co-ordination (from Regency) to the factory's elements.

4.3 Problem

The force in the Command question consists in to give commands to the factory's elements targeting some activities (e.g. a lathe machining a part). However, these commands must be exposed in high level of abstraction (to facilitate the instigation from co-ordination) and after, each command, must be transformed into a command of low level (comprehensible by the commanded element), respecting specific protocols and with the appropriate parameters, to be sent to the target element. All this process is called command-refinement.

Another problem (or force) pertinent to the Command is the synchronization of activities ordered by the co-ordination. The synchronization occurs when an element will receive an order, but it will not be executed because the element depends that another task be finished before (in one other element, which is its cooperator).

In the terms of design pattern, this problem must be exposed in a generic way, but also respecting a trade-off with the specific aspects needs to "easily" create instances of Command.

4.4 Solution

As solution to the command-refinement it is proposed generic class of agents, permitting derive more specific classes (therefore, a hierarchy of classes), which the consequent agents instantiated can work with specific and specialized knowledge. In fact, in the instances, there is

an agent to treat each command applicable over an element of the factory and these agents are aggregated in the same ***fba*** responsible by the monitoring process of this element.

The synchronization is carried out by the ***fba***, once that the solution is modeled (encapsulated) in classes of high level in the *FBA* hierarchy. In a generic way, this solution consists in knowing what the prerequisites to an activity are, and always that a prerequisite is not available, it must consider the possibility of synchronization. In this case, the ***fba*** asks to its collaborator if, in a determined low space of time, someone will make the prerequisite true. If the response is positive then the ***fba*** wait for its collaborator, or else this is the beginning of the solution to the fault detection by the correlation.

A didactic instance of synchronization (in a specific case) is a ***fba*** responsible for a machine that receives an order to process a part, but the part is not yet in its scope because a recent order given by a robot to transport the part to it is still in execution. Therefore, this order given to the machine will be possible to be executed in few instants of time, demonstrating the importance of the ***fba***s communication to know the future possibility of execution and making the consequent synchronization of their activities.

4.5 Structure

Each agent responsible by a command-refinement is called ***mt*** (acronym of method agent), instanced from the some class derived from the *MT* (i.e. *Method Agents*). As more specialized is the agent, more levels of derivations can have its class. In other words, the specialized knowledge to apply a command over an element is encapsulated in a ***mt*** (from a low level class in the *MT* hierarchy).

Also are the ***fba***s, being each one responsible not only by the monitoring and the synchronization problems, but also by aggregate each ***mt*** that treat some command to the element over its responsibility.

4.6 Dynamics

In a general way, the dynamic of the Command design part is:

- A ***mt*** is activated by someone (in fact, by an ***oa*** - order agent - defined inside of next pattern) as a high level order.
- The ***mt***, once activated, translate the high level order in low level order dependent of the context, i.e. dependent of the specific knowledge pertinent to the element that will receive the order.
- The ***fba***, that aggregate the ***mt*** in question, verify the prerequisite and solve any possible synchronization.
- The ***mt*** gives the low command to the ***fba*** that “transport” it to the equipment, respecting the specific protocol and some possible synchronization.

4.7 Consequences

- The specific knowledge about command of AMS’ elements is encapsulated in agent instantiated from a low level class (derived “from” the root class *Method Agent*), generating functional independence.

- As the specific knowledge and responsibility about each command of an element is embedded in a ***mt***, then the ***oa*** needs only activated the ***mt***, which out considers its specific details.
- The synchronization is made by the cooperation of ***mt***s, following a generic idea.
- The command and the monitoring are modeled inside of the same *FBA* class (or hierarchy of *FBA* classes), but independently because the subclasses *MT*s and *AT*s encapsulate the most responsibilities of each one.
- As liability, to develop the interfaces between the agents (from low level *MT*s) and the targeted elements (e.g. equipments) is still hard and dependently of specific knowledge and technology from the element. It is imperative (to real case) study the solution applicability with the actual technology state.

5. Design Pattern: *Regency*

5.1 Intent

The intent is to propose a design pattern, called Regency, as a generic solution to facilitate the composition of the "decision", "conflict-solution" and "co-ordination" integrated modules in the design of the Supervisory Control of AMS.

5.2 Context

The Regency (Decision, Conflict-Solution and Co-ordination) in SC-AMS.

The Regency responsibility is, regarding the facts monitored, to decide if some actions (pre-determined by the Planning) can be executed, resolve possible conflicts (using many information, included the arbitration from Scheduling) and co-ordinate the actions (pre-determined by the Planning), as well as given the orders that make part of the each action.

5.3 Problem

The problem can be divided in three sub-problems: decision, conflict and co-ordination.

Concerning to decision, the problem consist in relate or correlate observed facts by the monitoring (respecting the ways allowed by the Planning), make a logic calculus with the result of relations (and correlations) and decide what co-ordinations can be execute based in the resulting of the calculus and in the alternatives proposed by the Planning.

Related to Conflict, the problem is to identify conflicts (i.e. to know when there are two or more alternatives mutually exclusives) and solve it (i.e. to choose an alternative). More precisely, it is necessary identifies conditions where there are elements in competition by shared resources (e.g. a robot) and, based in some kind of parameter (e.g. from Scheduler), to decide what is the better option.

Finally, about Co-ordination, once having the conflict solved, it is needed to co-ordinate the orders (pre-defined) to instigate a certain number of (high level) commands [24].

In terms of design pattern, it is needed propose a generic solution to solve the Regency (Decision, Conflict and Co-ordination) problem, where to create instances be needed only give specific information to guide the generic solution.

5.4 Solution

The solution embedded in design pattern Regency is a sharing of responsibility, being the regency solved by a lot of computational (weakly-deliberative, cognitive, cooperative and reactive) agents, instantiated from a group of classes, that implement the knowledge of rules and also implement a conflict solver.

The solution generality is met in the structure of the group of agent classes, which allow instantiated agents only with the knowledge gave in rules, in a straightforwardly way, in the scope of the Supervisory Control targeted. In other others, the same structure can be used to any SC-AMS, being only needed give the parameters (e.g. knowledge of rules) to the generic structure. Still, is the knowledge of rules that will make to respect the restriction of Planning and Scheduling, once that this is specific to each system.

Each of these agents from rule is divided in others two to treat the condition and the action. The set of condition is the Decision sub-pattern[1] and the set of Action is the Coordination sub-pattern. Still there is the agent called Conflict-Solver justly to work over the conflict question.

In fact, the structure and interaction of agents compose the main solution of the regency. This solution is a new approach if compared with a lot of others solutions [23] [25].

5.5 Structure

The regency model is composed by the class *RA* (from *Rule Agent*) and *SA* (from *Solver Agent*). The *RA* instances are called ***ra***s and the *SA* instance is the ***sa***. The class *RA* has an aggregation relation to class *CA* (from *Condition Agent*, whose instances are the ***ca***s) and the *AA* (from *Action Agent*, whose instances are the ***aa***s). A ***ca*** is responsible by a fraction of the decision, as well as, an ***aa*** is responsible by a fraction of the coordination.

A ***ca*** is connected with ***pa***s (that are instances of *Premise Agents*), which collaborate with it to carry out its responsibilities. Each ***pa***s has the discrete value of an ***at*** (received by notification) called *Reference*, a logical operator (to make comparisons) called *Operator* and another value, called *Value,* that can be a constant. The ***pa*** makes a logic calculus comparing the *Reference* with the *Value*, using the *Operator*. The *Value* can be, still, other ***at*** value permitting, therefore, to correlate values of ***at***.

An ***aa*** is connected with ***oa***s (that are instances of *Order Agents*), which collaborate with it to carry out its responsibility. Each ***oa*** instigates changes in the factory elements by means of ***mt***s activations.

The way used to express the knowledge of the agents is a set of well-structured rules (oriented by agents attributes) as the exemplified in the Figure 3. In fact, the proposed approach is also a new way to compose the expert system, once each instance of the all architecture pattern is itself an expert system carried out by distributed agents [25].

The rule in Figure 3 is carry out by a ***ra*** (and its ***ca*** and ***aa***). The ***ca*** has the cooperation of tree ***pa***s and the ***aa*** has the cooperation of two ***oa***s. These agents make a robot transport a part from storage to a workstation when this workstation is free, the robot is free and the storage has a part.

[1] A sub-pattern is a well-identified and well auto-contained part of a pattern, but that cannot be separated because the cohesion with others parts of its pattern.

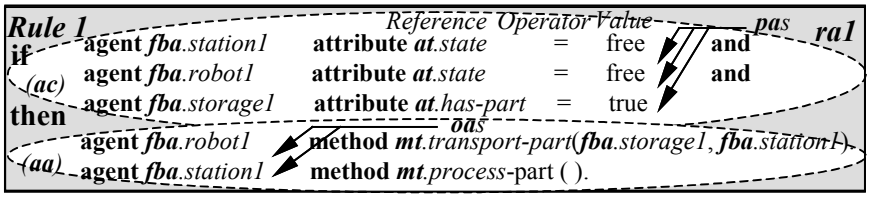


Figure 3 - Knowledge of agents in a rule format.

Expert agents that would create the ***ra***s can extract the knowledge from the rule. A way to implement this kind of agents is using linguistic comprehension or a friendly environment to rules composition.

Still there is the *SA*, which is created to generated an instance to work in the conflict moments. A conflict is established when two ***ar***s are in true state have an exclusive premise. An exclusive premise is one that has the *Reference* as being the "expression" of an exclusively shared resource, e.g. a robot that can serve two workstations, but in different times slice.

The ***sa*** is structured by a mechanism where each conflict established by *Premises* has a sub-agent responsible by taken the priority of the ***ra***s or other decision parameters (if the priorities are the same) and resolve the question. These alternative parameters can be, as instance, the values specified by a dynamic scheduler.

The Figure 4 is a UML class diagram of the proposed design pattern, where all the relation of the class agents (stated above) is expressed, included the class *SA*. These classes will allow to instance objects, which are a way to implement agents

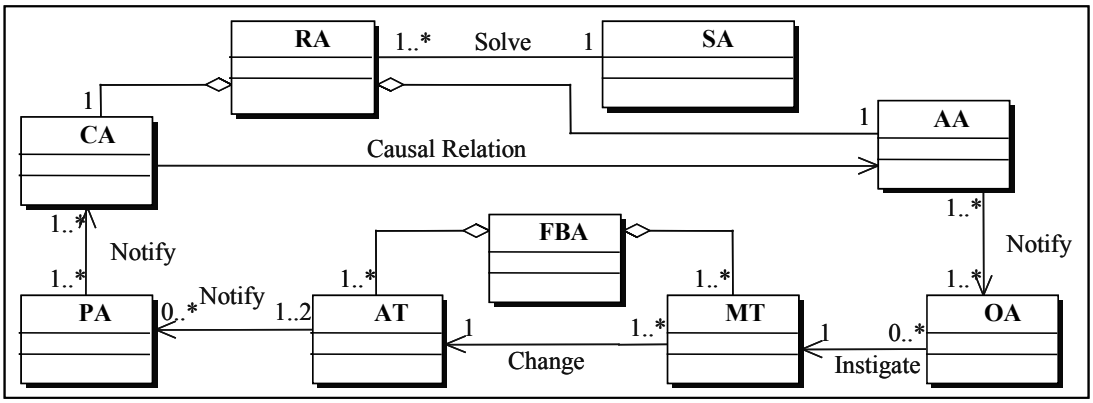


Figure 4 – Regency class diagram.

5.6 Dynamics

The ***pa***s receive notifications from the ***at***s (i.e. from its *References* and, when it is the case, from its *Values*) about the state change, once that the ***at***s know what ***pa***s have interest in its state. After the ***pa*** has received the notification with the new state, it uses this information to make a comparison (i.e. logical calculus), generating a boolean value to itself.

If the new boolean value is different from the last one, this is notified to the interested ***ca***s, that use this boolean value to make or re-make a logical calculus by conjunction with the boolean values of all connected ***pa***s. If the result of this calculus is true, then the ***ca*** put it respective ***ra*** in a true value.

After the ***at*** has notified all interested ***pa***s, it wait by a confirmation that the information was propagated by the ***pa***s. But the ***pa*** only confirms the propagation after the interested ***ca***s have confirmed their propagations. Evidently if (after a predetermined time) someone has not confirmed, it need to solve the problem (e.g. to notify again). This guarantees that all interested ***ra***s will be contemplated by the new facts.

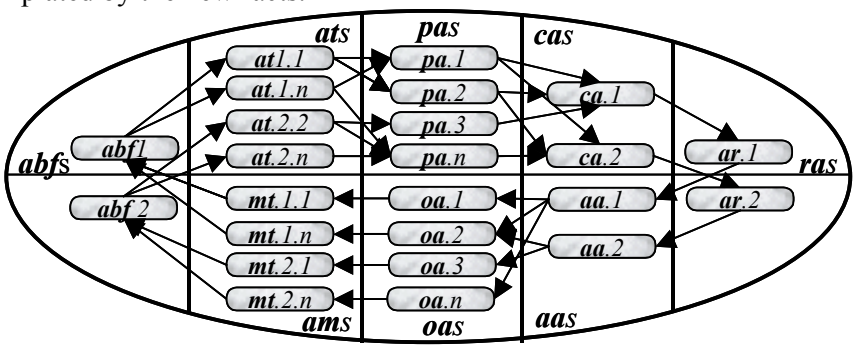


Figure 5 – The dynamic collaboration of agents (ellipses) using notifications (arrows).

When a ***ra*** has value true, it ***aa*** is passible of execution. To an ***aa*** be executed, its ***ra*** firstly verifies if all the ***at***s referenced in the collaborative ***pa***s are with the propagation confirmed. Then, after the resolution of a possible conflict, the ***ra*** activate its ***aa***. The ***aa*** is executed by the activation of the connected ***oa***s. Each ***oa*** instigates works in the ***mt***s.

The Figure 5 represents the notification process allowed by the agent structure of the architectural pattern, which this design pattern makes part.

Other relevant dynamic is the conflict identification. Once that a ***pa*** with exclusive attribute as *Reference* has been approved, and it collaborates to prove a rule it has a counter incremented. This counter represents the number of rules that it collaborates to be approved. Being this counter greater than one, the ***pa*** by itself notifies the ***sa*** to resolve the conflict. Once that ***sa*** is notified, some of its subagents take the priorities rules to decide impasse. But, if the priorities are the same, the ***sa*** demand a solution for whoever (e.g. the Dispatcher from Scheduler) or, in the absence of one interlocutor, it can choose randomly (being a default politics). The rule choose has the true from the exclusive premise confirmed and being approved, while the others have the true from the exclusive premise disapproved and, consequently, are disapproved too.

5.7 Consequences

The adoption of the Regency design pattern brings the following *benefits*:

- It makes easier to express the causal relation (to carry out the decision and coordination) by means of rules that work over objects attributes (or other methods that have mapping to this kind of rules, e.g. Object Petri nets) [4].
- It distributes the responsibilities in agents inside of the net.
- It resolves a complex problem with generic and simple classes of agents, where the complexity solution coming from the relations and cooperation among the instantiated agents that works following the relation established between the classes, mainly the relation called notification mechanisms.
- It better carries out the IE (Inference Engine), once that in an architectural instance the inference process works by notification relationship between agents, where the computational complexity is incremental in reaction to the

number of the premises, because only the interested agents are notified and it is possible share information (by the share of ***pa***s among ***ra***s).

- It allows quickly identify the conflicts and resolve them by many ways.
- It promotes a well conjunction, cooperation and function separation of the decision and coordination, as well as, a good cooperation between monitoring and decision and between coordination and command.
- The respect about the determination of the Planning and Scheduling are implicit in the rules composition, letting the SC model more independently of this relation.
- As liability, in fact, it is a little complex to understand all the cooperation among the agents. But, happily, it seen "easy" to apply the solution only understand as compose the rules (it considered that the Monitoring & Command can be composed by expert people).

6. Architectural Pattern: Supervisory Control

6.1 Intent

Define the SC-AMS in three Designing Pattern: Monitor, Command and Regency. Each one carries out macro-functions in the subject system and works in an interactive way with each one, forming the whole Supervisory Control. The idea is "divide to conquer", i.e. divide the SC-AMS allow better understand its functions and presents solutions more functionally independent.

6.2 Motivation

In this section it is proposed an architectural pattern to an important area in computation and automatics, known as Supervisory Control of Automated Manufacturing System (SC-AMS). Effectively, contributions to conceive the systems in supervisory control are necessary due to the development complexity of this kind of computational system.

6.3 Known Uses

The ideas of the proposed architectural pattern can possibly be used in Supervisory Control of Automated Manufacturing System (SC-AMS) and it has been used in SC of emulated AMS. Also, there are efforts to demonstrate the model generality, as well as the major applicability of the solution [25].

To be more specifically, the robustness of the constituted architectural pattern, as well as the efficacy of the instanced systems of this pattern, have been observed inside the supervisory control systems applied over the industrial plant simulations made in ANALYTICE II. These tests include the presented plant as an example in this work (in section II).

6.4 Structure

The architectural pattern is composed looking for the maximum high degree of functional independence between the parts (i.e. design patterns). To each design pattern, it was adopted a

policy “divide to achieve”, being the functions distributed in separated elements with simple action, maintaining the complex cooperation among them.

The diagram of Figure 6 shows the structure of the solution proposed. These elements present the follow (generic) dynamic:

- The Monitor knows the states of the factory’s elements (e.g. equipment) and notifies them to the Regency.
- The Regency, respecting the Planning and Scheduling, decides what to do (based in a set of options and solving possible conflicts among alternative solutions) and when to do the action to start the work, and make the coordination of orders to the factory’s elements.
- The Command, instigated by the orders from the Regency, effectively gives the command to the element (with all needed parameters) and can also make some needed synchronizations.

After that, the factory elements receive the commands, the Monitor makes new observation, instigating the Regency and, consequently, stimulating the Command to become a cycle or work regime.

6.5 Problem Forces

The plant’s elements need to receive discrete orders to carry out actions that allow the factory to work. However, these orders must be given in the appropriate moment, respecting the decision elements (planning and scheduling) and the viabilities of the elements (i.e. its discrete states), resulting in a harmonic interrelation among the commanded elements.

In terms of Architectural Pattern, all these functions exposed above must be modeled in a generic way, but allowing easy instantiation and generate robust, efficacy and efficient instances.

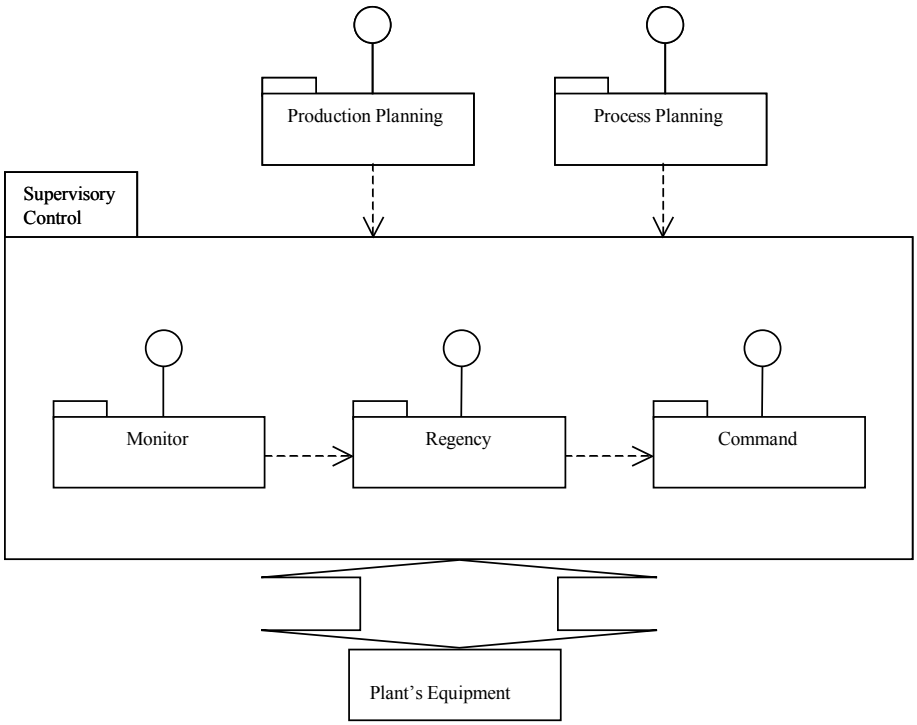


Figure 6 - Supervisory Control architectural pattern structure.

6.6 Benefits

The proposed architectural pattern is an improvement of the essay presented in the last SugarLoafPLoP 2002 [24]. The evolution is met in the specification of the architectural pattern in terms of design patterns, as well as the own advancement of the solution, like in the specification of the Conflict-Solver or in the aggregation of this conflict solver with the Decision and Co-ordination inside the unique element called Regency.

The solution presented includes concepts of artificial intelligence, once the model adopted is a kind of generic rule based system (GRBS), which instances allow carrying out CS-AMS. This model employs the agent concept in the instantiation of classes and uses an advanced and unique inference mechanism, by means of notification, reached by the agent collaborations (that permit the knowledge expansion) with an incremental time in the inference process.

In fact, the class agent concept utilization allows abstracting sub-systems that are cohesive, allowing creating well-defined frontiers and specifying the interrelation among them. As consequence, this agent-based solution still facilitates the archetype exposition in terms of design pattern. And then, the design pattern use make easier the reutilization, once the ideas are better explained inside a well-known standardization.

The utilization easiness is more evident observing the process to conceive instances. The instantiation of the Monitoring & Command takes place by the derivation of classes from the predefined generic hierarchical classes. Actually, in the case of the physics elements (e.g. equipment and its controls devices), this job would be easier if there were a well-defined way (e.g. protocol) to communicate with a computer, or else some artifices should be applied (e.g. deduction or sensors). While, the instantiation of the Regency is divided into the Decision & Co-ordination and Solver Conflict. To the first one it is enough to express the dynamic by rules (or other kind of compatible expression, like object Petri net) and transfer the knowledge to the predefined agent. To the last one it is possible to use the default solution specified, as well as, derive another one (like use of dispatcher agent).

As the architecture has well-defined interfaces, this facilitates the work of Planning and Scheduling, once they have to generate rules in the format predefined and standardized. Also, the incremental inference engine solution permits the use of a great number of alternatives without great effect over the SC-AMS performance.

6.7 Liabilities

Still, it was not developed a complete study about the availability (or weak features) of the solution to real cases in the industry.

6.8 See Also

As parallel work, it is being realized experiments to demonstrate that the architectural pattern can be viewed as a Petri net player, because it is known that exist a strong similarities between the syntax and applicability of rules of expert systems and Petri nets [4]. If the instances of the proposed Architectural Pattern can play any kind of ordinary Petri net, this is an interesting way to demonstrate the possible major range of applicability of the solution, once that Petri net are applicable to great number of discrete event controls.

Still in the theme of generality, one article was proposed in a congress called Logic Applied to the Technology – 2002 [25]. The article underlines a computational architecture as a

generic and advantaged alternative form to compose expert systems. The idea consists basically in the use of more generic levels of the Monitoring & Command, as well as the use of the Regency. However, the article was not presented as an architectural pattern and even the architecture was less developed.

Another aspect already developed (and being improved) is a solution to compose rules oriented to class, and not only to objects, following and improving this good practice already known in the literature. However, it is still necessary to write this solution (called Formation Rules) in terms of a design pattern, in agreement to the explained architectural pattern.

An objective, as future work, is to (study the possibility) and applies the proposed architectural pattern to real systems. Future work also includes: (i) defining a distribution computational model of the design pattern; (ii) refining the framework, enveloped by a friendly computational environment to constitute the Expert Systems (to SC-AMS), following the proposed architectural pattern; (iii) exposing the design patterns, from the proposed structural pattern, in terms of the existent standardizations in the literature, like the Gamma's Patterns [13]; and (iv) developing other architectural patterns for the conception and realization of other decision systems to the AMS, e.g. Planning, Scheduling and Fault Supervision, in an integrated way with the proposed architectural pattern to SC-AMS.

7. References

[1] ALEXANDER C., ISHIKAWA S. and SILVERSTEIN M., *A Pattern Language: Towns Buildings, Constructions*, Oxford University Press, New York, 1977.

[2] AARSTEN A., BRUGALI D. and MENGA G. *Designing Concurrent and Distributed Control Systems: an Approach Based on Design Patterns.* In Communications of the ACM - Special Issue on Design Patterns. 1996.

[3] AARSTEN A., ELIA G. and MENGA, G. *G++: A Pattern Language for the Object Oriented Design of Concurrent and Distributed Information Systems, with Applications to Computer Integrated Manufacturing* . In Pattern Languages of Program Design. Coplien, J. e Schmidt, D. (eds.). Addison-Wesley, 1995.

[4] BAKO V. and VALETTE R. *Towards a decentralization of rule-based systems controlled by Petri Nets: an application to FMS*. Fourth International Symposium on Kowledge Engineering, Barcelona - Spain. 1990.

[5] BONGAERTS L. *Integration of Scheduling and Control, In Holonic Manufacturing Systems*. (Ph.D. Thesis) KatholiekeUniversiteit Leuven, 1998.

[6] BUSCHMANN F., MEUNIER R., ROHNERT H., SOMMERLAD P. and STAL M. *Pattern-Oriented Software Architecture - A System of Patterns*. Wiley and Sons Ltd., 1996.

[7] BRUGALI D., MENGA G and AARSTEN A. *The Framework Life Span: A Case Study for Flexible Manufacturing Systems*. In Communications of the ACM. Out 1997.

[8] CHAAR J. K., TEICHROEW D. and VOLZ R. A. *Developing Manufacturing Control Software: A Survey and Critique*. The International Journal of Flexible Manufacturing Systems. Kluwer Academic Publishers. Manufactured in The Netherlands, pp. 53-88. 1993.

[9] COPLIEN J. and SCHMIDT D. (eds.) *Pattern Languages of Program Design*. Addison-Wesley, 1995.

[10] CURY J. E. R., De QUEIROZ M. H. e SANTOS E. A. P. *Síntese Modular do Controle Supervisório em Diagrama Escada para uma Célula de Manufatura*. V Simpósio Brasileiro de Automação Inteligente, Canela, RS, Brasil. 2001.

[11] FLETCHER M., BRENNAN R. W. and NORRIE D. H. *Modeling and reconfiguring intelligent holonic manufacturing systems with Internet-based mobile agents*. Jounal of Intelligent Manufacturing, 2003. Kluwer Academic Publishers in The Netherlands.

[12] FRANKLIN S. and GRAESSER A. *Is it an Agent, or Just a Program? A Taxonomy for Autonomous Agents*, Institute for Intelligent Systems – University of Memphis, In Proceedings of the Third International Workshop on Agent Theories, Architectures and Languages, Springer-Verlag. 1996.

[13] GAMMA E., HELM R., JOHNSON R. and VLISSIDES, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

[14] KOSCIANSKI A., ROSINHA L. F., STADZISZ P. C. and KÜNZLE L. A. *FMS Design and Analysis: Developing a Simulation Environment*. In Proceedings of the 15th International Conference on CAD/CAM, Robotics and Factories of the Future, Águas de Lindóia, v.2. p.RF25 - RF210, 1999.

[15] KÜNZLE L. A. *Controle de Sistemas Flexíveis de Manufatura - Especificação dos níveis equipamento e estação de trabalho* (Dissertação de mestrado) CEFET/PR, 1990.

[16] LANGER G., SORENSEN C., SCHNELL J. and ALTING L. *Design of a Holonic Shop Floor Control System for a Steel Plate Milling-Cell*, In 2000 Int. CIRP Design Seminar on Design with Manufacturing: Intelligent Design Concepts Methods and Algorithms, Israel, 2000.

[17] MCFARLANE D., SARMA S., CHIRN J. L., WONG, C.Y. and ASHTON, K. *The Intelligent Product in Manufacturing Control and Management*. IFAC 2002, 15th Triennial World Congress, Barcelona, Spain. 2000.

[18] MENDES R. S. *Modelagem e Controle de Sistemas a Eventos Discretos - Manufatura integrada por computador*, Belo Horizonte, Fundação CEFET-MG, 1995.

[19] MIYAGI P. E. *Controle Programável – Fundamentos do Controle de Sistemas a Eventos Discretos*, Edgard Blücher, 1996.

[20] PRADO J. A., ABE J. M. and ÁVILA B. C. *Inteligência artificial distribuída; Aspectos*. Série Lógica e Teoria da Ciência, Instituto de Estudos Avançados – Universidade de São Paulo. 1998.

[21] SCHMEIL M. A. H., *Sistemas Multiagente na Modelação da Estrutura e Relações de Contratação de Organizações*. Faculdade de Engenharia Universidade do Porto. (Tese de doutorado). 1999.

[22] SCHMID H. A. *Creating the Architecture of a Manufacturing Framework by Design Patterns*. Fachbereich Informatik, Fachhochschule konstanz. OOPSLA'95, 1995.

[23] SIMÃO J. M. *Proposta de uma arquitetura para sistemas flexíveis de manufatura baseada em regras e agentes*. (Dissertação de mestrado). CPGEI/CEFET-PR. Brasil, 2001.

[24] SIMÃO J. M., QUINAIA M. A. e STADZISZ P. C. *Um Padrão Arquitetural para Sistemas Computacionais de Controle Supervisório*. In Segunda Conferência Latino-Americana em Linguagens de Padrões para Programação (SugarLoafPLoP), Itaipava, RJ, Brasil, 2002.

[25] SIMÃO J. M. and STADZISZ P. C. *An Agent-Oriented Inference Engine applied for Supervisory Control of Automated Manufacturing Systems*. Advances in Logic, Artificial Intelligence and Robotics, Volume 85, IOPress Ohmsha - 3rd Congress of Logic Applied to Technology - LAPTEC 2002, São Paulo, Brazil. 2002.

[26] YUFENG L. and SHUZHEN Y., *Research on the Multi-Agent Model of Autonomous Distributed Control System*, In 31 International Conference Technology of Object-Oriented Language and Systems, IEEE Press, China. 1999.

Operations and Maintenance 2

Robert Hanmer
Lucent Technologies
2000 Lucent Lane
Naperville, IL 60566-7033 USA
hanmer@lucent.com
+1 630 979 4786

Abstract

Large computer systems consist of many parts, only some of which contribute directly to the application for which the system was built. This collection of patterns describes several capabilities of a system that perform this background, supporting role. The roles of these patterns is to aid in application specific data, performance and health measurements, managing faults, and also managing the entire system complex remotely. The patterns here, as well as others previously workshopped at other conferences, combine to form a pattern language describing a small telecommunications switch. Previous patterns have discussed the call processing application and other portions of the system's supporting capabilities.

There is a partitioning of functionality within any complex computer system of two main parts: one that performs the application for which the system was built, and the other one that is not directly involved with that application but that support the application functions. The patterns here address some of the important things that are not directly involved with the application functions. These patterns are part of a larger collection of patterns that describe the architecture of a telephone switching system.

Call Processing [6] introduced the main application component of a telephone switching system, the HALF CALL. This is the pattern that handles the details of the telephone or data connection to exchange information between two parties to the call. To conserve resources are shared and *switched* in real-time to provide the needed connection. This switching function makes a logical electronic path in a circuit switching system, or directs a packet between specific ports.

Defining the application architecture is important to guarantee that the system will meet the needs of its users. Equally important is the design of the supporting infrastructure. In many systems while the application is the glamorous part of the system, the supporting functions actually make up a majority of the system. The patterns in [7] describe several functions that are necessary to switch telephone calls, but which are not actually involved in seeing that a particular telephone connection or call is made. The patterns documented here continue and expand these supporting functions.

The following figure sketches out the relationship between the patterns presented here and those found in Call Processing [6] and OAM-1 [7].

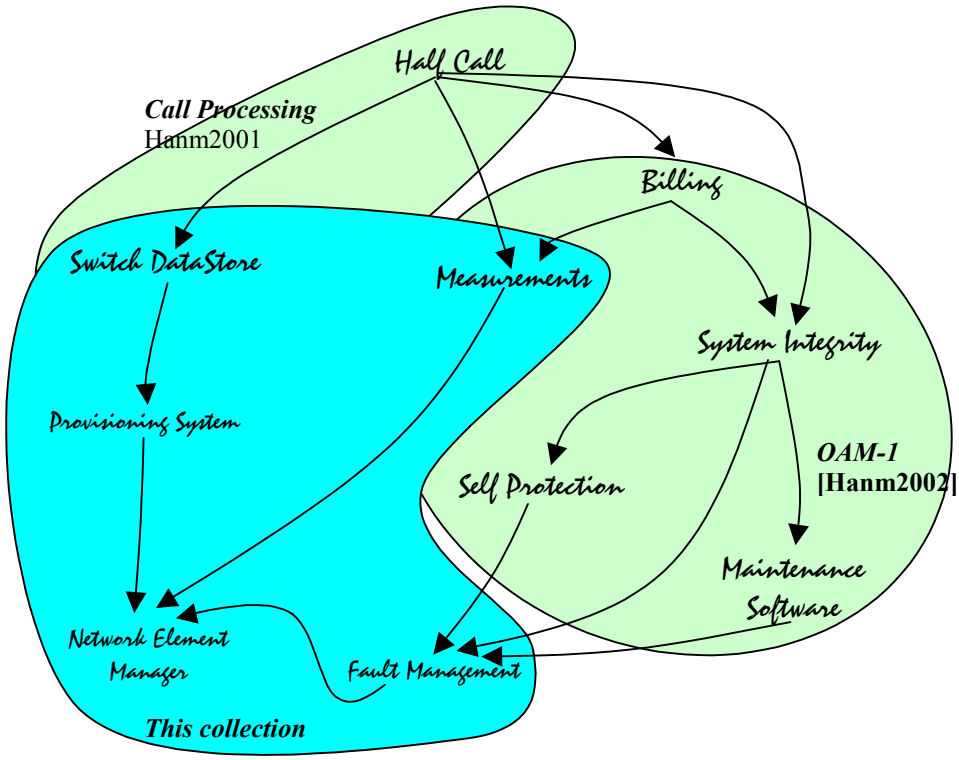


Figure 1

The patterns are presented in the Alexandrian form. The structure of each pattern is as follows: Each pattern begins with a numbered title. The number is used to reference to the other patterns in this collection. A photograph is then used to illustrate some essential aspect of the pattern. For example, the pattern MEASUREMENTS (3) shows a scientist taking a measurement at a field site. Following the photograph is a description of the context where the problem exists. Three diamonds follow this.

The next section of the pattern is the problem. It is printed in a bold font. A description of the problem, how it relates to the context and some possible solutions are discussed. The keyword *therefore* follows to introduce the solution section, which is also printed in a bold font and includes a sketch of the solution. Three diamonds provide a separation between the solution and the resulting context. The resulting context section introduces the situation after the solution has been applied. It frequently will point the reader to other patterns that can be used to resolve new forces that this solution has introduced.

1. Switch Data Store[1]

[2]

Some applications in telecommunications or data communications construct a permanent connection and mapping between endpoints. This is also true at lower levels, e.g. the physical layer, in the overall protocol hierarchies. If a call arrives on one endpoint, it is always routed to the same other endpoint as shown in figure 2.

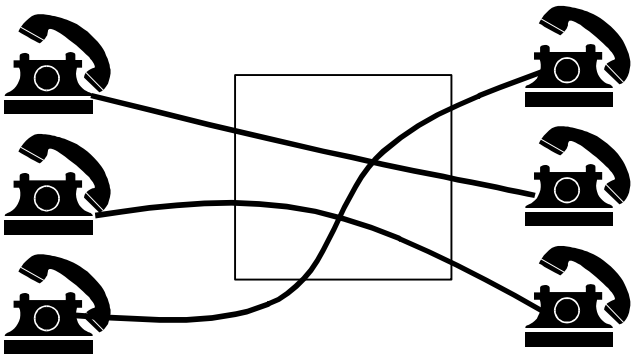


Figure 2

As networks grow and become more pervasive, direct connections between endpoints are no longer flexible enough. They also require dedicated resources because the connections are usually idle for a large percentage of the time. This also requires that a customer have multiple devices – one for each connection that they have. Figure 3 shows direct connections between different pairs of customer endpoints.

[1] An example implementation of this pattern is discussed briefly in [4].

[2] The photos that begin each of the patterns are provide a real-life example of an aspect of the pattern. The photos were selected from the USA National Archives and Records Administration, www.nara.gov

Figure 3

It is more efficient to reconfigure the connection for each call. Then when a call arrives on one endpoint, the destination endpoint it is routed to may vary from call to call. Variable connections, see figure 4, require some information about the connection is required. The endpoints might have different capabilities. Data about these differences can also be saved to prevent mismatching capabilities. This data describes the endpoints and also contains information about the types of calls that can be handled.

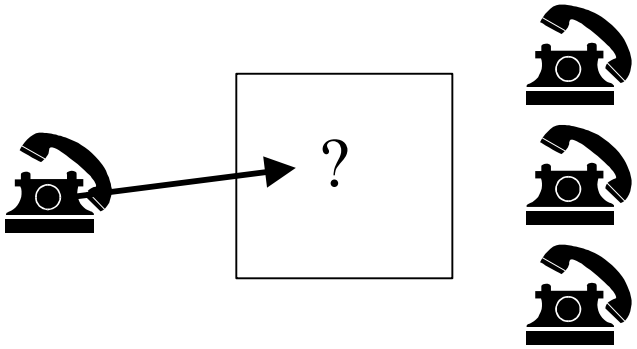


Figure 4

HALF CALL [6] discusses how we can arrange the call processing components that will participate in a connection between endpoints within our system. It describes the statics of the system's components, but it doesn't discuss the dynamics of an actual connection request. System architecture must address both the static and dynamic behaviours and attributes.

❖ ❖ ❖

How can the system dynamically handle connections that aren't predefined?

Telephone systems once solved this problem with an intelligent agent -- a human operator. Calls would arrive from the originating party and ask the human operator to speak with a specific party. The operator would either know from prior experience or look up which connection to make to connect the parties.

This prior knowledge might be in the form of a lookup such as *to connect Frank to Jane, connect ports 3 and 14 together*, as shown in Table 1. As the number of possible connections grew, the amount of knowledge that is required grows.

Frank	3
Joe	17
Jane	14
Billy	10

Table 1

Eventually the amount of information becomes more than the operator can easily handle. A better solution is needed.

An important thing to note is that the queries to the system are very consistent. The same information is sought each time and the request is almost of the same form.

How do I connect calling party a to called party b?

The calling party might not even be described by name; the name is not important. And sometimes it can be distracting. Frank may be calling Jane, but he might be using Ralph's telephone. What is important is the port through which the calling party accesses the system as shown in figure 5.

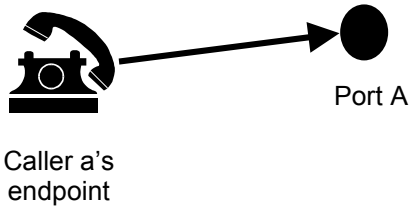


Figure 5

This changes the query to:

How do I connect the calling party on port A to called party b?

The modern telephone network identifies particular parties with a number. Usually a telephone number, such as 123 4567, or an IP address 123.12.34.56. The actual party at the end of the number may or may not be the desired party B, but the number points to a place that B has been and is expected to answer regularly.

So the query is now:

How do I connect the calling party on port A to some called party on port B?

A small database can store the data needed for such a regular query. Originally human operators performed these lookups. Early automatic systems implemented the translation and switching in hardware. If a call desired destination **b** it was switched to port **B** all in hardware. Eventually software replaced this hardware function.

The database must support the simple queries like those just described, and also be able to support a number of administrative actions.

Sometimes customers receive new, additional telephone numbers or network addresses. For example, customer **b** obtains an additional number, and is now on both ports **B1** and **B**? Some way of updating the database is required.

Transactions that are required include:

- the initial population of number to port mappings into the switch database (customer c is assigned to port C),
- changing entries (customer b is moved to port D), and

- deleting an entry altogether (customer d leaves the system).

Few other types of transactions will be required. The data needs are quite simple.

Since this database will be interrogated on every connection within the system it must return its answer quickly.

If the database is unavailable due to a failure, the system cannot connect calls, so the database system must have few faults.

A general-purpose database could be used to store and retrieve the information needed. However most database systems require more overhead than can be afforded, both in terms of retrieval time and memory overhead. A better solution is to use a small custom configured database system that is tailored to the situation. Therefore,

Install a small, custom data storage system that will be able to quickly and reliably decide how to connect two parties to the call. Figure 6 shows the connection of a database containing customer data with one of the Half Call entities associated with the call. Table 2 outlines the responsibilities of the customer data database.

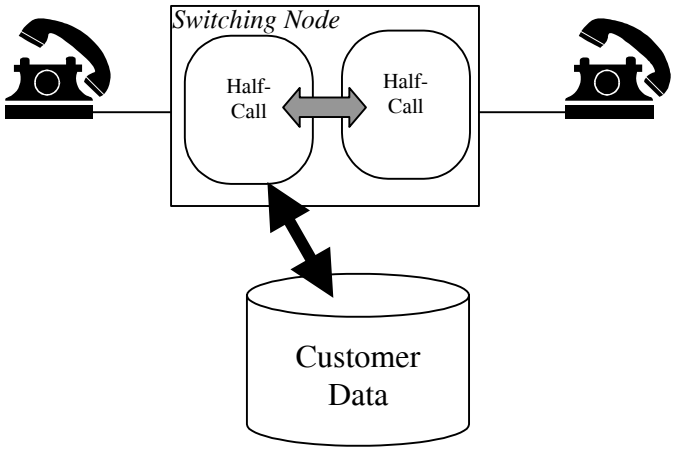


Figure 6

Switch Data Store
Responsibilities: • Reply to queries about port/customer location • Update port/customer location mapping • Delete port/customer location mapping • Add new port/customer location mapping

Table 2

❖ ❖ ❖

In many systems, such as at the lower level protocol routers and switches this database might be implemented in hardware. And some protocols might include all the addressing information within the contents of the message, eliminating the need for a custom database.

The database can get much more complicated as additional features such as address translations are required. An example of this is when a telephone call with one of the toll-free area codes such as 800, 877, or 888 is made. In these cases the systems processing the call request must translate the number that was entered (888 123 4567) into the number of a real

telephone. [10] This complication evolved into telephone systems. Initially toll-free service in the USA was provided to serve the needs of traveling salesmen to call the home office.

In all probability the system's database will not be populated through entirely manual actions, nor will it remain constant. The types of transactions are simple, yet the system can benefit through having a PROVISIONING SYSTEM (2) that will administer the changes.

2. Provisioning System

There is a lot of data in a telephone switch that must be maintained. This is the data that the SWITCH DATA STORE (1) contains. Getting it into the system is to "provision" it.

❖ ❖ ❖

Somehow we need to get the data into the SWITCH DATA STORE **(1).**

We could type it in each time we need it. Or each time we turn the system on. Since the system tries to run non-stop (SYSTEM INTEGRITY [7]), we wouldn't even need to do this very often. But we might get something wrong. We might forget something or get something wrong (e.g. a typographical error).

The switch is designed to process telephone calls, not to interface to a provisioning person. Its primary purpose is to switch telephone calls, not to look to its provisioning staff to be putting in data updates.

Keyboards or really any human controlled interface to the system is slow. With 1000 bytes of data to be entered over a 9600 bps serial link (Does this make sense today? What's the speed of the keyboard connection of a PC?) This takes { 1000 bytes / (8 bits/byte) * 1 sec/9600 bits} seconds to get the data into the system. There are higher speed links nowadays, such as high speed Ethernet. High-speed interfaces allow the data to be transferred much more quickly, which minimizes out of service time.

Humans can't communicate with the system at these high speeds but another computer can.

The data in the system might not be in a human readable form. So someone must translate it, or the system must translate it. This translation takes time from the primary purpose of the system.

Several different switches might need the same data. If a common source were to provide the data to each of them then there is less chance that the data will differ erroneously.

Therefore,

Build a computer system that will interface with the switch to put in the data it needs in its Switch Data Store. This system will have a model of the Switch Data Store

and will interface with the switching node to access and update the node's database, as shown in figure 7.

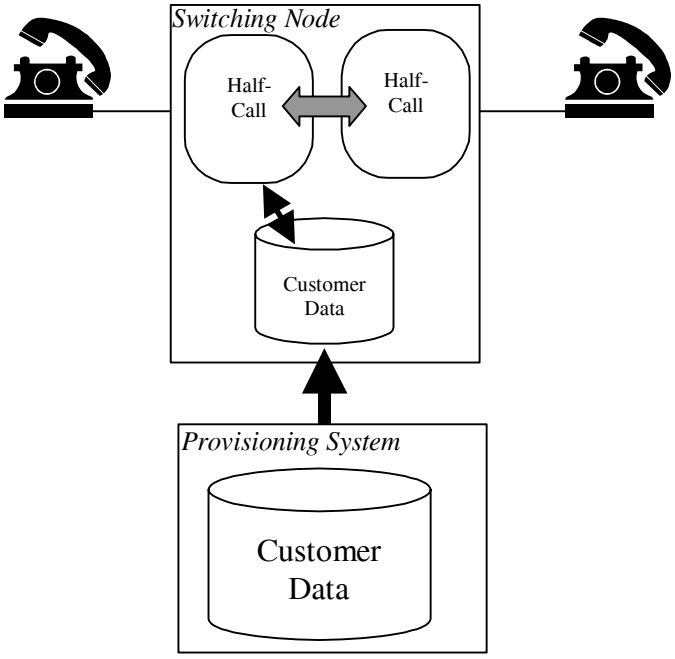


Figure 7

❖ ❖ ❖

Initial and ongoing provisioning will be done much faster than if the only interfaces on the system are the human interfaces or a serial link.

Interactions with this other system must be restricted somehow (PROTECT THYSELF [7]) to prevent it from using too much of the switch's resources. Remember, that Provisioning is necessary for the switch's primary function, but the act of provisioning is not the switch's primary function.

3. Measurements[3]

… The telecom system consists of many parts, some hardware and some software. You hope to make money by using this system to serve people who will pay you for services. You might need to add additional capabilities or additional systems to support additional customers in the future. In order to supply the appropriate amount of service to a region — not too much, wasting your capital, nor too little, and missing revenue opportunities — you need information about the activities. If a particular geographic region supported by a switching system grows very rapidly the system might not be able to handle the load. By knowing how much the system is being used, your engineering staff can make good engineering decisions about deployment of new capacity, or moving capacity that currently exists. You can't afford to guess at how much the system's components are being used.

❖ ❖ ❖

How do you know how much your system is being used?

The system is engaged in many activities, some of which are not externally visible. Each of these activities can report its usage in whatever manner their developers decide. Having many different mechanisms for reporting usage can easily lead to chaos, and chaos makes it harder to keep the system operational.

Some of the data that you are interested in to support and administer the system consists of raw counts from the hardware or software subsystems. Some of the data needs to be aggregated or somehow processed to be useful.

BILLING [7] collects key information from the HALF CALLS [6] to be able to charge customers for their usage. This information is extremely important, but does not paint a complete picture of the system.

The system generates much information that can help the network engineers put the appropriate amount of switching capacity in an area. The people associated with the engineering functions like to have real data, rather than just working from their hunches. The

[3] An example implementation of this pattern is discussed in [3].

information will also have internal uses, such as SYSTEM INTEGRITY [7] who can use it to assess system state.

Collecting data about the system's usage and activities is not the primary application of the switching system. The amount of time that is required to do this should be minimized.

Therefore,

Create a subsystem that will keep track of counts and measurements from the parts of the system and provide a framework to produce meaningful reports with meaningful data, refer to figure 8. The measurement subsystem will be most useful if it creates information reports for the maintenance staff at regular intervals.

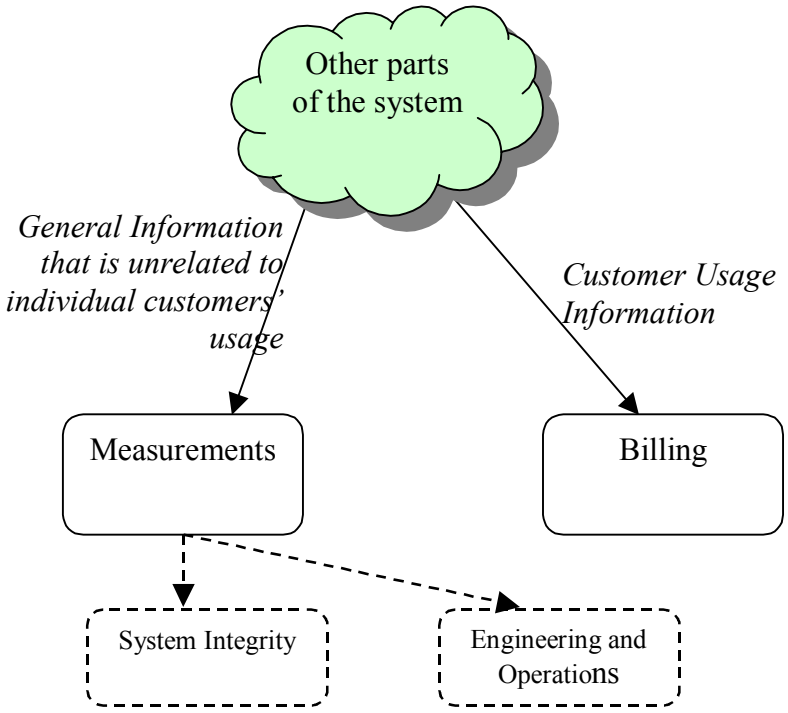


Figure 8

❖ ❖ ❖

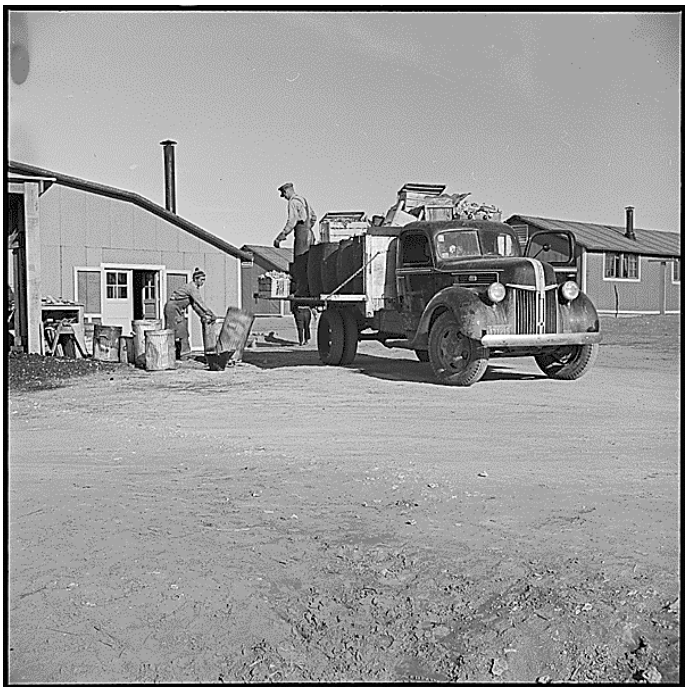
MEASUREMENTS PROVIDE a way for SYSTEM INTEGRITY [7] to check the system activity levels.

Not all of the people charged with system maintenance are, or will be, local. Some way of getting the data "upstream" is required. The NETWORK ELEMENT MANAGER (4) subsystem supports this.

MEASUREMENTS will provide a way to cross check and validate the results of the BILLING [7] system.

TELECOM DATA HANDLING [2] provides additional details useful in building a MEASUREMENTS system.

4. Network Element Manager



SWITCH DATA STORE (1) benefits from PROVISIONING SYSTEM (2). Making sure that the switch data store is working at peek efficiency is something that can be monitored by MEASUREMENTS (3). MEASUREMENTS need to be sent somewhere to be looked at and maintained otherwise their generation was a waste of time.

❖ ❖ ❖

The administrative and operational interfaces of the switch should go somewhere other than /dev/null, the "bit bucket".

If we don't need these subsystems that generate information that will help manage the system effectively then we shouldn't build them. But as their descriptions suggest there are reasons to collect measurements. To make the best decisions about engineering of the network it's elements (the switching systems) should be studied for a period of time.

Once collected by the system they should go somewhere.

We can print them out…but that isn't always useful. A network of several switches will all be collecting measurements. Individual paper measurements from several different switching systems would be difficult to analyze. We could send them to another computer system that will store them and allow later retrieval.

If another computer is involved in collecting the data then why not have it do some other functions, such as providing a convenient interface for a human manager to monitor the switch. And why not adapt it to support several switches at the same time?

Our switch is like an embedded system that has certain interfaces that the primary application sees the telephoning public and others that the primary users don't. For example, the primary throttle interface from a driver to a car's primary computer as well as the interface to the mechanic's diagnostic machine.

Therefore,

Build a computer system, external to the switch that can be used to oversee and administer the operations of the switch as shown in figure 9. This system should support several "target" switches at a time through collecting MEASUREMENT **and provisioning**

related data from these switches. Switch control functions, such as those helpful to provide human oversight of SYSTEM INTEGRITY **[7] should be included in this system.**

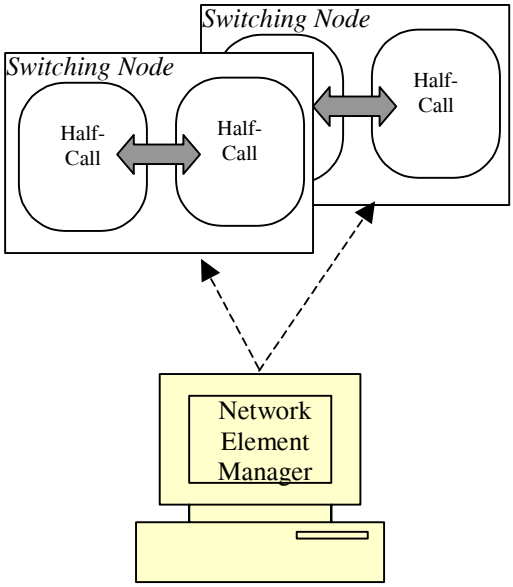


Figure 9

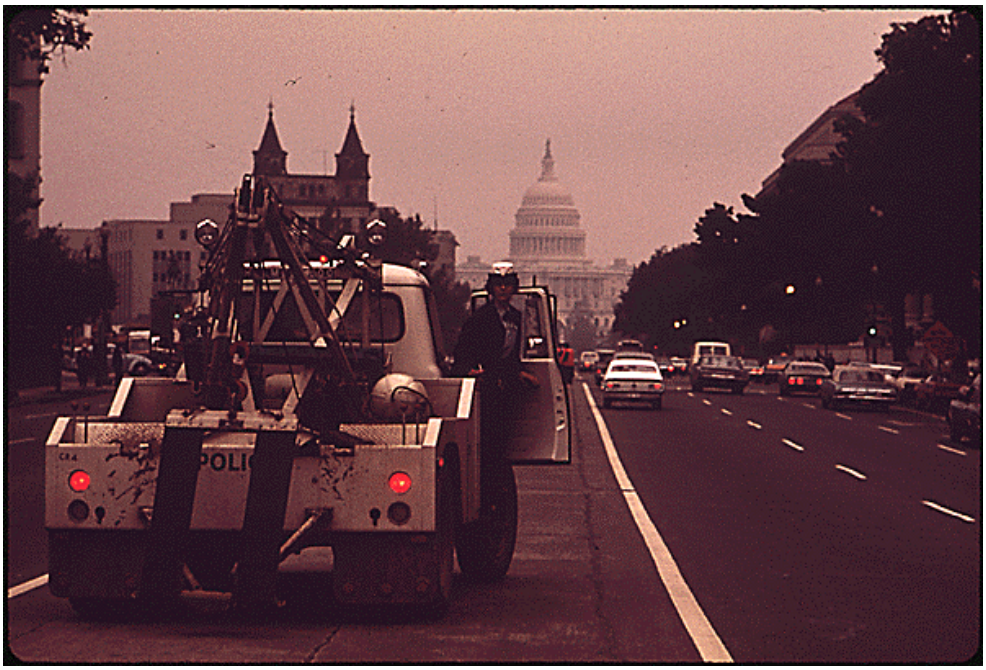
❖ ❖ ❖

Unlike the switching system network elements being monitored and controlled, the primary function of the NETWORK ELEMENT MANAGER will be to monitor the other systems and to interface with human network managers and engineers. Remember that the primary function of the switching system is to process calls or packets, not to deal with MEASUREMENTS. The designers must manage this difference in concerns or else inappropriate responsibilities will be created.

With a NETWORK ELEMENT MANAGER our system can effectively be monitored and administered by humans. The Measurements that it produces can be analyzed over a period of time and in conjunction with its network peers.

Deciding the capabilities of the system in terms of how many switches it should support should not be made in haste. Monitoring too many or via too much measurement data can make this system unusable.

5. Fault Management[4]



SYSTEM INTEGRITY [7] is responsible for monitoring system health and invoking corrective action. When an error is detected action must be taken so that the effects don't ripple throughout the system. Once isolated they should be remedied, by a software or human induced correction being introduced.

During corrective actions the system might generate other errors. Focus and attention on the overall situation

❖ ❖ ❖

How can SYSTEM INTEGRITY'S focus on monitoring and controlling be preserved when it might run into errors during its corrective action?

The first thing that must be done to isolate a fault is to determine what is faulty. In order to do this the system must be able to "look inside" of the other parts of the system to look for discrepancies. This might mean that a component just drop its barriers of encapsulation to allow examination.

This requires clearly defined interfaces between the potential targets of correction and the entity that is trying to locate and correct the fault. There is the potential for very many interfaces.

Another way of handling this problem is for SYSTEM INTEGRITY to ask each component to resolve issues on its own. The problems with this are that if a component is suspected of causing an error then it can't be trusted to take care of itself, or to keep the interests of the system forefront in its operation. Another difficulty is that this results in very much duplication of code – each object (or family of objects) must have its own fault handling capabilities. In a large system where many developers are writing the code, probably on a functional basis, this will mean many implementations of the same things, and thus potentially very many latent faults.

The part of the system that looks for faults to correct must be pretty fault-free itself. Otherwise a fault in it can spread quickly.

[4] An example implementation of this pattern can be found in [9].

Sometimes the system is configurable with a different set of hardware or software components. The ability to add and subtract specialized fault handlers will make the system more reliable, as only the necessary ones need to be loaded. This in turn becomes a system software maintenance problem as the correct software modules must be loaded and available for whatever configuration the system currently has.

There are many techniques to determine if something is faulty. One simple one is through a LEAKY BUCKET COUNTER [1]. This is mentioned to point out the need for data that describes the system's fault history. The MEASUREMENTS () data will be helpful, but it might not really capture the data that will be most interesting for looking at historical faults. MEASUREMENTS data is probably too high level, or too system specific; fault handling data will need to be lower level, more related to individual components.

Therefore,

Create a FAULT MANAGEMENT **subsystem. This subsystem collects the data it needs by interrogating other parts of the system or through its own historical data. It then isolates and perform corrective repairs to the system as needed. Give the** FAULT MANAGEMENT **subsystem the power it needs to take corrective actions.** SYSTEM INTEGRITY **should be small to monitor and invoke the** FAULT MANAGEMENT **subsystem, refer to figure 10, with little risk to the** SYSTEM INTEGRITY **system.**

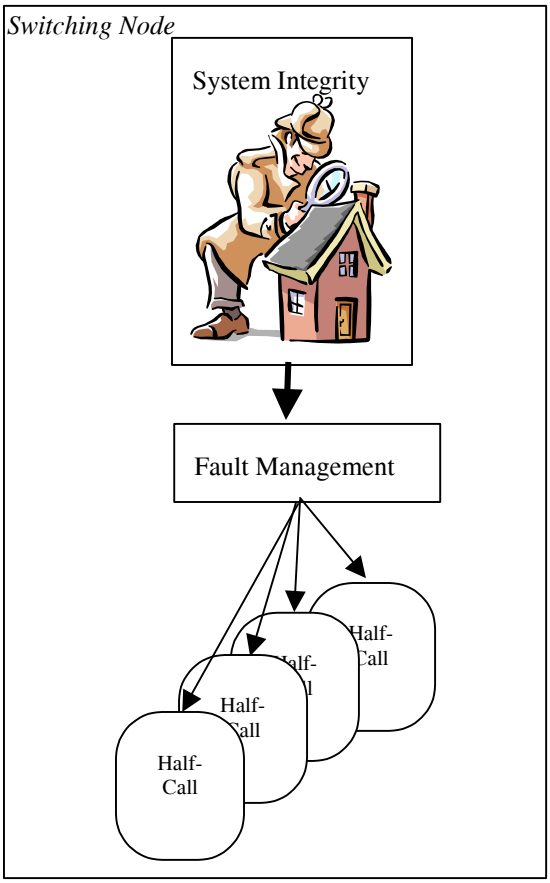


Figure 10

❖ ❖ ❖

The FAULT MANAGEMENT system needs to be simple enough to work even if parts of the system are incapable of working. It must also be able to examine and repair itself.

6. Acknowledgments

All photographs used courtesy of the National Archives and Records Administration of the USA

Thanks to John Letourneau who served as the SugarLoaf PLoP shepherd. Thanks go to my SugarLoaf PLoP Writers' Workshop Group.

7. Pattern Thumbnails

PATTERN	*Reference (Single digits are in this work)*	*Pattern Intent*
BILLING	[7]	Keep records of usage in a centralized object.
FAULT MANAGEMENT	5	A specialized subsystem should quickly handle errors and failures by isolating and treating faults.
HALF CALL	[6]	Use a 2 part (half call) model for call processing.
MAINTENANCE SOFTWARE	[7]	Identify and isolate faults before they are encountered in an operational system.
MEASUREMENTS	3	Provide a single object to collect counts and measurements and then distribute them to concerned parties.
MINIMIZE HUMAN INTERVENTION	[1]	Design the system so that human intervention is not required.
NETWORK ELEMENT MANAGER	4	Provide a single point to consolidate human interaction with switches for both integrity and measurement purposes.
PEOPLE KNOW BEST	[1]	Provide a way for an expert human to override the system's automatic responses.
REASSESS OVERLOAD DECISION	[5]	Conditions change and the system should periodically reassess its decisions to see if they are still valid.
SELF PROTECTION	[7]	In the face of too much incoming traffic, try to push traffic back to neighbors and thus keep your own sanity.
SHED WORK AT THE PERIPHERY	[6],[8]	Try to prevent work from reaching the core of the system; stop it close to the periphery, where fewer system resources will have been expended on it.
SICO FIRST AND ALWAYS	[1]	Give SYSTEM INTEGRITY the power and ability to handle the situations that might arise.
SYSTEM INTEGRITY	[7]	Provide an object that will watch for abnormal system activity and will initiate necessary reactions.
TELECOM DATA HANDLING	[2]	Telecom systems have some unique attributes for their measurements.

8. References

[1] Adams, M. J. Coplien, R. Gamoke, R. Hanmer, F. Keeve and K. Nicodemus. 1996. "A Pattern Language for Improving the Capacity of Reactive Systems" in **Pattern Languages of Program Design — 2**, edited by J. Vlissides, J. Coplien and N. Kerth. Reading, MA: Addison-Wesley Publishing Co.

[2] DeLano, D. 1998. "Telephony Data Handling" in Proceedings of PLoP 1998 Conference.

[3] Greene, T., D. Haenschke, B. Hornbach and C. Johnson, 1977. "No 4 ESS: Network Management and Traffic Administration." **Bell System Technical Journal**, vol. 56, no. 7, Sept., 1977: 1169-1201.

[4] Giunta, J. A., S. F. Heath III, J. T. Raleigh, M. T. Smith, Jr., 1977, "No. 4 ESS: Data/Trunk Administration and Maintenance." **Bell System Technical Journal**, vol. 56, no. 7, Sept., 1977: 1203-1237.

[5] Hanmer, R. 2000. "Real Time and Resource Overload in Proceedings of PLoP 2000 Conference.

[6] Hanmer, R. 2001. "Call Processing" in Proceedings of PLoP 2001 Conference.

[7] Hanmer, R. 2002. "Operations, Administration and Maintenance-1" in Proceedings of PLoP 2002 Conference.

[8] Meszaros, G. 1996 "A Pattern Language for Improving the Capacity of Reactive Systems" in **Pattern Languages of Program Design — 2**, edited by J. Vlissides, J. Coplien and N. Kerth. Reading, MA: Addison-Wesley Publishing Co.

[9] Meyers, M., W. Routt and K. Yoder, 1977. "No 4ESS: Maintenance Software." **Bell System Technical Journal**, vol. 56, no. 7, Sept., 1977: 1139-1167.

[10] Sheinbein, D., and R. P. Weber, 1982. "800 Service Using SPC Network Capability." **Bell System Technical Journal,** vol. 61, No. 7, Part 3, Sept. 1982: 1737-1757.

A Parallel Algorithmic Pattern

Marcos C. d'Ornellas[1]
ornellas@inf.ufsm.br
Grupo de Pesquisas em Processamento de Imagens (PIGS)
Centro de Tecnologia (CT)
Universidade Federal de Santa Maria (UFSM)
Av. Roraima - Campus Universitário
97105-900 Santa Maria - RS - Brasil

Abstract

This paper briefly discuss the general class of algorithms that can be implemented using parallel constructions. Common characteristics of these algorithms are also described in order to provide a generic representation for parallel algorithms. In addition it describes the parallel pattern in terms of image scannings and its relationship within mathematical morphology. Such a pattern is essential for the development of morphological operators and operations. Examples of the application of the parallel generic pattern are given for both scalar lattices and non-scalar lattices. Scalar lattices are used to give a parallel pattern representation for real values, parabolic morphology, and b-bit integers. Non-scalar lattices are restricted to color lattices. Each case study presented in this paper match the generic representation of the parallel pattern.

1 Name

Parallel

2 Intent

Software developers in image processing are often confronted with questions concerning the design and implementation of algorithms. The reason for the questions is that developers as well as end users rarely share the same knowledge. In turn, what an algorithm precisely does remains unclear. Specific implementation details and different pixel types, not taken into account in the initial phases of the design, are examples of the problems that occur.

A mathematical framework for the developing of image processing algorithms is mathematical morphology [22]. The available arithmetical and relational operators make the implementation of its basic operations (erosion, dilation, opening and closing) an easy task. However, when algorithmic implementations are derived from the straight mapping of morphological operators such as erosion and dilations, the computational complexity is known to be

higher since it contains nested loops. Without resorting to parallel computers, the computational complexity due to the number of pixels in an image cannot be reduced. Therefore, the only way to reduce computational complexity significantly is through the application of efficient neighborhood operations.

The parallel pattern solves a well-known algorithmic design problem in image processing, that is, how to keep a one-to-one relationship between mathematical formulation and its effective implementation. The mathematical formulation explains the way it should be constructed and how data is manipulated. This pattern acts as a first step towards the design and implementation of morphological image operators under the complete lattice framework. The generic pattern definition is suggested in order to characterize the basic building blocks given by the iterator, pixel lattice, and adjunction. The mathematical approach leads to the design of consistent methods that address all the requirements needed for the implementation of the parallel pattern.

The parallel pattern is intended for those who wish to facilitate code reuse in software projects in image processing and computer vision. This pattern gives rise to a family of algorithms, which are often used to build useful constructions and to solve a general list of problems that often ocurr. This pattern assumes the reader is reasonably proficient in image processing or mathematical morphology, and have some experience in generic programming as well.

3 Motivation

Designing software from scratch is the choice when the problem at hand requests unconventional solutions, or when the operational circumstances are different from anything seen in the past. In general, when the problem is very difficult one can imagine that everything has to be in line to reach an acceptable solution. For morphological image processing, this is no longer a good engineering practice. Implementing software this way will generate code that will be dedicated to the data, to the problem, and to the platform.

Combinations of various software design techniques such as component-based programming, object-oriented programming and generic programming have proved effective for code optimization for frequently performed operations in mathematical morphology. Due to the versatility of generic programming, one does not need to invest time in specific implementations for every variation of a general algorithm. This means that a small number of generic algorithms are sufficient to implement the functionality of a traditional image processing library.

The straight mapping of erosions and dilations in the pixel and image lattices into algorithm representations are the necessary requirements to introduce the generic pattern in this paper. Such pattern must be fully characterized by its basic building blocks. A specialization of this pattern, namely parallel pattern, is used to describe parallel algorithm implementations. Other specializations of the generic pattern are described in [5].

4 Applicability

This paper proposes a generic model for realizing and using parallel design patterns in mathematical morphology. The term parallel is used to represent the application independent, reusable set of attributes associated with a pattern. It turns out that the model is an ideal candidate for object-oriented style of design and implementation. It can be implemented in C++ with

the Sandard Template Library (`STL`) without requiring any language extension. The generic model, together with the object-oriented and library-based approach, facilitates extensibility.

Algorithms that use a parallel pattern share a certain number of characteristics, which are listed in the following:

- **Fast Pixel Manipulation**: in the lowest level macro of a function implementation all the pixel manipulation is performed with pointers. Typically there is at least one pointer to source image(s) and a pointer to a destination image;
- **Nested Loops**: for a rectangular image the method used to traverse the image data depends on the operation being applied. For an operation using x and y coordinates, two nested loops are used with the pointers being incremented in the inner loop. For a neighbourhood operation, surrounding pixel data is accessed by adding and subtracting offsets from the pointers. An offset of 1 allows data one pixel to the left and right to be accessed and an offset of width allows data one pixel above and below to be accessed. This easily extends for larger neighbourhoods;
- **Image Shape**: for an arbitrarily shaped image there is a boundary structure associated with the image's image size window. The actual data stored corresponds to the rectangular bounding area which encloses the shape of the image. There is a some space redundancy here, as more data is stored than strictly required but this is offset by the fact that the simple pointer manipulations just described can still be used for arbitrarily shaped images, leading to simpler, faster and more robust code. The only difference need be in the loops which control the traversal of the data. These loops consist of an outer loop (to follow the linked list in the boundary structure) and an inner loop to move along each horizontal line segment. Treating the image as rectangular allows the same code to be used for all cases. The cost of processing the extra pixels needs to be weighed against the extra complexity of the traversal loops. This depends on the cost of processing an individual pixel and the particular shape of the image;
- **Origin Included in the Support**: algorithms assume that the origin is included in the structuring element support, which is important because only anti-extensive erosions and extensive dilations are considered.

5 Structure

Mathematical morphology aims at analyzing geometrical properties of objects. The analysis is quantitative in order to provide a complete framework for describing spatial organizations. So far, the use of such a framework has allowed for the development of a class of morphological algorithms to deal with binary, grey-scale, and recently also for color images.

The first study on a general framework underlying mathematical morphology was proposed in [23]. It was shown that such framework could be achieved if one starts from the assumption that the object space is a complete lattice. This idea has been carried further by various people [1] [12] [16] [20] [21].

Recently, much of the morphological software-related practice has evolved from merely a set of fundamental operators, such as erosions and dilations, to the concept of an entire software package. Morphological tools within these packages usually range from classical filters [14] [15] to watersheds [2].

Consider the most general structure of erosion and dilation for images $f \in \mathrm{Fun}\,(\mathbb{E}, \mathcal{L})$, where $\mathbb{E}$ is a discrete space and $\mathcal{L}$ is the complete pixel lattice. The basic morphological operators are erosion and dilation written as:

$$(\varepsilon f)(x) = \bigwedge_{y \in \mathbb{E}} \varepsilon(x, y)(f(y)) \tag{1}$$

and

$$(\delta f)(x) = \bigvee_{y \in \mathbb{E}} \delta(x, y)(f(y)) \tag{2}$$

where $\varepsilon(y, x)$ and $\delta(x, y)$ are erosions and dilations in the pixel lattice $\mathcal{L}$. Equations 1 and 2 form an adjunction on $\mathcal{L}$.

A pixel lattice $\mathcal{L}$ is characterized by the value set V (the collection of all possible pixel values) and the partial ordering on V (denoted $\leq$). It is explicitly described by the tuple: $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$, where $\bigvee$ and $\bigwedge$ are the supremum (least upper bound) and infimum (greatest lower bound) operators respectively. Observe that this lattice includes additional elements other than V and $\leq$ since they are needed in the implementation of morphological image operators. In the discrete case, only the supremum and infimum of a finite number of values are needed. Then, $\bigvee$ and $\bigwedge$ can be rightfully called maximum and minimum respectively. $\bigvee_{\mathcal{L}}$ and $\bigwedge_{\mathcal{L}}$ are also needed as the lattice supremum and infimum.

A straightforward implementation of equations 1 and 2 is called parallel because the order in which the locations $x \in \mathbb{E}$ are being accessed has no influence on the result. The same algorithm can be implemented in a sequential hardware. It can also run on parallel hardware where every processor is responsible for processing a part of the image [17].

Consider, for instance, the dilation in the image lattice:

$$(\delta f)(x) = \bigvee_{y \in \mathbb{E}} \delta(x, y)(f(y)) \tag{3}$$

where $f \in \mathrm{Fun}\,(\mathbb{E}, \mathcal{L})$, and $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. The domain $\mathbb{E}$ of the images in this paper are rectangular subsets of $\mathbb{Z}^d$ where d is the dimension of the image f. Note that the pixel lattice $\mathcal{L}$ is a complete lattice, i.e. a partial ordered set (*poset*) with extra requirements including a supremum and an infimum.

The pixel lattice dilation $\delta(x, y)$ is parameterized by two locations in the image domain $\mathbb{E}$. In words, equation 3 dictates that at every location x, $(\delta f)(x)$ is calculated by looping over all locations $y \in \mathbb{E}$ and dilating $f(y)$ with the pixel lattice dilation $\delta(x, y)$ resulting in $\delta(x, y)(f(y))$. Then the supremum of all these values is taken as the result. An algorithmic representation for $h = \delta f$ is given in figure 1.

6 Participants

The parallel pattern includes the following elements:

- **iterator**: an iterator is a fundamental structure that abstracts the process of moving through a finite set of elements. It allows for the selection of each element of the set without knowing the underlying structure of the set. Using a programming language's

Figure 1 Parallel dilation algorithm representation demonstrating the straight mapping from its mathematical equation.

for all $(x \in \mathbb{E})$ **do**
  $h(x) = \bigwedge_{\mathcal{L}}$
  for all $(y \in \mathbb{E})$ **do**
    $h(x) = \bigvee\{h(x), \delta(x,y)(f(y))\}$
  end for
end for

terminology, an iterator can be considered as an abstract pointer to an element in the set. Algorithms use iterators to operate on data structures (containers). Iterators set bounds for algorithms, regarding the extent of the container. This is a powerful feature, partly because it allows for learning a single interface that works with all containers, and partly because it allows containers to be used interchangeably.

Iterators are more complex at the implementation level. Therefore, there is a need for a generalization of the iterator concept for two or more dimensions. That is, traversal functions must be provided to tell the iterator in which coordinate direction to move. For instance, the algorithm in figure 1 would be better characterized with a two-dimensional iterator to move through all the elements $x \in \mathbb{E}$ and another to move through all elements $y \in \mathbb{E}$. Since images are bounded and have a finite number of elements, algorithm characterizations using one-dimensional iterators are also feasible;

- **pixel lattice**: The mathematical theory states that only two elements are needed to describe complete lattices, i.e. the value set V and the partial ordering relation $\leq$, yielding $\mathcal{L} = (V, \leq)$. Note, however, that a decision was made to have all the operators and constants explicitly available. Therefore, the supremum $\bigvee$ and infimum $\bigwedge$ operator as well as the lattice supremum $\bigvee_{\mathcal{L}}$ and infimum $\bigwedge_{\mathcal{L}}$ are included in the lattice definition $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$.

  It is the responsibility of the software developer (who instantiates a particular pixel lattice) to make these operators and constants explicitly. For example, the algorithm in figure 1 needs the appropriate values for the pixel lattice infimum $\bigwedge_{\mathcal{L}}$ and the supremum operator $\bigvee$;

- **adjunction**: the image dilation $(\delta f)(x)$ included in the algorithm in figure 1 is characterized by the pixel lattice dilation $\delta(x, y)$. With any pixel lattice dilation, a unique pixel lattice erosion is associated. Such a pair of adjoint operators is called an adjunction $\mathcal{A} = (\varepsilon, \delta)$.

These three elements, when joined together, form the building block representation of the generic parallel pattern. The generic implementation of the algorithm in figure 1 is really all that is needed to implement morphological operators on $\mathrm{Fun}(\mathbb{Z}^d, V)$. If an image operator does not fit into this scheme, it is not placed into mathematical morphology theory.

7 Collaborations

- **Iterator** is coupled with pixel scanning methods acting on containers (e.g. vectors). All data are accessed in parallel by one-dimensional iterators for every pixel scan. Pixels

scans might differ from each application even though they must produce the same results;

- **Pixel Lattice** defines the value set used for the data involved. In other words, it determines a set of common rules to be used in the parallel pattern for a family of data types;
- **Adjunction** is closely related with the way pixel scannings are conducted. Therefore, it works together with the **iterator**.

8 Consequences

- The applicability of a general theory for mathematical morphology, well-established in the complete lattice framework, is needed for the construction of the generic pattern for morphological algorithm representations. The generic pattern is made of the three basic building blocks namely the iterator, the pixel lattice and the adjunction and was applied in the effective implementation of algorithms and applications. It is known that many algorithms in the literature are all reduced to the parallel pattern.
- The implementation of generic algorithms for morphological image operators still need a nested loop `for` construction to process the image data by means of pixel scans, a typical characteristic of the parallel pattern. Consider the nested loop within the dilation algorithm. If the time to execute $h(x)$ is independent of any indexing variable, the complexity is $\mathcal{O}(\#N_\delta)$, $\#N_\delta$ being the number of pixels in N_δ respectively. Due to the performance penalty its high complexity produces, this approach requires additional strategies to reduce such complexity.
- Flat operators are defined in such a way that pixel lattice erosions and dilations are not taken into account. They play an important role in mathematical morphology for its theoretical aspects and practical use. Therefore, flat morphology introduces changes in the representation of the parallel pattern with respect to the adjunction since only the minimum and maximum are needed to characterize erosions and dilations in the image lattice over a specified neighborhood. Flat morphology works with structuring elements of all shapes and sizes.
- A generic programming approach might be applied to morphological image operators based on pixels scans like thinnings and thickenings, parabolic morphology and skeletons. Generic programming tools as `C++` with `STL` applies a more evolutionary and experimental approach to morphological algorithm development. The proposed parallel pattern enhances morphological algorithm reuse.
- Morphological operators are often used with large structuring elements, which makes them highly inefficient for practical purposes. Alternative solutions for the lack of speed of flat morphological operators and applications derived from the parallel pattern employ structuring element decomposition. Other fast algorithms for flat morphology based on bitmapped representations and linear structuring element decompositions uses van Herk's algorithm.

9 Implementation

Morphological operator design must comply with the complete lattice framework theory, i.e. algorithmic implementations must be tied to the generic pattern representation that includes the iterator, the pixel lattice, and the adjunction. Iterators in the generic representation for parallel algorithms are not influenced by the containers since they do not influence or change the final result, which must be the same for all set of one-dimensional or two-dimensional iterators. Later, the iterator is combined with the pixel lattice and the adjunction, producing the parallel pattern.

The following implementation issues are relevant for the parallel pattern:

Containers: Vectors are often used in the implementation of algorithms that are governed by n-dimensional pixel scans. Items stored in a vector structure are pixel addresses or pixel values, but the vector must be able to handle other features according to the implementation. The size of the vector can be fixed or controlled dynamically.

`C++` **and** `STL`**:** The STL is a relatively small library which achieves a remarkable degree of reuse through its basis in the principles of generic programming and its use of `C++` templates. Because of this, it has a particularly clear shape. The distinction between containers, iterators and algorithms is its most striking structural feature: dynamically, the way a container delivers iterators which are then used by algorithms is a consistent and fundamental pattern of use.

10 Algorithm Samples and Usage

Applications of the parallel pattern are widely used in image processing. Examples are given in this section with respect to scalar lattices and non-scalar lattices. Such applications are grounded on the theoretical pattern concepts and pattern description and can be easily seen as a natural mapping from morphological formulas into algorithm representation. Examples of the parallel pattern are expressed in terms of iterator, pixel lattice, and adjunction. These examples are restricted to changes in the generic pattern with respect to pixel lattice and adjunction. Additional attention is given to color lattices in the sense that it is still not a completely solved concept in mathematical morphology.

Real Values: Real valued images $f \in \mathrm{Fun}(\mathbb{E}, \mathbb{R})$ are considered in this section. The definition of an erosion in the image lattice is given by:

$$(\varepsilon f)(x) = \bigwedge_{y \in \mathbb{E}} \varepsilon(x, y)(f(y)) \tag{4}$$

where $\varepsilon(x, y)$ is the erosion in the pixel lattice $\mathcal{L}$.

A translation invariant definition is derived from equation 4, yielding:

$$(\varepsilon f)(x) = \bigwedge_{y \in \mathbb{E}} \varepsilon_t(y - x)(f(y)) \tag{5}$$

where $\varepsilon_t(y-x)(f(y)) = f(y) - g(y-x)$, where g denotes the structuring element. Such definition leads to the classical structural erosion given by:

$$(\varepsilon f)(x) = \bigwedge_{y \in \mathbb{E}} \{f(y) - g(y-x)\} \tag{6}$$

If the definition in equation 6 is used for finite image domains, the border effect only shows up in case one implements g as a function $g : \mathbb{E} \rightarrow V$ as well. The confusion comes from the fact that the notion of translation of images (through $g(y-x)$) is implicitly defined, which is only valid on infinite domains.

The classical structural erosion described by equation 6 can be seen as a special function lattice since it has a purely geometrical interpretation [6] [7] [10] [24]. This interpretation matches the topographical view for a two-dimensional Euclidean space, where points are given by triples of coordinates; the first two locate the position in the two-dimensional support set and the third coordinate gives the height.

The parallel pattern for the real values is presented as follows:

- **iterator**: parallel algorithms using iterators (e.g. `for` loops) are the most classical not only in conventional image processing but also in mathematical morphology. An algorithm is said to be parallel if the pixel values in the neighborhood $N(x)$ are taken in the original image. Implementations derived from this pattern are independent of the order of image scannings;
- **pixel lattice**: all the elements must be explicitly defined by $\mathcal{L} = (\mathbb{R}, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$.
- **adjunction**: the adjunction $\mathcal{A} = (\varepsilon, \delta, N_\varepsilon, N_\delta)$ is given by the classical definitions of structural erosions and dilations:

$$(\varepsilon f)(x) = \bigwedge_{y \in N_\varepsilon(x)} \{f(y) - g(y-x)\} \tag{7}$$

$$(\delta f)(x) = \bigvee_{y \in N_\delta(x)} \{f(y) + g(x-y)\} \tag{8}$$

where $N_\varepsilon(x) = \{y \mid g(-y) \neq -\infty\}$ and $N_\delta(x) = \{y \mid g(y) \neq +\infty\}$.

Parabolic Morphology: Consider A to be a $n \times n$ symmetric positive definite matrix. The parabolic or quadratic structuring function (`QSF`) associated with this matrix is denoted as q_A and is given by $q_A(x) = -\frac{1}{2}\langle x, A^{-1}x\rangle$. Note that the simplest `QSF` is the rotationally symmetric one: q_I with I being the identity matrix ($q_I(x) = -\frac{1}{2} \parallel x \parallel^2$).

If two `QSF's` q_A and q_B are dilated, then $q_A \oplus q_B = q_{A+B}$, which implies that the class of `QSF's` is closed under dilation. A `QSF` q_A can be dimensionally decomposed along the eigenvectors of the matrix A. For a diagonal matrix, the eigenvectors are along the x and y axis and thus the one-dimensional dilations are along the rows and columns. Note that the `QSF` q_I is the unique rotational invariant function that can be dimensionally decomposed with respect to dilation [2] [25][18] [26].

[2] This is a property that it shares with the Gaussian function being the unique isotropic kernel that can be dimensionally decomposed with respect to convolution.

The erosion scale function $F^{\ominus}(x,\rho)$ is obtained by eroding the original image f with structuring function q^{ρ} : $F^{\ominus}(x,\rho) = (f \ominus q^{\rho})(x)$. Dually, the dilation scale function $F^{\oplus}(x,\rho)$ is obtained by dilating the original image f with structuring function q^{ρ} : $F^{\oplus}(x,\rho) = (f \oplus q^{\rho})(x)$. For increasing values of ρ the scale in the resulting image, (either $F^{\ominus}$ or $F^{\oplus}$) decreases as the original image f is processed with a structuring function with increasing size. Therefore, $F^{\ominus}$ and $F^{\oplus}$ are two sequences of images, each derived from the original image and ordered with respect to their internal scale. A parabolic scale-space was proposed in [26] based on those scaling properties.

The parallel pattern for parabolic morphology is the one presented as follows:

- **iterator**: the iterator is the same for the real values (using `for` loops) and does not need to be changed in this case;
- **pixel lattice**: all the elements must be explicitly defined by $\mathcal{L} = (\mathbb{R}, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$;
- **adjunction**: the classical adjunction, defined by $\mathcal{A} = (\varepsilon, \delta,$ is extended to $\mathcal{A} = (\varepsilon, \delta, \mathbb{E})$, since $N_{\varepsilon} = N_{\delta} = \mathbb{E}$. Parabolic erosions and dilations are governed by the equations:

$$F^{\ominus}(x,\rho) = (f \ominus q^{\rho})(x) \tag{9}$$
$$F^{\oplus}(x,\rho) = (f \oplus q^{\rho})(x) \tag{10}$$

where $q(x) = -\frac{1}{4} \parallel x \parallel^2$, a unique structuring function that can be separated by dimension.

b-bit Integers: A b-bit integer is the term used to express that an image can be mapped using a byte representation (e.g. 8-bits for a grey-scale image). Consider the set $\mathcal{S}$ being one of $\mathbb{N}, \mathbb{Z}, \mathbb{R}, \ldots$ and assume that l_{inf} and l_{sup} are the lattice infimum and lattice supremum respectively. If the set $\mathbb{N}$ is assigned to $\mathcal{S}$, one obtains a finite set with values between $[0, \ldots, K], K \in \mathbb{N}$ which is not closed under addition and subtraction. Therefore, by approaching this problem using truncated values between $l_{inf} = 0$ and $l_{sup} = K$, the adjunction property is lost.

In order to solve this problem, a new formulation for erosions and dilations for grey-scale images was introduced in [11] [12] [21]:

$$(\varepsilon f)(x) = \bigwedge_{y \in N_{\varepsilon}(x)} \{f(y) \dot{-} g(y-x)\} \tag{11}$$
$$(\delta f)(x) = \bigvee_{y \in N_{\delta}(x)} \{f(y) \dot{+} g(x-y)\} \tag{12}$$

The dot operators $\dot{-}$ and $\dot{+}$ are given by:

$$a \dot{-} b = \begin{cases} K & \text{if } a = K; \\ K & \text{if } (a < K \ \textit{and} \ a - b > K); \\ 0 & \text{if } (a < K \ \textit{and} \ a - b \leq 0); \\ a - b & \text{if } (a < K \ \textit{and} \ 0 \leq a - b \leq K). \end{cases} \tag{13}$$

$$a \dot{+} b \;=\; \begin{cases} 0 & \text{if } a = 0; \\ 0 & \text{if } (a > 0 \;\; and \;\; a + b \leq 0); \\ K & \text{if } (a > 0 \;\; and \;\; a + b > K); \\ a + b & \text{if } (a > 0 \;\; and \;\; 0 \leq a + b \leq K). \end{cases} \tag{14}$$

where $a \in [0, \ldots, K]$ and $b \in \mathbb{R}$.

The parallel pattern for the b-bit integers is the one presented as follows:

- **iterator**: the iterator is just like the same for the real values (using `for` loops) and does not need to be changed in this case;
- **pixel lattice**: all the elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. For instance, for binary and grey-scale images:

$$\begin{aligned} \mathcal{L} &= ([0, 1], \leq, \bigvee, \bigwedge, 1, 0) \\ \mathcal{L} &= ([0, 255], \leq, \bigvee, \bigwedge, 255, 0) \end{aligned}$$

  Note that other integer intervals can be used to compose a new pixel lattice;
- **adjunction**: the adjunction $\mathcal{A} = (\varepsilon, \delta, N_\varepsilon, N_\delta)$ follows the classical definition except for the fact that erosions and dilations are governed by equations 11 and 12 respectively.

Color Lattices: In color morphology, operators that are applied to the whole image can also be applied to the components separately, because the filters commute with infimum and supremum respectively. This kind of marginal processing is equivalent to the vectorial approach defined by the canonic lattice structure when only supremum and infimum operators and their compositions are involved and induces a totally ordered lattice as presented in [8] [9]. Consider two images f and f', each one made up by a number of i components. Therefore:

$$f \leq f' \iff f(i) \leq f'(i), \forall i \in 1, \ldots, m$$

With these relations, the supremum of a family $\{f_j\}$ is the vector $\bigvee f$ where each component $\bigvee f(i)$ is the supremum of the $\{f_j(i)\}$. Respectively, the infimum of a family $\{f_j\}$ is the vector $\bigwedge f$ where each component $\bigwedge f(i)$ is the infimum of the $\{f_j(i)\}$. However, this morphological procedure fails because every color can be seen as a vector, which cannot be totally ordered and so the supremum or infimum of the two is a mixture of both the colors. Using this procedure, one obtains the same results as the simple marginal processing of the data and new colors not contained in the input image will appear.

Another approach, grounded on the use of a vector transformation from $\mathbb{R}^m$ into $\mathbb{R}^Q$ followed by a marginal ordering on $\mathbb{R}^Q$ was defined in [8]. If $Q > 1$, the marginal ordering induces a partial ordering on the vectors. $Q = 1$ is required to obtain a total ordering with an h-adjunction. The important point is to transform the image data by means of a surjective transformation h, which is better suited for the morphological approach. However, h is neither bijective nor injective. A major drawback in practice is that the extrema of a family $\{f_j\}$ are not necessarily unique. Therefore, many different vectors can lead to

the same result $h(f) = \bigwedge_i \{f_i\}$ or $h(f) = \bigvee_i \{f_i\}$ for erosions and dilations respectively. Consequently, there is a need for a new equivalence relation $=_h$.

To extend the vector approach to color images, it is necessary to define an order relation who orders the colors as vectors, rather than ordering the individual components as suggested in [13]. This can be done using reduced ordering. This kind of ordering imposes a total ordering relationship that can be accomplished by the lexicographical ordering [4] [3] as reported in [28]. The flat structuring element for the vector morphological operations defined here is the set g, and the scalar-valued function used for the reduced ordering is $r : \mathbb{R}^3 \rightarrow \mathbb{R}$. Erosions and dilations are given by:

$$(\varepsilon f)_r(x) = \bigwedge_{y \in N_\varepsilon(x)} \{f_r(y) - g_r(y - x)\} \tag{15}$$

$$(\delta f)_r(x) = \bigvee_{y \in N_\delta(x)} \{f_r(y) + g_r(x - y)\} \tag{16}$$

where f_r and g_r are the coded representations for f and g under the reduced ordering r.

Erosions and dilations in conjunction with a total ordering induced by a reduced ordering generate no new colors. Under the total ordering relation, the infimum or supremum is one of the actual colors. Therefore, the only colors in the output image are those obtained from translations of the input ones.

Since the output of the vector filter depends on the scalar-valued function used for reduced ordering, the selection of this function provides flexibility in incorporating spectral information into the multivalued image representation. For example, linear combinations of the tristimulus values can be used. This can be written as:

$$t \leq t' \leftrightarrow \begin{cases} r(t) = a_1 t_1 + a_2 t_2 + a_3 t_3 \\ r(t') = a_1 t'_1 + a_2 t'_2 + a_3 t'_3 \\ r(t) \leq r(t') \end{cases} \tag{17}$$

if the image is filtered in the `RGB` color model. For the case where $a_1 = 0.299, a_2 = 0.587$, and $a_3 = 0.114$, $r(t)$ and $r(t')$ become the luminance component. Multiscale opening in this case would suppress bright objects at each scale. The values of a_1, a_2, and a_3 can also be selected to enhance or suppress specific colors. For example, if $a_1 = 1, a_2 = 0$, and $a_3 = 0$, then the effect of a multiscale opening would be to suppress objects with high `red` content. The same holds to the `green` and `blue` color channels.

The parallel pattern for the non-scalar lattice working on color images is given as follows:

- **iterator**: just like for all the examples in this paper, the iterator (e.g. `for` loops) structure does not change;
- **pixel lattice**: all the elements must be explicitly defined by $\mathcal{L} = (V, \leq, \bigvee, \bigwedge, \bigvee_{\mathcal{L}}, \bigwedge_{\mathcal{L}})$. For instance, for a `RGB` color image:

$$\mathcal{L} = (([0, 255], [0, 255], [0, 255]), \leq, \bigvee, \bigwedge, [255, 255, 255], [0, 0, 0])$$

[3] An ordered pair (i, j) is lexicographically earlier than (i', j') if either $i \leq i'$ or $i = i'$ and $j \leq j'$. It is lexicographic because it corresponds to the dictionary ordering of two-letter words.

Observe that the partial ordering $\leq$ can vary in accordance with the scalar valued functions introduced previously. Indeed, any other partial ordering can be used, leading to different output results;

- **adjunction**: the adjunction is represented by $\mathcal{A} = (N)$ since color operators are extensions of flat operators using a flat and symmetric structuring element. Erosions and dilations are given by:

$$(\varepsilon f)_r(x) = \bigwedge_{y \in N_\varepsilon(x)} \{f_r(y)\} \tag{18}$$

$$(\delta f)_r(x) = \bigvee_{y \in N_\delta(x)} \{f_r(y)\} \tag{19}$$

These equations are vectorial representations for flat structural erosion and dilation of an image f with r components.

Some results derived from the application of the parallel pattern in color erosions and dilations are provided in figures 2 and 3 respectively. Take notice the results for two ordering relationships: bitmix and maximum.

Figure 2 RGB Erosion showing the results for a flat 5×5 structuring element: input image(a), bitmix(b), and maximum(c).

(a) (b) (c)

Figure 3 RGB dilation showing the results for a flat 5×5 structuring element: input image(a), bitmix(b), and maximum(c).

(a) (b) (c)

11 Known Uses

Several parallel operators and operations have been proposed in the literature leading to a large collection of applications, which are characterized as follows:

fast algorithms for flat morphology: A fast algorithmic implementation for basic binary morphological operations on general-purpose sequential computers was proposed in [27] and works with structuring elements of arbitrary size and shape. Rather than representing binary images as bit-planes inserted in grey-scale images, the bitmap representation was used. This representation is very efficient both in terms of memory requirements and in terms of algorithmic efficiency because the CPU operates on 32 pixels in parallel. A combination of a bitmapped representation with the logarithmic decomposition of structuring elements leads to very fast algorithms for the basic morphological operators;

parabolic morphology: The morphological scale-space is generated by dilations (erosions) with a parabolic structuring function of increasing width. Most often, extending the scalar parameter to the entire real axis combines the erosion and dilation scale spaces. Negative scales are interpreted as the erosion scales (background) and positive scales as the dilation scales (foreground). Foreground and background are treated distinctly, which make a great difference from linear theory since convolution cannot have this as it is a self-dual operator [25];

color morphology: To extend the vector approach to color images, it is necessary to define an order relation who orders the colors as vectors, rather than ordering the individual components as suggested in [13]. This can be done using reduced ordering. This kind of ordering imposes a total ordering relationship that can be accomplished by the lexicographical ordering [4] Erosions and dilations in conjunction with a total ordering induced by a reduced ordering generate no new colors;

thinnings: Thinning is the method used to remove selected foreground pixels from a binary image. It is generally used after an edge detection operation to tidy up by reducing all the lines to one pixel thickness. The operation is quite similar to the Hit-or-miss operation. The structuring element is moved over the input image pixels and if the foreground and background pixels of the structuring element and the image are the same then the input pixel below the origin of the structuring element is set to the background pixel value. Otherwise it is left unchanged [3];

thickenings: Thickening is the operation of adding to an image pixels with the same configuration. It really grows certain pixels of the foreground. The structural element is superimposed on the input image and if the foreground and background pixels of the structuring element match that of the image, the input pixel is set to the foreground pixel value. Otherwise it is left unchanged. Thickening is the dual of thinning [3];

skeletons: Shape representation is an important image analysis task which can be used for contour coding and feature extraction. The morphological skeleton is a geometrical shape description by means of maximal inscribed structuring elements. The form of the structuring element is usually chosen a priori, and such process can be implemented using a parallel pattern.It provides improved progressive contour transmission and the extraction of shape features [19];

polygon filling: Polygon filling is often executed in the image or frame buffer. It is assumed that a polygon with proper closed borders are given and that inside the polygon no pixel has the color value with which it is to be filled. For the seed fill algorithm, we need a starting point which is inside the polygon. Starting at this seed position, the algorithm proceeds in a way determined by the pixel scans and sets each pixel to the required fill color until the border is reached;

robot planning: Research in robot planning has considered the interleaving of planning and execution for example, as well as the issue of planning sensor actions as a way of gaining information for planning. A major theme in robot planning is also the integration of predictive and reactive planning as a way of making activity more resilient in the face of a changing environment. Predictive planning often applies the parallel pattern in the configuration space using a visibility graph and through free space decomposition techniques.

12 Related Patterns

Sequential and Queue-based Patterns[5] are also used in mathematical morphology and in image processing. Although these patterns are more eficient than the parallel pattern, the parallel pattern itself is essential for several applications ranging from parallelized erosions and dilations to thinnings, thickenings and skeletons. A parallel pattern needs to be used when there is a chance to perform several operations at the same time and all results does not impact or produce changes in the final output. Morphological operations, which involves the rotation of the structuring element are known to be time consuming but ar implemented using the parallel pattern.

References

[1] G. J. F. Banon. Formal introduction to image processing. Research report INPE-7682-PUD/43, Instituto Nacional de Pesquisas Espaciais - INPE, São José dos Campos, 2000.

[2] S. Beucher and F. Meyer. The morphological approach to segmentation: the watershed transformation. In E. R. Dougherty, editor, *Mathematical Morphology in Image Processing*, chapter 12, pages 433–481. Marcel Dekker, New York, 1993.

[3] J. Brown and A. Hoger. A morphological point thinning algorithm. *Pattern Recognition Letters*, 17:197–207, 1996.

[4] M. L. Comer and E. J. Delp. An empirical study of morphological operators in color image enhancement. *Proceedings of the SPIE Conference on Image Processing Algorithms and Techniques III, February 10-13, 1992, San Jose, California*, 1657:314–325, 1992.

[5] M. C. d'Ornellas. *Algorithmic Patterns for Morphological Image Processing*. PhD thesis, Univesiteit van Amsterdam, 2001.

[6] E. R. Dougherty. Euclidean grayscale granulometries: Representation and umbra inducement. *Journal of Mathematical Imaging and Vision*, 1(1):7–21, 1992.

[7] E. R. Dougherty and C. R. Giardina. Morphology on umbra matrices. *International Journal of Pattern Recognition and Artificial Intelligence*, 2:367–385, 1988.

[8] J. Goutsias, H. J. A. M. Heijmans, and K. Sivakumar. Morphological operators for image sequences. *Computer Vision and Image Understanding*, 62:326–346, 1995.

[9] C. Gu. *Multivalued Morphology and Segmentation-based Coding*. PhD thesis, Ecole Polytechnique Federale de Laussane - EPFL, Lausanne, Switzerland, 1995.

[10] H. J. A. M. Heijmans. A note on the umbra transform in gray-scale morphology. *Pattern Recognition Letters*, 14:877–881, 1993.

[11] H. J. A. M. Heijmans. Mathematical morphology: Basic principles. In *Proceedings of Summer School on Morphological Image and Signal Processing*, Zakopane, Poland, 1995.

[12] H. J. A. M. Heijmans and C. Ronse. The algebraic basis of mathematical morphology – part I: Dilations and erosions. *Computer Vision, Graphics and Image Processing*, 50:245–295, 1990.

[13] R. Jones and H. Talbot. Morphological filtering for color images with no new colors. *IVCNZ'96 (Image and Vision Computing New Zealand)*, pages 149–154, 1996.

[14] P. Maragos and R. W. Schafer. Morphological filters – part I: Their set-theoretic analysis and relations to linear shift-invariant filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35:1153–1169, 1987.

[15] P. Maragos and R. W. Schafer. Morphological filters – part II: Their relations to median, order-statistics, and stack filters. *IEEE Transactions on Acoustics, Speech and Signal Processing*, 35:1170–1184, 1987.

[16] G. Matheron. Filters and lattices. In J. Serra, editor, *Image Analysis and Mathematical Morphology, Volume II: Theoretical Advances*, chapter 6, pages 35–43. Academic Press, London, 1988.

[17] W. K. Pratt. *Digital Image Processing*. John Wiley and Sons, New York, 1991.

[18] S. Makram-Ebeid R. v. d. Boomgaard, L. Dorst and J. Schavemaker. Quadratic structuring functions in mathematical morphology. In R. W. Schafer P. Maragos and M. A. Butt (Eds.), editors, *Proceedings of the ISMM'96 Workshop Mathematical Morphology and Its Applications to Image and Signal Processing*, pages 147–154. Kluwer Academic Publishers, 1996.

[19] J. M. Reinhardt and W. E. Higgins. Comparison between the morphological skeleton and morphological shape decomposition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 18(9):951–956, 1996.

[20] C. Ronse. Why mathematical morphology needs complete lattices? *Signal Processing*, 21:129–154, 1990.

[21] C. Ronse and H. J. A. M. Heijmans. The algebraic basis of mathematical morphology – part II: Openings and closings. *Computer Vision, Graphics and Image Processing: Image Understanding*, 54:74–97, 1991.

[22] J. Serra. *Image Analysis and Mathematical Morphology*. Academic Press, London, 1982.

[23] J. Serra, editor. *Image Analysis and Mathematical Morphology. II: Theoretical Advances*. Academic Press, London, 1988.

[24] S. R. Sternberg. Grayscale morphology. *Computer Vision, Graphics and Image Processing*, 35:333–355, 1986.

[25] R. v. d. Boomgaard. *Mathematical Morphology: Extensions Towards Computer Vision*. PhD thesis, University of Amsterdam, 1992.

[26] R. v. d. Boomgaard and A. W. M. Smeulders. The morphological structure of images: the differential equations of morphological scale space. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 16:1101–1113, 1994.

[27] R. v. d. Boomgaard and R. v. Balen. Methods for fast morphological image transforms using bitmapped binary images. *Computer Vision, Graphics and Image Processing: Graphical Models and Image Processing*, 54(3):252–258, May 1992.

[28] D. Wood. *Data Structures, Algorithms, and Performance*. Addison Wesley Publishing Company, New York, 1993.

Padrões de Reengenharia Auxiliados por Diretrizes de Qualidade de Software

Gizelle Sandrini Lemos
DC – Universidade Federal de São Carlos
gizelle@dc.ufscar.br

Edson Luiz Recchia
Universidade Anhembi-Morumbi
erecchia@terra.com.br

Rosângela D. Penteado
DC – Universidade Federal de São Carlos
rosangel@dc.ufscar.br

Rosana T. V. Braga
ICMC–Universidade de São Paulo
rtvb@icmc.usp.br

Resumo

Atualmente, a definição de padrões de reengenharia tem sido abordada por diversos autores pela necessidade da existência de diretrizes mais consistentes para guiar na realização desse processo. Aliada a isso, a preocupação com o resultado do processo, ou seja, a qualidade do produto gerado, contribui para a descrição da reengenharia sob a forma de padrões com o objetivo de torná-la mais clara para o engenheiro de software. Este artigo tem por objetivo descrever, na forma de sete grupos de padrões, o Processo de Reengenharia Orientada a Objetos (PRE/OO), que detalha a reengenharia de sistemas legados procedimentais para sistema orientados a objetos. Os grupos relacionados à qualidade do processo de reengenharia utilizam as áreas-chave para alcance do nível 2 de maturidade do SW-CMM como base para composição dos padrões.

Abstract

The reengineering patterns definition has been approached by several authors due to the need for more consistent guidelines in this process. In addition, the concern with the result, that is, the quality of the final product, contributes to the description of the reengineering process in the pattern format, in order to make it available to the software engineering community. The goal of this paper is to describe, in the form of seven pattern clusters, the Object-oriented Reengineering Process (PRE/OO), which details the reengineering of procedural legacy systems into object-oriented systems. The clusters concerned with the reengineering process quality use key process areas to reach SW-CMM level 2, which are the basis to compose the patterns.

1. Introdução

A reengenharia é a forma que muitas organizações estão buscando para manter/refazer seus softwares, livrando-se das manutenções difíceis e da degeneração de suas estruturas. Por este motivo, é importante que o resultado desse processo seja confiável. Desta forma, a garantia da qualidade também pode ser considerada como mais uma etapa da reengenharia.

O objetivo deste artigo é apresentar e documentar padrões para realizar a reengenharia de sistemas legados implementados em Delphi [1] de forma procedimental para sistemas orientados a objetos presentes na linguagem Object Pascal, utilizada no ambiente Delphi. Desta forma, foram elaborados vinte padrões que se encontram organizados em grupos relacionados às etapas de engenharia reversa e engenharia avante e, ainda, à qualidade de processo. Apesar dos padrões serem específicos para sistemas legados implementados em Delphi de forma procedimental, os mesmos podem ser utilizados, após algumas adaptações, em sistemas implementados em outras linguagens procedimentais.

2. Trabalhos relacionados aos padrões do PRE/OO

A partir do estudo de diretrizes, métodos, linguagens e famílias de padrões existentes para conduzir a reengenharia de sistemas procedimentais para sistemas orientados a objetos, e da verificação da necessidade de que a reengenharia seja conduzida levando em conta o aspecto qualidade, foram compostos os padrões do PRE/OO, influenciados pelos trabalhos:

- Fusion/RE [8, 9]: o método aplicado de forma seqüencial foi utilizado em diversos estudos de caso para a realização da segmentação (reengenharia orientada a objetos sem mudança de linguagem) em sistemas legados procedimentais. No PRE/OO foram adaptados os seis passos que o compõe:
  - Revitalização da arquitetura do software legado;
  - Recuperação do Modelo de Análise do Sistema Atual (MASA);
  - Criação do Modelo de Análise do Sistema (MAS);
  - Mapeamento do MAS para o MASA;
  - Elaboração do Projeto Avante;
  - Segmentação do Sistema.
- UML (*Unified Modeling Language*) [6]: os diagramas UML foram utilizados para documentação dos produtos elaborados durante o processo;
- Processo Evolutivo [10]: este modelo de processo foi utilizado por permitir ao engenheiro de software a repetição de passos anteriormente realizados do processo como forma de refinar os produtos gerados;
- FaPRE/OO [11,12]: padrões da Família de Padrões de Reengenharia foram adaptados para uso na reengenharia orientada a objetos de sistemas legados implementados em Delphi;
- SW-CMM (*Capability Maturity Model for Software*) [7]: as KPAs (key process areas) do Nível 2 de maturidade serviram como base para qualificação do processo de reengenharia. Propôs-se a realização de planejamento e acompanhamento para prover o mínimo de visibilidade com relação às necessidades do processo de reengenharia orientada a objetos de sistemas procedimentais. Essas práticas consideradas essenciais no processo de desenvolvimento de software não estavam explicitamente incluídas em processos de reengenharia, sendo uma das contribuições deste trabalho.

3. PRE/OO - Processo de Reengenharia Orientada a Objetos

O Processo de Reengenharia Orientada a Objetos foi composto sob a forma de cinco grupos de padrões divididos como descrito a seguir e ilustrados pela Figura 1:

- Grupos 1 e 2: implementam a melhoria da qualidade no processo de reengenharia baseados nas áreas-chave de processo (KPAs) para alcance do Nível 2 de maturidade do SW-CMM. O grupo 1 trata da preparação (baseada na KPA 1) e do planejamento da reengenharia (baseada na KPA2), sendo realizado antes do inicio do processo de reengenharia. O grupo 2 trata do acompanhamento da aplicação dos grupos 3 a 7 (com base nas KPAs 3, 4 e 5). Somente a KPA 6 – Gerenciamento de Sub-Contratados não foi utilizada por estar fora do escopo do processo;
- Grupos 3 a 5: os grupos do PRE/OO relacionados à engenharia reversa foram elaborados com base nos passos 1 a 3 do Fusion/RE, respectivamente. Porém, cada um desses passos pode ser realizado de forma evolutiva, como indicam as setas em torno das elipses na Figura 1 diferentemente do Fusion/RE, que é seqüencial linear. O conteúdo de alguns padrões foi influenciado, também, pela FaPRE/OO [11, 12];

- Grupos 6 e 7: correspondem à engenharia avante, elaborados com base na extensão do Fusion/RE [9] e no modelo de processo evolutivo [10].

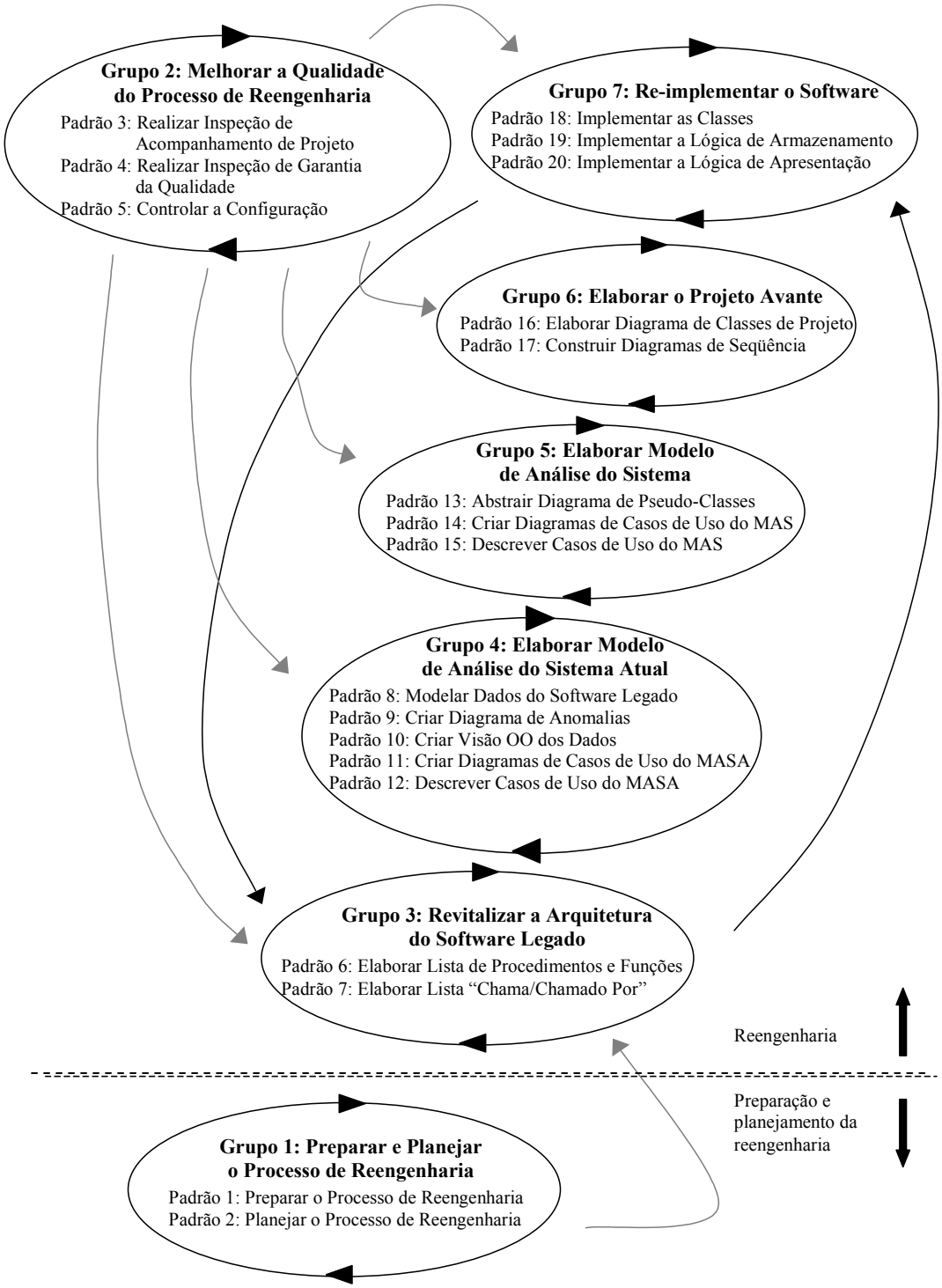


Figura 1. Estrutura de Grupos de Padrões do PRE/OO [5]

Cada padrão do PRE/OO é apresentado no seguinte formato: **Número, Nome, Intuito, Problema, Contexto, Solução, Exemplo, Usos Conhecidos, Padrões Relacionados** e

Produtos Obtidos, formato derivado de [2, 3, 4]. O item **Contexto** expressa também as Forças do padrão.

O item **Exemplo** não foi descrito pelo fato de ocupar muito espaço no artigo, tornando-o excessivamente longo. A descrição desse item e a aplicação completa num estudo de caso dos padrões do PRE/OO podem ser encontradas on-line em [5]. O item **Usos Conhecidos** não consta da estrutura de cada padrão por conter informações comuns a todos, sendo, portanto, descrito a seguir.

Usos Conhecidos: utilização em um sistema legado que realizava o controle de vendas e estoque implementado em Delphi, porém sem características orientadas a objetos [5]. Os padrões foram utilizados ainda em um trabalho de iniciação científica que consiste na reengenharia de um sistema de vídeo-locadora[1] e em trabalho de reengenharia de um sistema de biblioteca ministrado na disciplina de engenharia de software do curso de graduação em ciência da computação da UFSCar.

GRUPO 1 - Preparar e Planejar o Processo de Reengenharia.

1. Nome: Preparar o Processo de Reengenharia

Intuito:

Obter quais atividades serão realizadas durante o processo de reengenharia e quais produtos serão elaborados.

Problema:

Definir as atividades e os produtos de trabalho resultantes da aplicação do PRE/OO.

Contexto:

A lista com os produtos de trabalho que serão elaborados pode ser obtida analisando-se os recursos disponíveis. Já as atividades que devem ser seguidas para a obtenção desses produtos dependem diretamente do que precisa ser realizado e da análise do processo descrito nos padrões seguintes (PRE/OO).

Solução:

Elaborar o documento Levantamento de Atividades conforme modelo do Quadro 1.

Produto Obtido:

Documento Levantamento de Atividades.

Padrões Relacionados:

A aplicação desse padrão resulta na entrada para o padrão 2 (Planejar o Processo de Reengenharia).

Quadro 1. Modelo do documento Levantamento de Requisitos

LEVANTAMENTO DE ATIVIDADES	
Projeto *<<contém o nome do software legado que será submetido à reengenharia>>* **Versão do Documento** *<<consta do número adotado para diferenciar as versões do documento durante a evolução do processo de reengenharia e documentadas no controle de configuração, caso realizadas>>*	**Data de Criação do Documento** *dd/mm/aaaa* **Responsável pela Criação do Documento** *<<nome da pessoa responsável pela criação da versão correspondente ao documento>>*

[1].Relatório de IC- PIBIC/CNPq, abril 2003 - UFSCar. Alunos: Raquel G. Freitas, Rodrigo Murta e Hallen Fontana.

Exame da Situação *<<deve conter a contextualização do software em relação ao domínio em que está inserido, contendo a descrição detalhada de suas funcionalidades para uso posterior como fonte de auxílio ao seu entendimento>>*
Itens Disponíveis *<<lista que contém as fontes de informações disponíveis para utilização durante o processo de reengenharia. Em geral, consiste de: código-fonte, arquivos de dados e programa executável>>*
Produtos de Trabalho a Serem Elaborados *<<composto de produtos que contém resultados simples ou compostos desenvolvidos ao longo da reengenharia. O conteúdo desse campo varia conforme os Itens Disponíveis>>*
Atividades a Serem Realizadas *<< descrição de quais padrões são necessários à composição dos produtos de trabalho acima citados. Dependendo dos produtos já disponíveis, alguns padrões podem ser omitidos >>.*

2. Nome: Planejar o Processo de Reengenharia.

Intuito:

Planejar as atividades de reengenharia definidas com a aplicação do padrão Preparar Processo de Reengenharia.

Problema:

Estimar os tempos, recursos e itens de configuração relacionados ao projeto de reengenharia do software legado.

Contexto:

Esse padrão requer o planejamento das atividades, a partir do Documento Levantamento de Atividades obtido com a aplicação do padrão 1 (Preparar Processo de Reengenharia). Podem ser considerados os seguintes aspectos: é difícil estimar tempos para a realização do processo de reengenharia e dos passos nos quais esse se subdivide. Essa dificuldade pode ser minimizada com a adoção de métricas de tamanho e/ou com o auxílio da experiência do engenheiro de software.

Solução:

Elaborar o documento Plano para Realização da Reengenharia contendo as informações, referentes à aplicação de cada um dos grupos 3 a 7. A estrutura, que segue a forma ilustrada pelo Quadro 2.

Esse documento deve ser preenchido considerando-se cada um dos passos do processo de reengenharia (grupos 3 a 7). Assim, esse plano estará completo somente quando todos os passos forem estimados.

Produto Obtido:

Plano Para Realização da Reengenharia.

Padrões Relacionados:

Esse padrão deve ser aplicado, necessariamente, após o padrão 1, (Preparar o Processo Reengenharia), por sua saída ser utilizada aqui como entrada. Após o planejamento do processo, deve ser iniciado o processo de reengenharia efetivamente. Relaciona-se com o padrão 3, (Acompanhar o Progresso do Processo de Reengenharia – Grupo 2).

Quadro 2. Modelo de documento do Plano para Realização da Reengenharia

PLANO PARA REALIZAÇÃO DA REENGENHARIA	
Projeto **Versão do Documento**	**Data de Criação do Documento** **Responsável pela Criação do Documento**
Dados Gerais do Processo de Reengenharia **Tempo Disponível para o Processo** *<<tempo disponível para o processo de reengenharia, expresso em dias e horas>>* **Data Estimada para Finalização do Processo de Reengenharia:** *<<data obtida através do somatório dos tempos necessários para realização de cada passo do processo de reengenharia, estimados pelo engenheiro de software>>* **Itens de Configuração** *<<produtos de trabalho desenvolvidos durante o processo de reengenharia, para os quais é importante que seja realizado o controle de alterações>>*	
Inspeções de Garantia da Qualidade *<<lista quantas serão, no total, as inspeções realizadas nos produtos de trabalho>>* **Inspeções de Acompanhamento e Supervisão do Projeto** *<<lista quantas, quando e por quem serão realizadas as inspeções de acompanhamento de projeto. Sugere-se a realização de uma inspeção de acompanhamento de projeto ao final de cada passo do processo de reengenharia>>*	
Passo 1 – Revitalização da Arquitetura (grupo 3) **Produtos de Trabalho** *<<para cada passo do processo de reengenharia, devem ser especificados quais os produtos de trabalho devem elaborados, por quem, a estimativa de tempo e qual a ferramenta de auxílio será usada para sua elaboração, além de informações sobre a necessidade de controle de configuração>>* **Tempo Necessário para Conclusão do Passo** *<<é o tempo obtido a partir do somatório das estimativas de tempo para a elaboração de cada um dos produtos de trabalho produzidos nesse determinado passo>>*	
Número de Inspeções Inspeções de Garantia da Qualidade:	Inspeções de Acompanhamento e Supervisão:
Passo 2 – Recuperação do Modelo de Análise do Sistema Atual (grupo 4) ...	
Passo 3 – Obtenção do Modelo de Análise do Sistema (grupo 5) ...	
Passo 4 – Elaboração do Projeto Avante (grupo 6) ...	
Passo 5 – Re-implementação do Software (grupo 7) ...	
Totalizações ...	

GRUPO 2 – Melhorar a Qualidade do Processo de Reengenharia

3. Nome: Acompanhar o Progresso do Processo de Reengenharia

Intuito:

Manter sempre atualizado o plano do processo de reengenharia.

Problema:

Contornar os possíveis desvios que o projeto de reengenharia do software legado possa vir a sofrer a partir das estimativas realizadas durante o seu planejamento.

Contexto:

A inspeção de acompanhamento do processo é a verificação do progresso do projeto de reengenharia de um software legado com relação ao que foi planejado. Caso, durante a inspeção de acompanhamento do processo, conclua-se que o desenvolvimento do processo esteja defasado em relação ao que foi planejado, ações corretivas devem ser tomadas. Tais ações incluem a correção do Plano para Realização da Reengenharia, de modo que esse reflita a execução real, ou o replanejamento dos passos restantes. Esse padrão é aplicado com base no Plano para Realização da Reengenharia elaborado no padrão Planejar o Processo de Reengenharia e no andamento real do processo.

Pode-se encontrar dificuldades em definir o que deve ser considerado como desvio no Plano, visto o aspecto subjetivo a ser definido pelo engenheiro de software e, ainda, qual atitude deve ser tomada em caso de verificação de um desvio.

Solução:

3.1) Elaborar um documento denominado "Inspeção de Acompanhamento do Processo" contendo as informações:

a) Cabeçalho: preenchido como nos documentos elaborados nos padrões anteriores;
b) Responsável pela Inspeção de Acompanhamento: nome da pessoa ou dos integrantes da equipe que realizou a inspeção de acompanhamento e supervisão de projeto;
c) Passo do PRE/OO: descreve em qual passo do processo de reengenharia que foi realizada a revisão de acompanhamento de projeto, ou seja, durante a aplicação de qual grupo de padrões. Por exemplo: Revitalização da Arquitetura do Software Legado, MASA, Projeto Avante, etc.
d) Número de Inspeção: contém o número seqüencial para identificação das inspeções de acompanhamento de projeto;
e) Itens Analisados: descreve os aspectos do Plano para Realização da Reengenharia que foram submetidos à inspeção. Podem ser divididos em sub-seções em que são analisados diferentes aspectos relacionados ao processo;
f) Resultados Obtidos: deve constar a comparação entre as estimativas e os resultados reais alcançados em termos de tempo gasto e produtos de trabalho elaborados;
g) Ações Corretivas Necessárias: descreve quais as ações corretivas foram utilizadas para correção ou adaptação do Plano para Realização da Reengenharia, no caso de desvio em relação ao planejado.

3.2) Para cada um dos passos do processo de reengenharia (grupos 3 a 7) deve-se realizar e documentar a inspeção de acompanhamento do progresso do projeto, bem como os ajuste feitos no Plano para Realização da Reengenharia, caso necessários.

3.3) Caso seja necessária a criação de uma nova versão do Plano para Realização da Reengenharia, deve-se realizar o controle de configuração.

Produto Obtido:

Documento de Inspeção do Acompanhamento do Processo.

Padrões Relacionados:

Esse padrão deve ser aplicado durante processo de reengenharia, preferencialmente após o fim de cada um dos passos (grupos 3 a 7) do PRE/OO.

4. Nome: Realizar Inspeção de Garantia da Qualidade

Intuito:

Garantir a qualidade dos produtos gerados durante o processo de reengenharia.

Problema:

Descobrir os erros em produtos de trabalho em relação ao que foi planejado, às especificações e/ou padrões propostos.

Contexto:

Cada produto de trabalho gerado ao longo do processo de reengenharia deve ser alvo da inspeção proposta com esse padrão. O engenheiro de software pode decidir sobre a realização das inspeções apenas em produtos críticos, de acordo com o porte do software submetido ao processo de reengenharia orientada a objetos.

Solução:

Elaborar um documento contendo as seguintes informações:

a) Cabeçalho: preenchido como nos documentos elaborados nos padrões anteriores;
b) Item Alvo da Inspeção de Garantia da Qualidade: contém o nome e a versão do produto de trabalho inspecionado;
c) Responsável pela Inspeção de Garantia da Qualidade: contém o nome da pessoa ou dos integrantes da equipe responsável pela realização da inspeção;
d) Número de Inspeção: contém um número seqüencial para identificação de cada inspeção de garantia da qualidade;
e) Aspectos Analisados: descreve todos os aspectos analisados no produto de trabalho, a forma de análise e a situação em relação ao desejado;
f) Diferenças Encontradas: relata as diferenças entre os aspectos analisados e os resultados esperados;
g) Ações Corretivas Necessárias: descreve a proposta para corrigir as diferenças encontradas.

Esse padrão é aplicado de forma a garantir a qualidade dos produtos de trabalho gerados durante o processo de reengenharia, verificando se tais produtos refletem o que foi solicitado e se seguem os padrões propostos.

Produto Obtido:

Documento de Inspeção de Garantia da Qualidade.

Padrões Relacionados:

Relaciona-se com os grupos 3 a 7 do PRE/OO, de forma a inspecionar cada saída produzida a partir da aplicação de seus padrões.

5. Nome: Controlar a Configuração

Intuito:

Estabelecer e manter a integridade dos produtos de trabalho elaborados ao longo do processo de reengenharia.

Problema:

Controlar as alterações realizadas nos vários produtos de trabalho de forma a não permitir a ocorrência de inconsistências.

Contexto:

A condução do processo de reengenharia de forma evolutiva torna indispensável a implementação desse padrão, como forma de controlar mudanças e versões dos produtos de trabalho construídos, fator que torna-se ainda mais crítico no caso do processo ser conduzido por mais de uma pessoa. Podem ser considerados os seguintes aspectos:

- A aplicação do padrão pode ser dificultada por não ser realizada de forma automatizada;
- O uso de ferramentas próprias para o Controle de Configuração e a criação de *Baselines* pode ser uma opção a aplicação desse padrão.

Solução:

5.1) Definir os itens de configuração, ou seja, os produtos de trabalho que terão suas versões controladas;

5.2) Documentar todas as alterações realizadas nos itens de configuração, descrevendo sua origem, quando e por quem foi feita a alteração (no caso de mais de uma pessoa estar associada ao processo) e em que versão do item resultou, com as informações:

a) Cabeçalho: preenchido como nos documentos elaborados nos padrões anteriores;
b) Responsável pelo Controle de Configuração e Passo do PRE/OO;
c) Data de Criação: dd/mm/aaaa;
d) Nome do Documento: refere-se ao nome dado ao produto de trabalho que é alvo do controle de configuração;
e) Versão: contém a versão do item de configuração criada após a realização das alterações;
f) Origem: especifica a partir de qual processo ou inspeção foi originada a versão desse item de configuração.

5.3) Estabelecer uma *baseline* ao final de cada passo do processo de reengenharia para que se possa ter um maior controle do projeto em termos de gerenciamento de configuração, ou seja, um conjunto de produtos de trabalho inspecionados que servem de base para a continuação do projeto e que só podem ser alterados mediante controle documentado. A documentação para o gerenciamento das *baselines* deve conter os seguintes itens:

a) Cabeçalho: preenchido como nos documentos elaborados nos padrões anteriores;
b) Responsável pelo Controle de Configuração e Passo do PRE/OO;
c) Data de Criação da *Baseline*: dd/mm/aaaa;
d) Descrição: contém a explicação sobre o conteúdo da *baseline* criada;
e) Itens de Configuração: lista o nome de cada item de configuração que compõe a *baseline*;
f) Versão: lista a versão do item de configuração que compõe a *baseline*;
g) Data de Criação: relata a data da criação da versão descrita do item de configuração.

Produtos Obtidos:

Lista de Controle de Configuração e Documento com as *Baselines* do Projeto.

Padrões Relacionados:

Relaciona-se com todos os padrões dos grupos 3 a 7, de forma a controlar as alterações em todos os itens de configuração gerados a partir de sua aplicação.

GRUPO 3 – Revitalizar a Arquitetura do Software Legado

6. Nome: Elaborar Lista de Procedimentos e Funções

Intuito:

Identificar todos os procedimentos e funções, definidos pelo programador, que serão transformados pelo processo de reengenharia.

Problema:

Destacar, a partir do código-fonte, somente os procedimentos e funções definidos pelo programador, ignorando os procedimentos e funções declaradas em bibliotecas ou APIs (*Application Programmable Interfaces*) fornecidas pelo fabricante do ambiente de desenvolvimento *Delphi*.

Contexto:

Consideram-se apenas os procedimentos e as funções definidos pelo programador para realizar a engenharia reversa. A base para a análise dos procedimentos e funções é o código-fonte todas as *units* do software legado;

No *Delphi*, grande parte do código-fonte é executado direta ou indiretamente em resposta a eventos. Um evento é um tipo especial de propriedade que representa uma ocorrência em tempo de execução, geralmente uma ação do usuário (como o *click* do mouse sobre um botão). Podem ser considerados os seguintes aspectos:

- Podem-se localizar os eventos clicando-se sobre a aba *Events* do *Object Inspector*.
- A ilegibilidade dos nomes utilizados para descrever os procedimentos e funções pode dificultar o processo;
- A aplicação do padrão é facilitada pelo fato dos protótipos dos procedimentos e funções serem descritos na seção Interface das *units* (arquivos-fonte do *Delphi* com extensão ".PAS").

Solução:

6.1) Identificar o protótipo (cabeçalho) de todos os procedimentos e funções presentes no código das units do software legado;

6.2) Classificar cada procedimento ou função de acordo com os critérios a seguir:

- Ev – Evento: procedimentos originados em resposta à eventos disparados pelo sistema. Encontram-se declarados acima da palavra reservada *public* na interface da *unit*,
- Pr – *Private*: procedimentos ou funções visíveis apenas internamente a *unit* em que estão declarados. Encontram-se declarados abaixo da palavra reservada *private* na interface da *unit*,
- Pb – *Public*: procedimentos ou funções visíveis a outras *units*, além daquela em que se encontram declarados. Encontram-se declarados abaixo da palavra reservada *public* na interface da *unit*.

6.3) Documentar os passos acima, de acordo com os itens:

a) Cabeçalho: como nos documentos elaborados nos padrões anteriores;
b) Módulos do Software: deve conter todos os arquivos com extensão ".pas" que compõem o software legado;
c) Nomes dos Procedimentos e Funções: lista todos os procedimentos e funções que são extraídos a partir do código-fonte;
d) Classificação: lista a visibilidade do procedimento ou função em relação ao módulo em que está declarado e ao software. Utiliza a nomenclatura Ev, Pb ou Pv conforme descrito no Passo 2 da solução.

Obs.: Os três últimos campos acima descritos devem ser repetidos até que se esgotem todos os procedimentos e funções que compõem um módulo e todos os módulos que compõem o software legado.

Produto Obtido:

Lista de Procedimentos e Funções.

Padrões Relacionados:

Esse padrão inicia o processo de reengenharia, sendo aplicado após a aplicação do grupo 1. Após a aplicação deste padrão, deve-se:

- realizar a inspeção de garantia da qualidade da Lista de Procedimentos e Funções: utilizar o padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- realizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia reversa: padrão 7 (Elaborar Lista "Chama/Chamado Por").

7. Nome: Elaborar Lista de "Chama/Chamado Por"

Intuito:

Obter entendimento da funcionalidade implementada em cada procedimento e função do software legado.

Problema:

Extrair, de cada procedimento e função presente na Lista de Procedimentos e Funções, a funcionalidade, os procedimentos e as funções que são por ele chamados e por quais outros procedimentos e funções ele é chamado.

Contexto:

O código-fonte do software legado contém muitas regras de negócio e detalhes de implementação importantes para o entendimento da funcionalidade que podem passar despercebidos pelo engenheiro de software que está conduzindo a engenharia reversa.

- Um ponto negativo que o engenheiro de software pode encontrar para a aplicação desse padrão é o tempo que este consome;
- Por se tratar do estudo e documentação do código, que é um trabalho minucioso, o engenheiro de software pode optar por não aplicá-lo ou aplicá-lo apenas nas *units* mais importantes do software legado;
- Após a aplicação desse padrão, o entendimento acerca do software aumenta consideravelmente.

Solução:

Preencher a Lista “Chama/Chamado Por” para cada procedimento e função que conste da Lista de Procedimentos e Funções, com as seguintes seções:

a) Cabeçalho: como nos documentos elaborados nos padrões anteriores;
b) Módulo do Software: contém o arquivo com extensão ".PAS", o qual teve seu código-fonte utilizado na composição da lista. Elabora-se uma Lista "Chama/Chamado Por" para os arquivos com extensão ".PAS" que compõe o software;
c) Procedimento/Função: contém o nome do procedimento ou função com descrição sucinta de sua funcionalidade, que é obtida a partir do entendimento do código implementado no corpo do procedimento ou função, encontrado na seção *implementation* da *unit* em que esse se encontra declarado;
d) Chama: lista os procedimentos chamados por outros procedimentos e funções da Lista de Procedimentos e Funções, existentes no código do procedimento ou função analisado;
e) Chamado Por: lista os procedimentos ou funções que são chamados pelo procedimento listado no item Chama. De acordo com a classificação, podem ocorrer três situações:
    - Caso o procedimento ou função apresente a classificação Pb (*Public*), procedimentos e funções de outras *units* podem chamá-lo (considerar apenas aqueles que constem da Lista de Procedimentos e Funções), sendo a coluna “Chamada Por” completada somente quando todos os procedimentos ou funções foram classificados, ou seja, quando o processo de revitalização das demais *units* for concluído;
    - Caso o procedimento ou função apresente a classificação Pv (*Private*), apenas procedimentos e funções da *unit* em que esse se encontra podem chamá-lo (considerar apenas aquelas que constam da Lista de Procedimentos e Funções),

portanto a coluna "Chamado Por" será completada ao término da revitalização da *unit* em que esse se encontra;
- Caso o procedimento tenha a classificação Ev, a coluna "Chamado Por" será vazia, pelo fato de se tratar de um evento, o qual somente é disparado por algum agente externo ao software e nunca por outros procedimentos/funções.

Produto Obtido:

Lista "Chama/Chamado Por".

Padrões Relacionados:

Esse padrão somente pode ser aplicado após ter sido aplicado o padrão 6 (Elaborar Lista de Procedimentos e Funções), pois a saída dele é utilizada como entrada para este padrão. Após a aplicação deste padrão, deve-se:
- realizar a inspeção de garantia da qualidade da Lista "Chama/Chamado Por": padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- inspecionar o andamento do processo em relação ao Plano para Realização da Reengenharia: padrão 3 (Acompanhar o Progresso do Processo de Reengenharia);
- continuar a engenharia reversa: padrão 8 (Modelar Dados do Software Legado).

GRUPO 4 – Elaborar Modelo de Análise do Sistema Atual

8. Nome: Modelar Dados do Software Legado

Intuito:

Construir Modelo Entidade Relacionamento (MER) correspondente à implementação atual dos dados do software legado a partir dos dados contidos no banco de dados relacional.

Problema:

Interpretar e abstrair os dados das tabelas software legado: nome de cada tabela, nomes dos campos de dados e tipo de cada campo. Essas informações geralmente encontram-se documentados sob a forma de *scripts* SQL (*Structured Query Language*) presentes nos arquivos de dados do software legado.

Contexto:

Caso os dados estejam documentados sob a forma de *scripts* SQL, é necessário que o engenheiro de software possua conhecimentos básicos sobre SQL;

Os dados podem, ainda, estar presentes num documento ou serem obtidas analisando-se a base de dados;

Como o banco de dados relacional não consegue representar o relacionamento entre as tabelas, é necessário analisar seus campos para obter tais informações;

As cardinalidades entre as entidades do MER nem sempre são triviais e, muitas vezes, só são obtidas a partir de investigação das regras de negócios embutidas no código ou percebidas através da execução do software legado.

Solução:

8.1) Montar, a partir dos arquivos de dados do software legado escritos sob a forma de *scripts* SQL, um documento com as seguintes seções:
   a) Cabeçalho: preenchido como nos padrões anteriores;
   b) Tabelas de Dados: contém as informações obtidas com a realização do passo 2 da Solução;
   c) Campos de Dados: contém as informações obtidas realizando-se o passo 3 da Solução;

d) Chave Primária: contém o(s) campo(s) que identifica(m) unicamente cada registro da Tabela de Dados, conforme instruções do passo 4 da Solução;
e) Chaves Estrangeiras: campo(s) da Tabela de Dados que compõe(m) a Chave Primária de outra(s) Tabela(s) de Dados, o que pode ser verificado através da análise do campo Chave Primária da Lista de Tabelas e Chaves;

8.2) Preencher a coluna Tabela de Dados a partir da identificação do nome de cada tabela de dados associado às cláusulas CREATE TABLE dos *scripts*;

8.3) Identificar, abaixo de cada cláusula CREATE TABLE os nomes e tipos dos campos relacionados às tabelas de dados identificadas no passo anterior. Esses campos deverão ser inseridos na coluna Campos de Dados;

8.4) Identificar as cláusulas PRIMARY KEY nos *scripts* SQL dos arquivos de dados. Cada cláusula desse tipo corresponde à chave primária da tabela de dados identificada no Passo 2 da solução. Cada chave primária identificada deve ter seu nome inserido na coluna Chave Primária;

8.5) Identificar as chaves estrangeiras, ou seja, as Chaves Primárias de Tabelas de Dados declaradas como campos em outras tabelas, e inseri-las na coluna Chaves Estrangeiras.

8.6) Construir o MER. A partir das informações da Lista de Tabelas e Chaves. Cada Tabela de Dados dá origem a uma entidade do MER, cada Chave Estrangeira identifica os relacionamentos entre as diversas Tabelas de Dados representadas. As cardinalidades entre as entidades devem ser obtidas através das regras de negócios, extraídas do código-fonte, com a execução do software legado e/ou de entrevistas com usuários.

Produtos Obtidos:

Lista de Tabelas e Chaves, Modelo Entidade Relacionamento.

Padrões Relacionados:

Após a aplicação do padrão, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade do MER: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- prosseguir com a engenharia reversa: padrão 9 (Criar Lista de Anomalias).

9. Nome: Criar Lista de Anomalias

Intuito:

Obter, a partir do código-fonte, todas as anomalias relacionadas aos procedimentos que acessam as tabelas de dados identificadas.

Problema:

Registrar e classificar os procedimentos e funções que observam e/ou consultam mais de uma entidade.

Contexto:

Para a aplicação do padrão, todo o código-fonte deve ser percorrido;

Permite a posterior divisão dos procedimentos e funções de forma a eliminar as anomalias e permitir a migração para o paradigma orientado a objetos.

Solução:

Criar uma lista a partir dos procedimentos e funções da Lista de Procedimentos e Funções e a classificação de cada um quanto as anomalias em relação às tabelas de dados, obedecendo aos critérios apresentados na Tabela 1.

A Lista de Anomalias deve conter as seguintes seções:

a) Cabeçalho: preenchido conforme padrões anteriores;
b) Módulos do Software: contém cada arquivo com extensão ".pas" que compõe o software legado;
c) Procedimento ou Função: deve conter todos os procedimentos e as funções que compõem a Lista de Procedimentos e Funções;
d) Tabelas de Dados: descreve os nomes das tabelas de dados observadas e/ou alteradas dentro do procedimento ou função declarado na coluna anterior;
e) Critério de Acesso à(s) Tabela(s): classificação de acordo com o tipo de acesso que o procedimento ou função faz a cada Tabela de Dados;
f) Classificação da anomalia: caso o procedimento/função seja anômalo, ou seja, altere e/ou observe mais de uma Tabela de Dados simultaneamente, o tipo final de sua anomalia deve ser transcrito, obedecendo os critérios da Tabela 1.

Tabela 1. Critérios para classificações das anomalias em procedimentos e funções

Símbolo	Significado	Critério
o	Observador	Procedimento ou função que observa a tabela de dados
c	Construtor	Procedimento ou função que altera a tabela de dados
i	de Implementação	Procedimento ou função relacionado(a) à implementação
+	Mais de uma tabela de dados observada ou alterada	Associado aos símbolos (o) ou (c), indica que o procedimento/função observa e/ou altera 2 ou mais tabelas.

Produto Obtido:

Lista de Anomalias.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação dos padrões 6 e 8 (Elaborar Lista de Procedimentos e Funções e Modelar Dados do Software Legado), pelo fato das saídas destes serem utilizadas aqui como entradas. De forma a prosseguir a reengenharia, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade da Lista de Anomalias: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar o processo de engenharia reversa: padrão 10 (Criar Visão OO dos Dados).

10. Nome: Criar Visão OO dos Dados

Intuito:

Obter a visão orientada a objetos dos dados a partir do sistema legado procedimental.

Problema:

Transformar o modelo de dados construído de forma procedimental em uma visão orientada a objetos.

Contexto:

Pode-se utilizar ferramentas, como o *Rational Rose*, para modelar o diagrama elaborado com a aplicação desse padrão. Além disso, o engenheiro de software tem conhecimento dos conceitos da UML, para gerar modelos orientados a objetos a partir do MER.

Solução:

10.1) Considerar cada entidade do MER como uma pseudo-classe (estrutura de dados que pode vir a representar uma classe na implementação orientada a objetos do sistema);

10.2) Transportar os Campos de Dados descritos na Lista de Tabelas e Chaves para o Diagrama de Pseudo-Classes, que está sendo gerado, como atributos das pseudo-classes consideradas;

10.3) Analisar a Lista de Anomalias e, cada procedimento e função que tenha sido classificado como oc, c+, o+, o+c+, oc+ ou o+c, deve ser considerado como método, no Diagrama de Pseudo-Classes, de todas as pseudo-classes com as quais se relaciona (as quais modifica e/ou consulta);

10.4) Verificar as cardinalidades entre as entidades do Modelo Entidade-Relacionamento:

a) Para relacionamentos binários (entre duas entidades) e com cardinalidade igual a 0..1, 1..1, 0..N ou 1..N, deve-se transpor a cardinalidade para o Diagrama de Pseudo-Classes da mesma forma que esta se encontra no MER. Deve-se verificar também, o tipo do relacionamento, de forma a representá-lo como associação ou agregação (todo-parte), de acordo com a funcionalidade que representa. Os relacionamentos de agregação podem ser percebidos nas situações em que uma pseudo-classe faz o papel, dentro do contexto do software, de parte de outra pseudo-classe. Os relacionamentos que não forem dessa forma, devem ser mapeados como associação;

b) Para relacionamentos ternários (entre três entidades) e com cardinalidade N..N, deve-se verificar, no MER, se o resultado desse relacionamento pode ser representado como um link-atributo das pseudo-classes envolvidas.

Produto Obtido:

Diagrama de Pseudo-Classes.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação dos padrões 8 e 9 (Modelar Dados do Software Legado e Elaborar Lista de Anomalias), pelo fato que as saídas deles são utilizadas como entradas para sua aplicação. Continuando o processo de engenharia reversa, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no Diagrama de Pseudo-Classes: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia reversa: padrão 11 (Criar Diagramas de Casos de Uso do MASA).

11. Nome: Criar Diagramas de Casos de Uso do MASA

Intuito:

Modelar as funcionalidades e o comportamento do software legado.

Problema:

Mapear o comportamento do software legado, de forma a completar o processo de engenharia reversa.

Solução:

Modelar os agentes externos que interagem com o sistema (atores) e os eventos (casos de uso) promovidos por eles:

11.1) Verificar, através de cada interface com o usuário, presente no software legado, quem dispara as interações com o ambiente externo. Essas interações podem ocorrer por meio de botões, menus, caixas de texto ou grades (*grids*). O responsável pelo disparo do evento deve ser considerado como ator do caso de uso;

11.2) Verificar quais as interações ocorrem em cada interface. Cada interação origina um caso de uso;

11.3) Nomear o caso de uso:

a) Caso o evento que origina o caso de uso seja tratado pelo ambiente *Delphi* através de componentes, não há código-fonte mapeado com o conteúdo do caso de uso. Nesse caso, o nome do caso de uso deve refletir sua funcionalidade;
b) Caso o evento que origina o caso de uso seja tratado pelo programa, deve-se obter, através do código-fonte, o nome do procedimento que trata o evento descrito, o qual deve constar da Lista de Procedimentos e Funções. O nome do caso de uso deve ser o nome do procedimento.

Produto Obtido:

Diagramas de Casos de Uso do MASA.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação do padrão 6 (Elaborar Lista Procedimentos e Funções), pelo fato de sua saída ser utilizada aqui como entrada. Após a aplicação desse padrão, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no Diagrama de Casos de Uso do MASA: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia reversa: padrão 12 (Descrever Casos de Uso do MASA).

12. Nome: Descrever Casos de Uso do MASA

Intuito:

Detalhar a documentação dos casos de uso do software legado.

Problema:

Documentar o curso normal e os cursos alternativos de execução dos casos de uso.

Contexto:

A elaboração das descrições de cada caso de uso pode variar em função de fatores específicos a cada software legado, como o conhecimento do engenheiro de software a respeito do domínio da aplicação, sua experiência em *Delphi* e o tempo disponível para a aplicação do padrão;

Apenas os casos de uso tratados em procedimentos implementados pelo programador devem ter suas descrições elaboradas. Os casos de uso tratados por componentes automáticos do *Delphi* devem ser desconsiderados.

Solução:

O código-fonte das *units* (arquivos-fonte do *Delphi* com extensão ".PAS") e os Diagramas de Casos de Uso formam a base para a elaboração das descrições dos casos de uso. Para cada caso de uso deve ser obtida a descrição correspondente, seguindo a forma:

12.1) Estudar e documentar o código do procedimento responsável pela execução do caso de uso, extraindo a seqüência das operações;
12.2) Verificar e documentar todas as ocorrências de cursos alternativos no código, com o objetivo de mapear todas as ramificações decorrentes do curso normal.

Produto Obtido:

Descrições dos Casos de Uso do MASA.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação do padrão 11 (Criar Diagramas de Casos de Uso do MASA), pelo fato da saída deste ser utilizada aqui como entrada. Continuando o processo de reengenharia, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade nas Descrições dos Casos de Uso do MASA: padrão 4 (Realizar Inspeção de Garantia da Qualidade);

- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- inspecionar o andamento do processo em relação ao Plano para Realização da Reengenharia: padrão 3 (Acompanhar o Progresso do Processo de Reengenharia);
- continuar a engenharia reversa: padrão 13 (Abstrair Diagrama de Pseudo-Classes).

GRUPO 5 – Elaborar Modelo de Análise do Sistema.

13. Nome: Abstrair Diagrama de Pseudo-Classes
Intuito:
Criar a visão orientada a objetos do software legado.
Problema:
Eliminar as anomalias presentes no Diagrama de Pseudo-Classes elaborado no MASA.
Contexto:
A análise dos relacionamentos modelados é, em muitos casos, subjetiva. Portanto, o conhecimento do engenheiro de software sobre orientação a objetos é indispensável;
No MAS, os procedimentos e funções anômalos do MASA são mapeados num número variável de métodos associados às classes modificadas e/ou consultadas, cabendo ao engenheiro de software a decisão sobre a modularização das funcionalidades implícitas a esses procedimentos e funções.
Solução:
13.1) Alterar os nomes das classes, caso necessário, para mnemônicos mais significativos;
13.2) Verificar a representatividade dos nomes de cada atributo das classes e sua utilidade no software legado;
13.3) Mapear as alterações realizadas nos passos 13.1 e 13.2 elaborando um documento com as seguintes seções:
a) Cabeçalho: preenchido como citado nos padrões anteriores;
b) Coluna MASA (Pseudo-Classes): contém a documentação dos itens presentes no código legado, antes da aplicação do padrão:
- Nome: nome da pseudo-classe,
- Atributos Modificados: nomes dos atributos antes de sua alteração no MAS,
- Métodos sem Anomalias: nomes dos métodos classificados como (o) ou (c) antes de sua alteração no MAS;

c) Coluna MAS (Classes): contém os nomes das modificações e exclusões realizadas nas classes, atributos e métodos sem anomalias. Nessa coluna são preenchidos:
- Nome: nome da classe,
- Atributos Modificados: nomes dos atributos após sua alteração no MAS,
- Métodos sem Anomalias: nomes dos métodos classificados como (o) ou (c) após sua alteração no MAS,
- Novos Métodos: métodos criados no MAS, a partir da divisão da funcionalidade de métodos sem anomalias, para a melhoria da modularização e aumento do reuso do código.

13.4) Verificar os nomes dos procedimentos e funções sem anomalias, ou seja, classificados como (o) ou (c). Esses procedimentos e funções devem ter o nome verificado quanto à representatividade e, ainda, o código-fonte analisado quanto a possibilidade de serem "quebrados" em um ou mais métodos, de forma a facilitar o reuso;

13.5) Mapear as alterações realizadas nos métodos sem anomalias na lista parcialmente preenchida no passo 13.3;

13.6) Transformar os métodos anômalos em quantos métodos forem necessários de forma a dividi-los nas classes que observam/constróem. Por exemplo, um procedimento (oc) declarado nas classes A e B deve originar pelo menos dois métodos, um observador na classe A e um construtor na classe B. Outros métodos podem surgir, com o objetivo de reutilizar código e torná-lo mais estruturado, ou ainda, pode-se fazer uso dos métodos já existentes (criados a partir da aplicação do Passo 13.4), caso tenham a mesma funcionalidade;

13.7) Documentar as transformações realizadas nos métodos anômalos, conforme as seções:

a) Cabeçalho: preenchido como citado nos padrões anteriores;
b) Módulo do Software: lista a *unit* do software legado que contém o método anômalo;
c) Método Anômalo: contém o nome do método com anomalia;
d) Classif.: descreve a classificação de acordo com o tipo da anomalia do método: oc, c+, o+, o+c+, oc+ ou o+c;
e) Classes: contém o nome das classes a que serão associados os métodos após a eliminação das anomalias;
f) Métodos: nomes dos métodos resultantes da eliminação das anomalias

13.9) Identificar as funcionalidades do software legado tratadas por meio de funções e procedimentos de componentes *Delphi* de acesso direto às Tabelas de Dados;

13.10) Transformar essas funcionalidades em quantos métodos forem necessários para prover seu encapsulamento nas classes definidas. Esses novos métodos deverão constar apenas da coluna MAS da Lista elaborada a partir do passo 13.3;

13.11) Verificar e, caso necessário, substituir os tipos de relacionamentos entre as classes (associação, agregação e herança). Para isso, identifica-se a representatividade de cada relacionamento junto ao domínio da aplicação e com relação às regras da orientação a objetos nos relacionamentos existentes;

13.12) Alterar os nomes dos relacionamentos, caso necessário, com o objetivo de melhorar a representatividade dos mesmos.

Produtos Obtidos:

Diagrama de Classes, Lista de Mapeamento MASA x MAS, Lista de Correspondência de Métodos Anômalos.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação do padrão 10 (Criar Visão OO dos Dados), pelo fato de sua saída ser utilizada aqui como entrada. Continuando o processo de engenharia reversa, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no Diagrama de Classes: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia reversa: padrão 14 (Criar Diagramas de Casos de Uso do MAS).

14. Nome: Criar Diagramas de Casos de Uso do MAS

Intuito:

Completar visão orientada a objetos do software legado.

Problema:

Refletir, no Diagrama de Casos de Uso do MASA, as modificações feitas no Diagrama de Classes, com a eliminação das anomalias, a reestruturação dos métodos existentes e a criação de novos métodos.

Contexto:

O engenheiro de software deve analisar cada caso de uso para definir a seqüência da realização das operações sem alterar a funcionalidade representada, mas de forma a refletir os métodos criados no MAS;

Não há aumento da abstração, apenas reestruturação em função das mudanças MASA x MAS realizadas nos diagramas já elaborados;

O engenheiro de software deve verificar a necessidade da criação de novos casos de uso a partir da divisão da funcionalidade dos métodos anômalos.

Solução:

14.1) Substituir, nos Diagramas de Casos de Uso do MASA, os nomes dos casos de uso pelos nomes que os métodos sem anomalias passaram a utilizar no MAS. Esses nomes são visualizados na Lista de Mapeamento MASA x MAS;

14.2) Substituir, nos Diagramas de Casos de Uso do MASA, os nomes dos casos de uso pelos nomes que os métodos criados a partir da eliminação das anomalias passaram a utilizar no MAS. Esses nomes são visualizados na Lista de Correspondência de Métodos Anômalos;

14.3) Re-especificar os Diagramas de Casos de Uso do MASA, mantendo os atores e atualizando os fluxos de entrada e saída dos casos de uso para que reflitam as alterações necessárias;

14.4) Elaborar a Lista de Mapeamento dos Casos de Uso para documentar as modificações efetuadas nos Diagramas de Casos de Uso do MASA. A lista deve conter as seções:

a) Projeto, Versão do Documento, Data da Criação do Documento e Responsável pela Criação do Documento: preenchidos como nos padrões anteriores;
b) Casos de Uso (MASA): contém o nome dos casos de uso descritos nos Diagramas de Casos de Uso elaborados no MASA;
c) Casos de Uso (MAS): contém o nome dos casos de uso após a abstração do MAS;
d) Classe: lista a classe a que pertence o método que trata o caso de uso.

Produtos Obtidos:

Diagramas de Casos de Uso do MAS e Lista de Mapeamento dos Casos de Uso.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação dos padrões 11 e 13 (Criar Diagrama Casos de Uso do MASA e Abstrair Diagrama de Pseudo-Classes), pois suas saídas são utilizadas aqui como entradas. Continuando o processo de engenharia reversa, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no Diagrama de Casos de Uso do MAS: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia reversa: padrão 15 (Descrever Casos de Uso do MAS).

15. Nome: Descrever Casos de Uso do MAS

Intuito:

Re-especificar as descrições dos casos de uso alterados durante a abstração dos modelos elaborados.

Problema:

Atualizar as descrições dos casos de uso modificados após a abstração do Diagramas de Casos de Uso do MASA.

Solução:

15.1) Obter, na Lista de Mapeamento de Casos de Uso, os nomes dos métodos que originam os casos de uso do MAS e que tenham sofrido modificações, além da troca do nome para melhoria da representatividade;

15.2) Re-especificar as descrições dos casos de uso, para cada método encontrado no passo anterior, de forma que reflitam as modificações necessárias com relação à funcionalidade e acesso a classes;

15.3) Descrever os casos de uso que, no MASA, eram tratados por componentes do ambiente Delphi e que no MAS passaram a ser tratados por métodos das classes.

Produto Obtido:

Descrições dos Casos de Uso do MAS.

Padrões Relacionados:

Esse padrão somente pode ser aplicado após a aplicação dos padrões 11 e 13 (Criar Diagrama Casos de Uso do MASA e Abstrair Diagrama de Pseudo-Classes), pelo fato de suas saídas serem utilizadas aqui como entradas. Continuando o processo de reengenharia, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade nas Descrições dos Casos de Uso do MAS: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- inspecionar o andamento do processo em relação ao Plano para Realização da Reengenharia: padrão 3 (Acompanhar o Progresso do Processo de Reengenharia);
- iniciar o processo de engenharia avante: padrão 16 (Elaborar Diagrama de Classes de Projeto).

GRUPO 6 – Elaborar o Projeto Avante.

16. Nome: Elaborar Diagrama de Classes de Projeto

Intuito:

Modularizar a funcionalidade do software, de forma a separá-la do acesso aos dados.

Problema:

Criar, uma nova instância do Diagrama de Classes, com menor nível de abstração, para visualizar as modularizações necessárias à separação da lógica de negócios da lógica de armazenamento do software.

Contexto:

Separar o acesso a dados dos aspectos funcionais e da interface do software é difícil para um programador sem experiência em programação orientada a objetos.

Solução:

16.1) Derivar cada classe constante do Diagrama de Classes elaborado com a aplicação do padrão 13 (Abstrair Diagrama de Pseudo-Classes);

16.2) Criar um método para cada funcionalidade presente em relação à manipulação e ao acesso aos dados da(s) tabela(s) com as quais cada classe se relaciona;

16.3) Caso necessário, criar atributos que facilitem a manipulação dos dados;

16.4) Criar um Diagrama de Classes de Projeto para documentação das classes definidas nos Passos 16.1 a 16.3.

Produto Obtido:

Diagrama de Classes de Projeto.

Padrões Relacionados:

Esse padrão utiliza como entrada, a saída da aplicação do padrão 13 (Abstrair Diagrama de Pseudo-Classes). Continuando a engenharia avante, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no Diagrama de Classes de Projeto: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia avante: padrão 17 (Construir Diagramas de Seqüência).

17. Nome: Construir Diagramas de Seqüência

Intuito:

Documentar a lógica de negócio do sistema legado.

Problema:

Descrever a lógica de negócio do sistema legado para facilitar sua futura re-implementação.

Contexto:

Representar a interação dos métodos obtidos no padrão 16, Elaborar Diagrama de Classes de Projeto, pode ser uma tarefa complexa;

Métodos já definidos a partir das Descrições de Casos de Uso viabilizam a solução desse problema.

Solução:

Construir um Diagrama de Seqüência, a partir de cada Descrição de Casos de Uso, obtidas a partir da aplicação dos padrões 12 e 15 (Descrever Casos de Uso do MASA e Descrever Casos de Uso do MAS), respectivamente. Devem ser considerados os métodos disponíveis nas classes para a construção dos diagramas.

Produto Obtido:

Diagramas de Seqüência do Sistema.

Padrões Relacionados:

Esse padrão utiliza como entradas, as saídas da aplicação dos padrões 12, 15 e 16 (Descrever Casos de Uso do MASA, Descrever Casos de Uso do MAS e Elaborar Diagrama de Classes de Projeto). Continuando a engenharia avante, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade nos Diagramas de Seqüência: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- inspecionar o andamento do processo em relação ao Plano para Realização da Reengenharia: padrão 3 (Acompanhar o Progresso do Processo de Reengenharia);
- continuar a engenharia avante: padrão 18 (Implementar as Classes).

GRUPO 7 – Re-implementar o Software.

18. Nome: Implementar as Classes

Intuito:

Codificar a lógica de negócio do sistema, previamente modelada durante a aplicação dos padrões anteriores.

Problema:

Definir os atributos, tratando o encapsulamento dos mesmos, além de implementar os métodos de acesso a esses atributos, de forma a tratar a lógica de negócio de cada objeto.

Contexto:

As classes referentes à lógica de negócios do sistema descritas no Diagrama de Classes de Projeto resultante da aplicação do padrão 16 (Elaborar Diagrama de Classes de Projeto) formam a base para a aplicação desse padrão;

O sucesso da aplicação desse padrão depende, em grande parte, da qualidade da modelagem previamente elaborada, ou seja, da correta expressão da lógica de negócio nos produtos de trabalho criados com a aplicação dos padrões anteriores;

O engenheiro de software tem facilidade na expressão da lógica de negócio por meio de diagramas.

Solução:

Para cada classe relacionada a lógica de negócio presente no Diagrama de Classes de Projeto, deve-se:

18.1) Criar a estrutura da classe com as seções destinadas aos atributos e métodos públicos, protegidos e/ou privados;

18.2) Criar o código-fonte correspondente aos atributos simples e aos atributos originados de relacionamentos com outras classes, verificando sua visibilidade (em qual seção serão declarados) e tipo;

18.3) Criar o código-fonte para os métodos que tratam das funcionalidades do objeto. Para isso, deve-se verificar a Lista de Mapeamento MASA x MAS, a Lista de Correspondência de Métodos Anômalos e os Diagramas de Seqüência, no sentido de buscar a origem do código-fonte legado que deverá ser re-implementado em cada método. Os métodos são implementados de acordo com a seqüência apresentada no Diagrama de Seqüência para que a funcionalidade do sistema legado seja preservada.

Produto Obtido:

Código-fonte das classes relativas à lógica de negócios do software.

Padrões Relacionados:

A saídas da aplicação dos padrões 16 e 17 (Elaborar Diagrama de Classes de Projeto e Construir Diagramas de Seqüência), formam a base para a aplicação desse padrão, aplicado em conjunto com os demais padrões do *grupo*, de forma a completar, simultaneamente, a interface, o acesso a dados e as funcionalidades previstas no software.

Em conjunto com a aplicação dos padrões, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no código fonte das classes implementadas: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração).

19. Nome: Implementar a Lógica de Armazenamento

Intuito:

Definir a lógica de armazenamento do software.

Problema:

Implementar os objetos responsáveis pelo encapsulamento dos métodos de acesso aos dados.

Contexto:

As classes que provêem a interface com o banco de dados, documentadas no Diagrama de Classes de Projeto construído a partir da aplicação do padrão 16 (Elaborar Diagrama de Classes de Projeto) são utilizadas como entrada;

A orientação a objetos permite que o código-fonte seja totalmente organizado, de forma que a hierarquia dos objetos é definida através de vários níveis de abstração;

Para aplicações multi-usuários, esse padrão deve contemplar a implementação, nos métodos, da manutenção da integridade dos dados, aspecto disponível em bancos de dados como *Oracle* e *Microsoft* SQL *Server*.

Solução:

Para cada classe relacionada a lógica de armazenamento dos dados, presente no Diagrama de Classes de Projeto, deve-se:

19.1) Criar a estrutura da classe, com as seções destinadas aos atributos e métodos públicos, protegidos e/ou privados;

19.2) Criar o código-fonte correspondente aos atributos simples e aos atributos originados de relacionamentos com outras classes, verificando sua visibilidade (em qual seção serão declarados) e tipo;

19.3) Criar o código-fonte para os métodos que tratam do acesso aos dados do objeto. Para isso, deve-se verificar a Lista de Mapeamento MASA x MAS e a Lista de Correspondência de Métodos Anômalos, no sentido de buscar a origem do código-fonte legado que deverá ser re-implementado em cada método.

Produto Obtido:

Código-fonte das classes relativas à lógica de armazenamento e manipulação de dados.

Padrões Relacionados:

Esse padrão relaciona-se com os padrões 16 e 18 (Elaborar Diagrama de Classes de Projeto e Implementar as Classes), por suas saídas serem utilizadas aqui como entrada. Continuando a engenharia avante, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no código-fonte das classes relativas à lógica de armazenamento e manipulação de dados do software: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração);
- continuar a engenharia avante: padrão 6 (Implementar a Lógica de Apresentação).

20. Nome: Implementar a Lógica de Apresentação

Intuito:

Redefinir a interface do software sem a utilização de componentes de acesso direto ao banco de dados.

Problema:

Implementar a interface do software e a ligação aos métodos das classes anteriormente criadas.

Contexto:

No *Delphi*, software implementado sem a utilização do paradigma orientado a objetos utilizam, para apresentar os dados na interface, componentes que manipulam diretamente o banco de dados, sem a interferência de procedimentos ou funções definidos pelo programador. Já, software implementado de acordo com o paradigma orientado a objetos, fazem uso de eventos que utilizam os métodos definidos nas classes implementadas com a aplicação dos padrões 18 e 19 (Implementar as Classes e Implementar a Lógica de Armazenamento do Software), de forma a prover a apresentação dos dados.

- Os componentes de manipulação de dados presentes no ambiente *Delphi* facilitam a implementação. Porém, estes somente podem ser utilizados no caso de acesso direto aos dados, o que fere o conceito de encapsulamento do paradigma orientado a objetos;

- Há componentes gráficos (campos de edição e grids) que apresentam dados fornecidos pelo programador, podendo ser utilizados para mostrar a saída de métodos das classes por este definidas;
- A codificação da interface orientada a objetos no ambiente Delphi é, em primeira instância, mais complexo que a programação procedimental, pelo fato do acesso aos dados ter que ser completamente implementado pelo programador de forma a apenas trocar informações com as interfaces. Porém, uma vez codificadas as interfaces básicas, estas podem ser reutilizadas, bem como as classes de auxílio à apresentação dos dados.

Solução:

20.1) Definir as interfaces do software de forma a prover todos os recursos disponíveis no legado utilizando somente componentes simples de apresentação de dados;

20.2) Implementar, a partir de eventos, o acesso aos métodos das classes codificadas durante a aplicação dos padrões 18 e 19.

Produtos Obtidos:

Interfaces e código-fonte referente à lógica de apresentação do software.

Padrões Relacionados:

Os padrões 17, 18 e 19 (Construir Diagramas de Seqüência, Implementar as Classes e Implementar a Lógica de Armazenamento) são complementares a esse padrão pelo fato de suas saídas serem utilizadas aqui como entradas. Complementando o processo de engenharia avante, o engenheiro de software deve:

- realizar a inspeção de garantia da qualidade no código-fonte referente à lógica de apresentação do software: padrão 4 (Realizar Inspeção de Garantia da Qualidade);
- atualizar o controle de configuração: padrão 5 (Controlar a Configuração).

4. Considerações Finais

A forma de descrição do processo por meio de padrões facilita o aprendizado pelos engenheiros de software, bem como a aplicação repetitiva de partes isoladas do processo para prover refinamentos de forma a gerar um produto final com qualidade. Ressalta-se que os padrões existentes na FaPRE/OO, originalmente desenvolvidos para serem utilizados em sistemas legados implementados em *Clipper* foram instanciados para *softwares* implementados em *Delphi*, comprovando assim que o processo de reengenharia apresentado pode ser utilizado, após algumas adaptações, independente da linguagem de programação do software legado.

Os padrões do PRE/OO foram elaborados para serem utilizados de forma evolutiva, garantindo um melhor resultado aos produtos resultantes dessa aplicação. Seus resultados, documentados com formato padronizado auxiliam os engenheiros de software no entendimento, manuseio e continuidade da reengenharia.

Qualquer sistema implementado no ambiente *Delphi*, do qual se tenha o código-fonte, tabelas de dados e programa executável, pode ser submetido aos padrões propostos. Como trabalho futuro pretende-se derivar os padrões propostos para sua aplicação em sistemas desenvolvidos em outros ambientes, como o *Visual Basic*. Além disso, pretende-se pesquisar a construção de ferramentas para automatização de alguns padrões que consistem de passos cabíveis de realização automática, como os padrões 6, 7 e 8 (Elaborar Lista de Procedimentos e Funções, Elaborar Lista de "Chama/Chamado Por" e Modelar Dados do Software Legado).

Outra possibilidade de trabalho futuro é a adaptação de outras KPAs do SW-CMM relacionadas ao processo de software, tornando-as novos padrões de qualidade do processo de reengenharia.

5. Agradecimentos

Agradecemos ao shepherd Tiago Lima Massoni pelas sugestões e acompanhamento dados a este trabalho.

6. Referências

[1] Cantu, M. **"Dominando o Delphi 6: a Bíblia"**, Makron Books. Isbn: 8534614083. 2001.

[2] Demeyer, S.; Ducasse, S.; Tichelaar, S. **"A Pattern Language for Reverse Engineering"**, 4th European Conference on Pattern Languages of Programming and Computing, Paul Dyson (Ed.), Germany. July, 1999.

[3] Demeyer, S.; Ducasse, S.; Nierstrasz, O. **"A Pattern Language for Reverse Engineering".** 5th European Conference on Pattern Languages of Programming and Computing. Pages 189-208. July 2000a.

[4] Demeyer, S.; Ducasse, S.; Nierstrasz, O. **"Tie Code and Questions: a Reengineering Pattern".** 5th European Conference on Pattern Languages of Programming and Computing. Pages 209-217. 2000b.

[5] Lemos, G. S. **"PRE/OO – Um Processo de Reengenharia Orientada a Objetos com Ênfase na Garantia da Qualidade"**, São Carlos-SP. Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos. Agosto/2002. Disponível em http://www.svconsultoria.com.br/pessoal/gizelle/

[6] OMG. **"Unified Modeling Language Specification"**. Object Management Group. Versão 1.3. Junho 1999.

[7] Paulk, M.; Weber, C.; Wise, C.; Withey, J. **"Key Practices of the Capability Maturity Model, Version 1.0".** Software Engineering Institute, CMU/SEI-91-TR-25. 1991.

[8] Penteado, R. A. D., **"Um Método para Engenharia Reversa Orientada a Objetos"**. Tese de Doutorado em Física Computacional apresentada ao Instituto de Física de São Carlos, Universidade de São Paulo. 237 páginas. 1996.

[9] Penteado, R. A. D.; Masiero, P. C.; Cagnin, M. I. **"An Experiment of Legacy Code Segmentation to Improve Maintainability".** III European Conference on Software Maintenance and Reengineering – IEEE. Amsterdã, Holanda. Páginas 111-119. 1999.

[10] Pressman, R. S. **"Software Engineering: A Practitioner's Approach".** Fifth Edition. McGraw-Hill Higher Education. 2001.

[11] Recchia, E. L., **"Engenharia Reversa e Reengenharia Baseadas em Padrões"** Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos. Junho/2002.

[12] Recchia, E. L.; Penteado, R. **"Avaliação da Aplicabilidade da Linguagem de Padrões de Engenharia Reversa de Demeyer a Sistemas Legados Procedimentais".** Artigo apresentado em 2nd Latin American Conference on Pattern Languages of Programming – Software Pattern Applications. (SugarLoafPLoP–SPA), Itaipava-RJ, Agosto/2002.

Padrões para o Processo de Engenharia Avante na Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais

Edson Luiz Recchia
Univ. Anhembi Morumbi
erecchia@terra.com.br

Gizelle S. Lemos
Depto. de Computação
Univ. Federal de São Carlos
gizelle@dc.ufscar.br

Rosângela Penteado
Depto. de Computação
Univ. Federal de São Carlos
rosangel@dc.ufscar.br

Rosana T.V. Braga
ICMC / USP
Univ. de São Paulo
rtvb@icmc.usp.br

Resumo

Padrões de reengenharia, incluindo engenharia reversa e engenharia avante são desenvolvidos por profissionais experientes, registrando como esses profissionais conduzem esses processos. Esses padrões têm como objetivo auxiliar engenheiros de software a conduzir esses processos em seus sistemas legados. A Família de Padrões de Reengenharia - FaPRE/OO - para a Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais, foi criada para conduzir esses processos a partir de sistemas legados procedimentais, tendo sistemas orientados a objetos como alvo. Foram apresentados no SugarLoafPLoP'2002 os padrões da FaPRE/OO para o processo de engenharia reversa. Neste trabalho, são apresentados os padrões para o processo de engenharia avante, ilustrando-se, passo a passo, o seu uso.

Abstract

Reengineering Patterns, including reverse engineering and forward engineering patterns, have been developed by experienced software engineers, in order to track how these processes are conducted in legacy systems. The goal of these patterns is to help software engineers to conduct these processes in their systems. The Family of Patterns for Object-Oriented Reengineering of legacy systems - FaPRE/OO - was developed to conduct these processes from procedural legacy systems to object-oriented target systems. At SugarLoafPLoP'2002, the patterns of FaPRE/OO for reverse engineering were presented. In this paper, forward engineering process patterns are presented step by step, illustrating their use.

1. Introdução

O processo de reengenharia é amplamente reconhecido como um dos desafios mais significativos para engenheiros de software. O problema é comum, afetando todos os tipos de organização; sério, pois uma falha na reengenharia pode dificultar os esforços da organização para manter-se competitiva; persistente, pois parece não haver razão para ficar confiante de que os novos sistemas não vão ser também os legados de amanhã (Stevens *et al*. [17]).

Dewar *et al*. [6] relatam que pretendiam entender como profissionais experientes conduziam a reengenharia de sistemas legados, a fim de desenvolver técnicas e materiais para transferir essa experiência. Em particular, queriam tratar o problema de sintetizar experiência com reengenharia de sistemas e de organizações, para ajudar engenheiros de software a adquiri-las, em paralelo.

Os padrões para o processo de reengenharia, especialmente para engenharia avante orientada a objetos de sistemas legados procedimentais, abrangem, particularmente, o domínio de sistemas de informação, bem como técnicas de condução desses processos.

A abordagem proposta para resolver esse problema é uma Família de Padrões de Reengenharia, denominada FaPRE/OO, contendo um conjunto de três grupos de padrões para o Processo de Engenharia Reversa [14,15] e um grupo de padrões para o Processo de Engenharia Avante. Dessa forma, a partir de um código pouco estruturado que mistura interface com lógica, obtém-se melhorias significativas com o novo código orientado a objetos, facilitando, assim, a manutenção, melhorando o desempenho do sistema, possibilitando a inclusão de novas regras de negócio e aprimoramentos que utilizem tecnologias atuais, como internet.

Neste trabalho, mostra-se como os padrões para o processo de engenharia avante da FaPRE/OO foram elaborados a partir de sistemas legados em Clipper [14,15,16] e Cobol [2], para sistemas alvo em Delphi [14,15,16] e Java [2]. Os padrões da FaPRE/OO foram aplicados a casos concretos de elaboração de orçamentos de obras da construção civil [14,15,16], controle de material em uma mineradora [2] e controle contábil [8].

Este trabalho está organizado da seguinte forma: na Seção 2, são introduzidos os padrões para o processo de engenharia avante, propostos pela Família de Padrões de Reengenharia (FaPRE/OO); na Seção 3, são apresentados os comentários finais.

2. Padrões para o Processo de Engenharia Avante da Família de Padrões de Reengenharia - FaPRE/OO

A FaPRE/OO é uma Família de Padrões de Reengenharia para gerar processos de engenharia reversa e de engenharia avante orientados a objetos, a partir de sistemas legados procedimentais. É composta de quatro grupos, cada um agrupando os padrões relacionados a situações similares da reengenharia, sendo três grupos para o processo de engenharia reversa e um para o processo de engenharia avante. A Figura 1 ilustra graficamente os grupos e os padrões existentes em cada um deles.

Grupo 1: Modelar os Dados do Legado: agrupa padrões que extraem informações a partir dos dados e do código fonte do sistema legado gerando o MER [14,15] - Modelo Entidade-Relacionamento (visão procedimental dos dados) e o MASA [10,11,12] - Modelo de Análise do Sistema Atual - Diagrama de Pseudo-Classes (visão orientada a objetos dos dados). Esses padrões conduzem as ações do engenheiro de software quando se tem o primeiro contato com um sistema de software. Fazem parte desse grupo os seguintes padrões: Iniciar Análise dos Dados; Definir Chaves; Identificar Relacionamentos e Criar Visão OO dos Dados.

Grupo 2: Modelar a Funcionalidade do Sistema: agrupa padrões para obter a funcionalidade do sistema, criando modelos que recuperem as regras de negócio da empresa, contidas no sistema legado. Esses padrões habilitam o engenheiro de software a obter um entendimento detalhado dos componentes (partes) do sistema de software, aprofundando, assim, sua compreensão sobre o sistema legado. Fazem parte desse grupo os seguintes padrões: Obter Cenários; Construir Diagramas de Casos de Usos; Elaborar a Descrição de Casos de Usos e Tratar Anomalias.

Grupo 3: Modelar o Sistema Orientado a Objetos: agrupa padrões para se obter o diagrama de classes e os diagramas de seqüência do sistema, através da interação dos produtos obtidos pelos padrões dos grupos anteriores. Esses padrões habilitam o engenheiro de software a obter o MAS [10,11,12] - Modelo de Analise do Sistema, sendo o modelo

orientado a objetos a servir de suporte ao processo de engenharia avante. Fazem parte desse grupo os seguintes padrões: Definir as Classes; Definir Atributos; Analisar Hierarquias; Definir Métodos e Construir Diagramas de Seqüência.

Grupo 4: Gerar o Sistema Orientado a Objetos: agrupa padrões que completam o processo de reengenharia do sistema, transformando o sistema legado, do paradigma procedimental para o paradigma orientado a objetos. Fazem parte desse grupo os seguintes padrões: Definir a Plataforma; Converter o Banco de Dados; Implementar os Métodos e Realizar Melhorias na Interface.

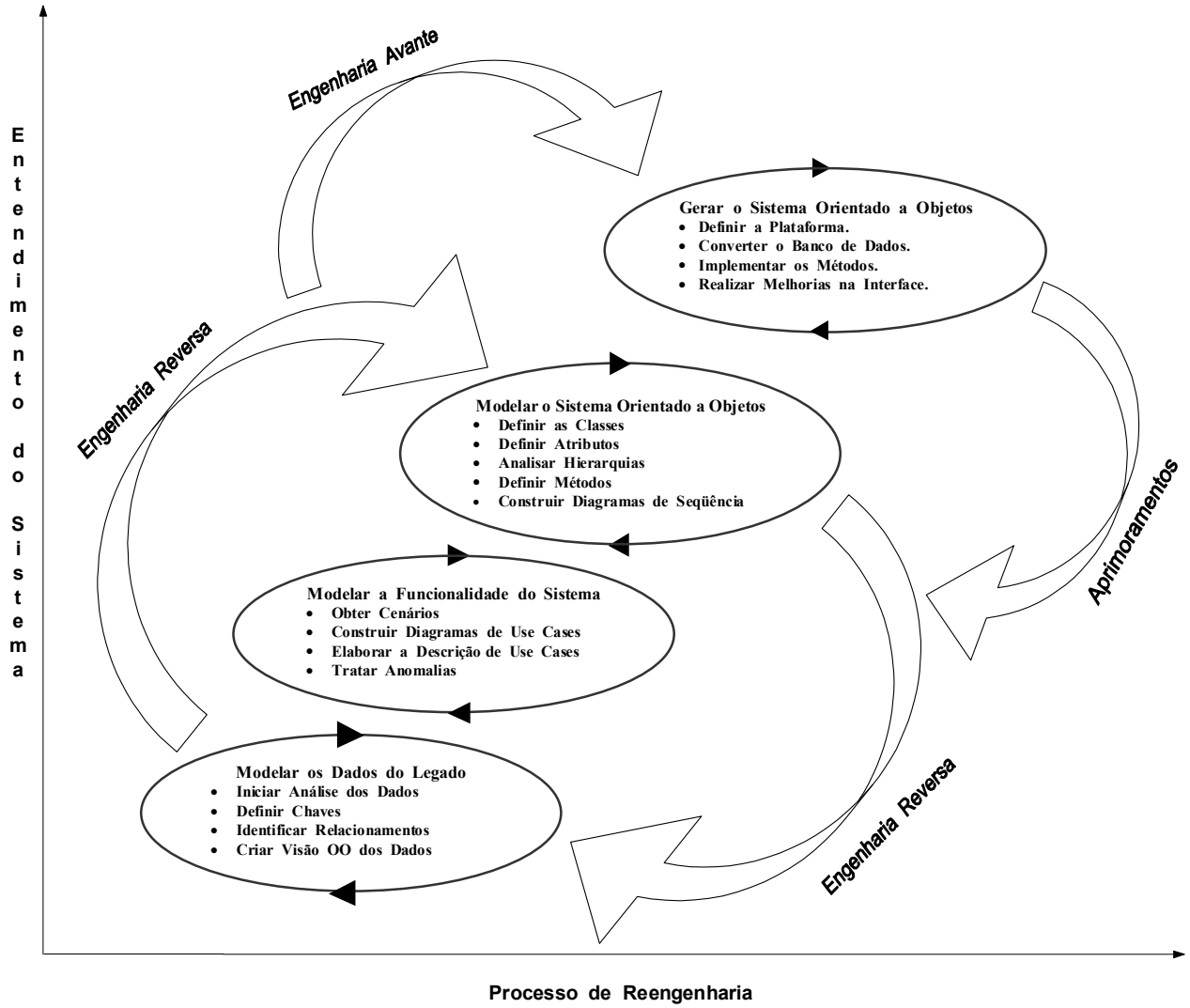


Figura 1: FaPRE/OO - Família de Padrões para Reengenharia Orientada a Objetos [14,15]

O leitor observará que muitos padrões terão como entrada a saída de algum padrão aplicado anteriormente, exigindo então sua aplicação seqüencial. No entanto, como se pode observar na Figura 1, o modelo é evolutivo, podendo-se, de qualquer padrão, avançar ou retroceder após a sua aplicação. A aplicação seqüencial somente será necessária durante o primeiro ciclo. Como exemplo, pode-se citar o caso de sistemas legados com centenas de arquivos de dados, divididos em módulos funcionais (compras, vendas, financeiro, estoque,

entre outros). Inicia-se a engenharia reversa a partir de um módulo qualquer, aplicando-se todos padrões apresentados na Figura 1, construindo-se assim todos os modelos envolvidos. Com isso, vai-se dominando paulatinamente o sistema, podendo-se incorporar outros módulos, num processo cíclico e progressivo.

Embora a FaPRE/OO seja composta de padrões para tratar tanto a engenharia reversa como a engenharia avante, somente os padrões referentes ao Processo de Engenharia Avante são considerados neste trabalho.

Os padrões de engenharia avante apresentados a seguir são descritos de acordo com o seguinte formato: Nome, Intuito, Contexto, Problema, Solução e Padrões Relacionados. Esse formato foi adotado com base nos já existentes na literatura (Demeyer *et al*. [4], Stevens *et al*. [17]), de acordo com a experiência dos autores, além do fato de não existirem padrões de reengenharia orientados a objetos que abordem sistemas legados procedimentais.

O item correspondente à Solução a ser adotada é apresentado na forma de passos, sempre que necessário e o item Contexto expressa, também, as Forças do padrão.

O Grupo Gerar o Sistema Orientado a Objetos.

Agrupa padrões para o processo de engenharia avante de um sistema, transformando o sistema legado, do paradigma procedimental para o paradigma orientado a objetos.

2.1 Nome: Definir a Plataforma

Intuito:

Definir a plataforma de desenvolvimento do novo sistema conforme as exigências do Negócio e dos Objetivos Estratégicos da organização empresarial.

Contexto:

- O atual e o futuro ambiente operacional da organização empresarial são objetos de estudo que devem incluir as seguintes análises:
  (1) O quê de fato já existe;
  (2) Quais os atuais e os futuros recursos operacionais disponibilizados;
  (3) A cultura da organização;
  (4) O histórico de desenvolvimentos realizados;
  (5) O foco estratégico da organização;
  (6) Quais o processos de negócios que serão suportados.
- Avaliar as considerações anteriores juntamente com a área estratégica da organização, pois a mudança de um ambiente operacional envolve custos, tais como:
  (1) Treinamento dos usuários na nova tecnologia;
  (2) Investimentos em infra-estrutura que envolvem a solução adotada, podendo ser necessário adquirir: novos computadores servidores e estações de trabalho, roteadores e switches para interligação remota dessas estações. Também, pode ocorrer a necessidade de reestruturar toda a rede de informática, efetuando-se, por exemplo, a mudança de cabos coaxiais para a tecnologia de par-trançado;
  (3) Investimento em todo o software necessário para a implementação da solução escolhida, podendo ser necessário adquirir: um novo sistema operacional, tal como Novell, Windows 2000 Server; um novo Sistema Gerenciador de Banco de Dados (SGBD) Relacional; uma ferramenta de desenvolvimento, tal como Java ou Delphi; softwares de apoio, tal como backup e correio interno.
  (4) Investimento no desenvolvimento do aplicativo. Poder-se-á optar pela terceirização desses serviços, ou treinar toda a equipe técnica da empresa na nova

tecnologia, considerando que essa mesma equipe deverá manter o atual sistema legado em operação durante toda a fase de desenvolvimento do novo aplicativo.

- O engenheiro de software tem à sua disposição cópias de demonstração de todos os pacotes, utilitários e ferramentas de suporte técnico.
- O engenheiro de software tem um bom conhecimento de técnicas de reuniões para se obter um consenso da área estratégica da organização.

Problema:

Organizações empresariais que desejam modernizar a Tecnologia da Informação usada como suporte às realizações dos seus Negócios, enfrentam dificuldades na condução desse processo.

Solução:

(a) Escolher o Ambiente Operacional.

Escolher o ambiente que dará suporte ao negócio da empresa, por exemplo, dentre as seguintes possibilidades:

Ambiente *Stand-alone*: caracterizado por uma aplicação mono-usuário. Esse ambiente utiliza os recursos de atualizações em *Cache* como transações.

Ambiente de Rede: caracterizado por uma aplicação multiusuário. Esse ambiente utiliza os recursos de atualização Cliente/Servidor como transações, podendo ser uma rede local (LAN), metropolitana (MAN) ou de grande abrangência (WAN).

Ambiente *Web*: caracterizado por uma aplicação multicamadas como transações. Essa arquitetura é composta de três camadas lógicas: camada de persistência (acesso ao SGBD); camada do servidor de aplicação (gerenciamento das regras de negócio); camada de apresentação (interface com cliente). Um protocolo de transporte é o elo entre as camadas de persistência, servidor de aplicação e de apresentação. Protocolos típicos são DCOM, CORBA, HTTP e soquetes TCP/IP. O elo entre o servidor de aplicação e a camada de persistência é usualmente feito por soquetes TCP/IP.

(b) Escolher o Sistema Gerenciador de Banco de Dados (SGBD) Relacional.

Escolher um sistema gerenciador de banco de dados relacional, dentre as várias opções existentes, por exemplo: Oracle, Sybase, SQL-Server, Informix, Ingres, InterBase, Progress, etc.

(c) Escolher a Linguagem de Implementação e o Ambiente de Desenvolvimento.

Escolher uma linguagem de implementação, dentre as várias opções existentes: Java, Delphi, entre outras.

Exemplo:

Como exemplo de definição da plataforma de desenvolvimento do novo sistema conforme as exigências do Negócio e dos Objetivos Estratégicos da organização empresarial, considere a aplicação desse padrão ao caso real do sistema de elaboração de orçamentos de obras da construção civil - Sistema SIGO [14,15,16].

(a) Escolher o Ambiente Operacional.

Decidiu-se pelo **Ambiente de Rede.** A escolha desse ambiente operacional deveu-se ao fato de aproveitar os investimentos em hardware já realizados pelas construtoras da região e escritórios de engenharia civil que utilizam a versão caracter do Sistema SIGO:

- Elas possuiam uma rede local implantada com a tecnologia de par-trançado, bem como microcomputadores Pentium. Nenhuma dessas empresas trabalha com estações remotas de acesso ao Sistema SIGO.

(b) Escolher a Linguagem de Implementação e o Ambiente de Desenvolvimento.
Escolher uma linguagem de implementação, dentre as várias opções existentes: Ada, C++, Delphi, Java, entre outras.
Optou-se pelo ambiente de desenvolvimento Delphi [1] pelo fato de ser uma opção para implementar a funcionalidade de sistemas de informação no paradigma de orientação a objetos. Esse ambiente de desenvolvimento tem obtido grande aceitação no mercado.

(c) Escolher o Sistema Gerenciador de Banco de Dados (SGBD) Relacional.
Escolher um sistema gerenciador de banco de dados relacional, dentre as várias opções existentes: Oracle, Sybase, SQL-Server, Informix, Ingres, InterBase, Progress.
Optou-se pelo InterBase , por já acompanhar o ambiente de desenvolvimento Delphi e por estar consolidado no mercado.

Padrões Relacionados:

Pode-se considerar a aplicação, em paralelo, do padrão *Arquitetural Layer* [19,21], para considerar os aspectos de segurança não abordados pelo padrão Definir a Plataforma.

2.2 Nome: Converter o Banco de Dados

Intuito:

Criar as estruturas de dados do sistema, fisicamente, de acordo com o Diagrama de Classes.

Contexto:

- A grande quantidade de arquivos de dados e a geração não automática desses arquivos para tabelas de um banco de dados, torna viável o desenvolvimento de um processo para a realização da conversão desses arquivos de dados em tabelas de um SGDB.
- Têm-se as classes geradas, a partir dos arquivos de dados, viabilizando esse processo. Essas classes são obtidas quando da aplicação dos padrões Definir as Classes; Definir Atributos e Analisar Hierarquias; no processo de engenharia reversa. Logo, essas classes serão as tabelas a serem criadas no SGBD.
- O engenheiro de software tem um bom conhecimento da linguagem de implementação, para derivar os arquivos de dados do sistema legado para tabelas de banco de dados relacional.
- O engenheiro de software tem à sua disposição ferramentas de conversão de dados, as quais acompanham a linguagem de implementação e o SGBD a serem utilizados na reengenharia.

Problema:

Gerar um banco de dados, a partir dos arquivos de dados que contêm os dados do sistema legado.

Solução:

Usar a ferramenta adequada à linguagem de implementação e ao SGBD escolhidos, para converter os arquivos de dados do sistema legado em tabelas do banco de dados relacional escolhido.

Exemplo:

Usar as ferramentas InterBase Windows ISQL, SQL Explorer e Data Pump, para converter os arquivos de dados (.dbf's) do sistema legado SIGO em tabelas do SGBD InterBase, de acordo com os seguintes passos:

(a) Criar um Banco de Dados, com extensão .gdb, utilizando a ferramenta: InterBase Windows ISQL. Ex.: SIGO.GDB;

(b) Criar um *alias* (objeto lógico) para a pasta (objeto físico) SIGO.GDB, que está no diretório C:\ SIGO, utilizando a ferramenta: *SQL Explorer*. Ex.: *Alias* SIGO;

(c) Para cada classe definida no Diagrama de Classes do Sistema, obtido no Processo de Engenharia Reversa, criar a respectiva tabela no banco de dados, utilizando a ferramenta *SQL Explorer*. Ex.:

```
CREATE TABLE CLIENTES (
     CODIGO-CLIENTE     INTEGER       NOT NULL,
     NOME               VARCHAR(40)   NOT NULL,
     ENDERECO           VARCHAR(30),
     CODIGO-PAIS        INTEGER       NOT NULL   )
```

(d) Para cada tabela definida no *alias* SIGO, transportar o respectivo arquivo de dados (.dbf) do sistema legado para o banco de dados, utilizando a ferramenta *Data Pump*. Essa ferramenta apenas transporta os dados de um arquivo de dados (.dbf) para um banco de dados (.gdb), não realizando a adição de *flags*, por exemplo *NOT NULL*. Ex.: Transportar o arquivo de dados Cliente.dbf para a Tabela temporária Transporte;

(e) Como o *Data Pump* apenas transfere dados, importar, para a tabela CLIENTE, os dados da tabela TRANSPORTE, usando o comando *select* de uma declaração SQL, sendo que, nesse momento são aplicadas as regras constantes na tabela CLIENTE, por exemplo, *NOT NULL*:

```
INSERT  INTO CLIENTE ( CODIGO-CLIENTE, NOME, ENDERECO, CODIGO-PAIS )
     SELECT  CODIGO, NOME, ENDERECO, CODPAIS
     FROM    TRANSPORTE;
```

(f) Para cada tabela gerada no banco de dados, criar um gerador de códigos para a geração automática de códigos pelo SGBD ao se inserir um novo registro.
Considere que o arquivo CLIENTE.DBF possuía 5000 registros, então inicializa-se o gerador de código `GERAR-CODIGO-CLIENTE` com o valor 5001, por meio da seguinte declaração SQL:

```
CREATE GENERATOR GERAR-CODIGO-CLIENTE
     SET GENERATOR GERAR-CODIGO-CLIENTE TO 5001;
```

(g) Criar um *Trigger* para a geração automática de código pelo SGBD ao se inserir um novo registro. Neste caso, o *Trigger* é disparado automaticamente toda vez que um novo registro for incluído. Para a tabela CLIENTE, a seguinte declaração SQL deve ser construída:

```
CREATE  TRIGGER  SETAR-CODIGO-CLIENTE
     FOR     CLIENTE
     BEFORE  INSERT   POSITION 0   AS
     BEGIN
         new.CODIGO-CLIENTE  =  gen-id( GERAR-CODIGO-CLIENTE, 1 )
     END;
```

(h) Criar a Chave Primária da tabela. Para a tabela CLIENTE, a seguinte declaração SQL deve ser construída:

```
ALTER TABLE  CLIENTE
      ADD CONSTRAINT  CLIENTE-CHAVE-PRIMARIA
      PRIMARY KEY     ( CODIGO-CLIENTE );
```

(i) Criar a Chave Estrangeira da tabela, se existir. Para a tabela CLIENTE, a seguinte declaração SQL deve ser construída:

```
ALTER TABLE  CLIENTE
      ADD CONSTRAINT  CLIENTE-CHAVE-ESTRANGEIRA
      FOREIGN KEY     ( CODIGO-PAIS )
```

```
        REFERENCES      PAIS
              ON DELETE   CASCADE
              ON UPDATE   CASCADE;
```

(j) Criar os índices de acesso para a tabela, se existirem. Para a tabela CLIENTE, a seguinte declaração SQL deve ser construída, para a criação de um índice de acesso ordenado pelo atributo NOME:

```
CREATE UNIQUE ASCENDING INDEX  INDICE-CLIENTE-NOME
     ON CLIENTE ( NOME );
```

Padrões Relacionados:

Pode-se considerar a aplicação, em paralelo, do padrão Examinar o Banco de Dados [16].

2.3 Nome: Implementar os Métodos

Intuito:

Escrever os métodos obtidos pela engenharia reversa na linguagem de programação escolhida para o processo de engenharia avante.

Contexto:

- Transformar em programas a representação textual das Descrições de Casos de Usos (códigos fonte do legado) é difícil.
- O engenheiro de software tem à sua disposição todos os métodos obtidos na engenharia reversa por meio do padrão Definir Métodos.
- O engenheiro de software tem um bom conhecimento da linguagem de implementação para derivar os diagramas de seqüências, obtidos por meio do padrão Construir Diagramas de Seqüências, em linhas de código.

Problema:

Necessidade de converter códigos fonte do sistema legado em códigos orientados a objetos, na linguagem de implementação escolhida para a reengenharia.

Solução:

Escrever o código para cada um dos métodos obtido pelos padrões Definir Métodos e Construir Diagramas de Seqüência, na linguagem de implementação escolhida.

Deve-se considerar a aplicação, em paralelo, do padrão de projeto *Persistence Layer* [20] e dos padrões: Verificar as Invocações dos Métodos, Observar a Execução dos Componentes e Refazer para Entender [16].

Exemplo:

- A aplicação SIGO foi implementada para ser executada no Ambiente Multiusuário (rede local), que utiliza os recursos de atualizações Cliente/Servidor como transações.
- Exemplifica-se esse padrão com a implementação dos métodos `botãoGravarClick()` e `ClienteAfterPost()`, para a tabela CLIENTE. Para as demais tabelas o mecanismo é o mesmo. A Figura 2 mostra esses métodos implementados na ferramenta Delphi.
- Observe-se, na Figura 2, que a atualização dos dados ocorre quando o usuário pressiona um botão Gravar, acionando evento *OnClick* desse botão, o qual executa automaticamente o procedimento `botãoGravarClick()`.
- O procedimento `botãoGravarClick()` atualiza o *buffer* de memória do computador do usuário (estação *Cliente*) por meio do comando *Post*. Esse comando, por sua vez, ao ser executado aciona o evento *AfterPost* da *query* Cliente, que representa a tabela CLIENTE na estação *Cliente*. O evento *AfterPost* executa automaticamente o

procedimento `ClienteAfterPost()`. Finalmente, esse procedimento efetua a atualização física dos dados no servidor de aplicações.

Padrões Relacionados:

- Padrão Verificar as Invocações dos Métodos. Objetivo: saber como uma classe está relacionada com outra verificando os parâmetros definidos nos métodos da interface da classe (assinatura do método). Vínculo ao padrão: obter o relacionamento entre classes.
- Padrão Observar a Execução dos Componentes. Objetivo: obter um entendimento detalhado do comportamento de uma parte do código, através da execução de seus componentes (encapsulamento). Vínculo ao padrão: é preciso obter o entendimento detalhado de uma parte encapsulada do código.
- Padrão Refazer para Entender. Objetivo: obter um melhor entendimento de uma parte específica do código fonte, refazendo-a. "Refazer para Entender" é o processo de modificar um sistema de software, de tal maneira que o comportamento externo do código não seja alterado, mas sua estrutura interna seja melhorada. Vínculo ao padrão: compreender um particular trecho de código que aparenta ser importante mas é muito difícil de analisá-lo completamente.

2.4 Nome: Realizar Melhorias na Interface

Intuito:

Conceber sistemas que atendam, cada vez mais, às necessidades dos usuários em relação não apenas a critérios de funcionalidade (conjunto de tarefas desempenhadas pelo sistema), mas também à usabilidade.

Contexto:

- Exige do engenheiro de software habilidade em conseguir transformar telas do ambiente caracter em telas do ambiente gráfico (GUI).
- Requer, do engenheiro de software, experiência no desenvolvimento de interfaces que:
  (1) Sejam consistentes;
  (2) Tenham atalhos para usuários experientes;
  (3) Tenham *Feedback* informativo;
  (4) Sejam organizada quanto aos diálogos de interação;
  (5) Previnam erros de usuários;
  (6) Possibilitem a reversibilidade de ações;
  (7) Reduzam a necessidade de memorização.
- O engenheiro de software tem à sua disposição todas as telas de tratamento da informação, do sistema legado.
- O engenheiro de software tem um bom conhecimento da linguagem de implementação para construir interfaces que viabilizem:
  (1) Facilidade de aprendizado;
  (2) Facilidade de utilização;
  (3) Ser intuitiva;
  (4) Diálogo simples e natural;
  (5) Velocidade na execução das tarefas;
  (6) Mensagens de erros consistentes;
  (7) Satisfação subjetiva.

```
unit UnitCliente;

interface

type
  TFormCliente = class(TForm)
    Cliente: TIBQuery;
    •••
    botãoGravar: TButton;
    •••
    procedure botãoGravarClick(Sender: TObject);
    procedure ClienteAfterPost(DataSet: TDataSet);
  private
    { Private declarations }
  public
    { Public declarations }
  end;

implementation

uses
  UnitConexao, •••

procedure TFormCliente.botãoGravarClick(Sender: TObject);
begin
  Cliente.Post;   // Atualiza o buffer de memória do computador: Cliente
end;

procedure TFormCliente.ClienteAfterPost(DataSet: TDataSet);
begin
  //
  // Executa um Refresh na Tabela CLIENTE
  //
  if Conexao.TransacaoGEST_ADM.InTransaction then
  begin
    try                                              // Atualiza fisicamente,
      Conexao.TransacaoGEST_ADM.CommitRetaining;  // computador: Servidor
    except
      ShowMessage('Outro Usuário Alterou ou Excluiu Esse Registro,
                   Pressione OK para Receber os Dados Atualizados do
                   Servidor' );
      Conexao.TransacaoGEST_ADM.RollbackRetaining;
    end;
  end;
end;

end.
```

Figura 2: Código Delphi que implementa as atualizações Cliente/Servidor como Transações

Problema:

Existem diversas telas de acesso às funções do sistema legado, desenvolvidas para com interface baseada em caracter, totalmente peculiar às aplicações implementadas em Clipper, Cobol, entre outras. É, pois, necessário conceber uma nova interface, baseada em gráfico (GUI), que atenda também aos critérios de usabilidade, e que possa ser construída na linguagem de implementação escolhida para o processo de engenharia avante.

Solução:

Para cada tela do ambiente caracter, do sistema legado, gerar conjunto de interfaces na linguagem de implementação da reengenharia, que realizem melhorias por meio dos conceitos de usabilidade. Utilizar para a implementação as ferramentas de suporte que acompanham o ambiente de desenvolvimento escolhido.

Exemplo:

Apresenta-se na Figura 3 a tela do sistema legado Clipper quando de sua apresentação. Aplicando-se os conceitos de usabilidade, a Figura 4 exibe a tela de apresentação do sistema, após passar pelo processo de engenharia avante, utilizando a Família de Padrões de Reengenharia Orientada a Objetos.

Padrões Relacionados:

Pode-se considerar a aplicação, em paralelo, de *A Pattern Language for Developing Graphical User Interfaces* [5], e de *Interaction Design Patterns* [18], para considerar aspectos não abordados pelo padrão Realizar Melhorias na Interface.

Figura 3: Apresentação do Sistema Legado SIGO

Figura 4: Apresentação do Sistema SIGO após o Processo de Engenharia Avante

3. Comentários Finais

A realização do processo de reengenharia de sistemas legados é considerada como um desafio para os engenheiros de software, pois esse processo envolve muitos fatores de risco. Há, então, interesse em tornar os engenheiros de software especialistas nesse processo. Para isso, surgem os padrões de engenharia reversa e de engenharia avante com o objetivo de registrar as técnicas e mecanismos que os engenheiros de software experientes utilizam para conduzir esses processos.

Este trabalho apresentou os padrões do Processo de Engenharia Avante da FaPRE/OO, ou seja, padrões que auxiliam a realização da engenharia avante orientada a objetos de sistemas legados desenvolvidos de forma procedimental e implementados em linguagens como Clipper, Cobol, RPG II.

Um sistema, originalmente desenvolvido de forma procedimental e implementado na linguagem Clipper [14,15,16], foi submetido ao processo de reengenharia seguindo, passo a passo, todos os padrões conforme proposto pela FaPRE/OO. Assim, é realizada a engenharia reversa procedimental do sistema legado e, a partir dos resultados dessa atividade, efetua-se a engenharia reversa orientada a objetos do legado. Em outras palavras, na primeira fase obtém-se uma documentação procedimental e, na segunda fase, com base na anterior, constrói-se a documentação de análise orientada a objetos. A partir dos produtos da engenharia reversa o processo de engenharia avante é aplicado.

Outro sistema, originalmente desenvolvido de forma procedimental e implementado na linguagem Cobol, foi submetido ao processo de reengenharia usando esta Família [3]. Nesse

trabalho a engenharia reversa foi realizada diretamente, isto é, sem a necessidade do produto intermediário. A documentação de análise orientada a objetos foi obtida diretamente do código procedimental a fim de identificar possíveis objetos. Assim, a FaPRE/OO dá plena cobertura para conduzir a engenharia reversa diretamente orientada a objetos a partir do sistema legado procedimental, sendo a ordem de aplicação dos vários padrões da Família, o diferencial das duas formas em que foram aplicados. Como o modelo do processo é evolutivo, isso fortalece o seu potencial em questão do domínio de sistemas de informação. A partir dos produtos da engenharia reversa, o processo de engenharia avante é aplicado.

Além disso, um outro sistema, desenvolvido na linguagem Delphi [9], foi submetido, com sucesso, ao processo de reengenharia seguindo, passo a passo, os padrões propostos na FaPRE/OO. Embora o ambiente de desenvolvimento deste trabalho tivesse sido Delphi, o qual viabiliza a construção de sistemas orientados a objetos, o sistema envolvido em um estudo de caso foi originalmente implementado sem os conceitos da orientação a objetos.

A geração do código fonte orientado a objetos, quando da aplicação do padrão Implementar os Métodos é realizada de forma tradicional, ou seja, não automatizada. O sistema transformacional utilizado por Jesus et al. [7], não pôde ser aqui utilizado, tendo em vista a forma como o Draco-Puc [13] converte um sistema legado Clipper para OO: Tranforma todo o conteúdo de Clipper para Java. Programas procedimentais desenvolvidos em Clipper que manipulassem arquivos de dados (.dbf's) com milhares de registros, tornavam-se extremamente lentos quando esses arquivos fossem acessados. Para melhorar o desempenho do sistema utilizava-se de artifícios de programação, de tal forma que eram criados arquivos secundários de apoio (persistentes e/ou temporários). Dessa forma, como o Draco-Puc [13] converte integralmente o codigo legado Clipper em Java, as centenas/milhares de linhas de código de manipulação aos arquivos secundários, seriam também incluídas no código Java, bem como, os arquivos secundários seriam transformados em classes.

4. Agradecimentos

Agradecemos ao Shepherd Vander Ramos Alves pelas sugestões e acompanhamento dado a este trabalho.

5. Referências Bibliográficas

[1] Borland Brasil, 2002. URL:http://www.borland.com.br/artigos/artigo_ccantu.htm. Consultado em 03/2002.

[2] Camargo, V., **"Reengenharia Orientada a Objetos de Sistemas COBOL com a Utilização de Padrões de Projetos e Servlets"**, São Carlos-SP, 2001. Dissertação de Mestrado. Universidade Federal de São Carlos.

[3] Camargo, V.; Recchia, E. L.; Penteado, R. – **"Aplicabilidade da Família de Padrões de Reengenharia FaPRE/OO na Engenharia Reversa Orientada a Objetos de Sistemas Legados COBOL"**, Artigo apresentado no The Second Latin American Conference on Pattern Languages of Programming – Software Pattern Applications. (SugarLoafPLoP–SPA), Itaipava-RJ, Agosto/2002.

[4] Demeyer, S.; Ducasse, S.; Nierstrasz, O., **"A Pattern Language for Reverse Engineering".** Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000.

[5] Deugo, D.; Sandu, D. **"A Pattern Language for Developing Graphical User Interfaces"**. EuroPlop 99, 8-10 Julho, 1999, Germany, http://www.argo.be/europlop/writers.htm.

[6] Dewar, R.; Lloyd, A.D.; Pooley, R.; Stevens, P. **"Identifying and Communicating Expertise in Systems Reengineering: a patterns approach".** IEEE Proceedings – Software, v.146, n.3, pp.145-152, 1999.

[7] Jesus, E.; Prado, A.F. **- "Reengenharia de Software para Plataformas Distribuídas Orientadas a Objetos"**, XIII Simpósio Brasileiro de Engenharia de Software - SBES'99, pg. 289-304.

[8] Kulk, E.; Camargo, V. V.; Masiero, P. C.; Penteado, R.; Germano, F., **"Reengenharia Orientada a Objetos de um Sistema Contábil Implementado em Cobol para Java"**, Documento de Trabalho, 2002. Universidade de São Paulo – SP.

[9] Lemos, G. S., **"PRE/OO - Um Processo de Reengenharia com Ênfase na Garantia da Qualidade"**, São Carlos-SP. Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos, em Agosto/2002.

[10] Penteado, R. A. D., **"Um Método para Engenharia Reversa Orientada a Objetos",** São Carlos, 1996. 237 p. Tese (Doutorado em Física Computacional) - Instituto de Física de São Carlos, Universidade de São Paulo.

[11] Penteado, R., Germano, F., Masiero, P. C., **"An Overall Process Based on Fusion to Reverse Engineering Legacy Code",** In: Working Conference Reverse Engineering, 3, 1996, Monterey-California. Anais. IEEE, p. 179-188.

[12] Penteado, R., Braga, R.T.V., Masiero, P.C., **"Improving the Quality Legacy Code by Reverse Engineering",** In: 4th International Conference on Information Systems Analysis and Synthesis, ISAS/98, Julho/1998, Orlando-Flórida.

[13] Prado, A. F., **"Estratégias de Engenharia de Software Orientada a Domínios"**, Tese de Doutorado, Rio de Janeiro - RJ, 1992. Pontificia Universidade Católica, 333p.

[14] Recchia, E. L., **"Engenharia Reversa e Reengenharia Baseadas em Padrões"**, São Carlos-SP. Dissertação de Mestrado apresentada ao PPGCC-DC. Universidade Federal de São Carlos, em Junho/2002.

[15] Recchia, E. L.; Penteado, R. – **"FaPRE/OO: Uma Família de Padrões para Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais"**, Artigo apresentado no The Second Latin American Conference on Pattern Languages of Programming. (SugarLoafPLoP– Writers' Workshop), Itaipava-RJ, Agosto/2002.

[16] Recchia, E. L.; Penteado, R. – **Avaliação da Aplicabilidade da Linguagem de Padrões de Engenharia Reversa de Demeyer a Sistemas Legados Procedimentais**, Artigo apresentado no The Second Latin American Conference on Pattern Languages of Programming – Software Pattern Applications. (SugarLoafPLoP–SPA), Itaipava-RJ, Agosto/2002.

[17] Stevens, P.; Pooley, R. - **"Systems Reengineering Patterns"**, Proceedings ACM-SIGSOFT, 6th Internation Symposium on the Foundations of Software Engineering, p.17-23, 1998.

[18] Tidwell, J. - **"Interaction Design Patterns"**, Conference on the Pattern Languages of Programs. PLoP'1998.

[19] Yoder, J.W.; Barcalow, J. - **"Architetural Patterns for Enabling Application Secutity"**, Conference on the Pattern Languages of Programs. PLoP'1997.

[20] Yoder, J.W.; Johnson, R.E; Wilson, Q.D. - **"Connecting Business Objects to Relational Databases"**, Conference on the Pattern Languages of Programs. PLoP'1998.

[21] http://www.openloop.com/softwareEngineering/patterns/architecturePattern/arch_Layers.htm Consultado em 05/2003.

Multi Locale Entity: Um padrão de projeto para persistência de entidades com internacionalização

Carlo Giovano S. Pires [1]
cgiovano@atlantico.com.br
Instituto Atlântico

Resumo

A ampla adoção de aplicações Web e distribuídas requer processos para internacionalização e localização que permitam que as aplicações possam dar suporte à diferentes línguas e regiões. Esse trabalho apresenta um padrão de projeto (padrão Multi Locale Entity) que fornece suporte para internacionalização e localização. O padrão apresenta uma solução para a camada de negócio, camada de integração e camada de dados.

Abstract

The large adoption of Web and distributed application requires processes of internationalization and localization in order to enable applications to support different languages and regions. This paper presents a design pattern (Multi Locale Entity pattern) that supports internationalization and localization. It presents a solution for the business tier, the integration tier and the data tier.

Introdução

A larga difusão de aplicações Web e distribuídas exige cada vez mais o suporte de internacionalização e localização de aplicações. Internacionalização [4] é o processo de projetar uma aplicação para se adaptar a várias línguas e regiões. Localização [4] é o processo de adaptar uma aplicação para uma língua e região.

A internacionalização de uma aplicação pode ser feita ao nível de elementos da interface e dos próprios dados informados. Apesar da existência de outros modelos de dados (modelo orientado a objetos [1], hierárquico, entre outros), o modelo relacional representa a grande maioria das aplicações com acesso à base de dados. O padrão proposto tem o objetivo de fornecer um mecanismo de representação e persistência para entidades com suporte a internacionalização, facilitando a localização de aplicações e otimizando o acesso aos dados.

[1] Este trabalho foi suportado pelo Instituto Atlântico (*www.atlantico.com.br*).

1 Contexto

Desenvolvimento de aplicações Web e distribuídas (N-Camadas como, por exemplo, CORBA ou J2EE) com bancos de dados relacionais que devem ser utilizadas em diversas regiões devem dar suporte aos processos de localização e internacionalização dos dados. Nesse tipo de aplicações, uma mesma entidade (instância, ou *pool* de instâncias) é compartilhada por diversos clientes. Cada cliente pode estar em alguma região do mundo e necessitar consultar e editar a entidade na sua própria língua.

Quanto aos dados, as tabelas em um banco relacional e as classes de negócio e persistência devem ser projetadas de forma a racionalizar o suporte a várias línguas.

2 Problema

Como desenvolver objetos para representação de entidades de negócio, a camada de acesso a dados e o modelo de dados relacional de forma a dar suporte à persistência de dados em várias línguas?

3 Forças

- O acesso à informação sensível a língua (localizada) deve ser simples para fornecer boa usabilidade para as classes localizadas;
- O processo de localização (por exemplo, adicionar novas línguas à aplicação e atualizar informações sensíveis à língua) deve ser feito sem recompilação da aplicação, oferecendo manutenibilidade à aplicação;
- O mecanismo de acesso aos dados deve minimizar as consultas à informação para cada língua de forma a não prejudicar o desempenho da aplicação;
- O mecanismo de acesso a dados deve estar separado da entidade de negócio para oferecer extensibilidade com relação ao modelo de dados e tecnologia de acesso ao repositório.

4 Solução

O padrão fornece uma solução que abrange a camada de negócios, camada de integração e a camada de dados. A solução para camada de negócio e integração mostra como projetar classes para representar a entidade que deve ser localizada e como projetar uma classe para armazenar e recuperar a informação dessa entidade em uma base de dados. A solução para a camada de dados mostra como projetar tabelas em um banco de dados relacional para armazenar informações de entidades com suporte a internacionalização.

Quanto às camadas de negócio, o padrão fornece a classe *Language* para representação da língua, as classes *MultiLocaleEntity* e *LocaleSensitiveValue* para representar a entidade internacionalizada com informação não dependente de língua e com informação dependente de língua. A classe *MultiLocaleDataAccess* é utilizada na camada de integração para gerenciar a persistência da informação. A classe de entidade *MultLocaleEntity* deve fornecer um método para leitura de cada atributo sensível a língua. Esse método recebe como parâmetro o objeto que representa a língua e retorna a informação localizada. O objeto de persistência realiza a interface com a base de dados, preenchendo a informação da entidade para cada língua, atualizando, criando e excluindo a informação quando necessário.

Quanto à camada de acesso a dados, deve-se fornecer uma tabela com a informação que não é sensível à língua e possui o ID da entidade, e uma outra que é o relacionamento entre essa primeira e a tabela de línguas. A tabela de relacionamento deve ter um campo para cada atributo sensível à língua.

5 Estrutura

A figura 1 apresenta o diagrama de classes da camada de aplicação do padrão *Entidade Multilíngua*. Uma descrição de cada participante é apresentada na seção *Participantes*.

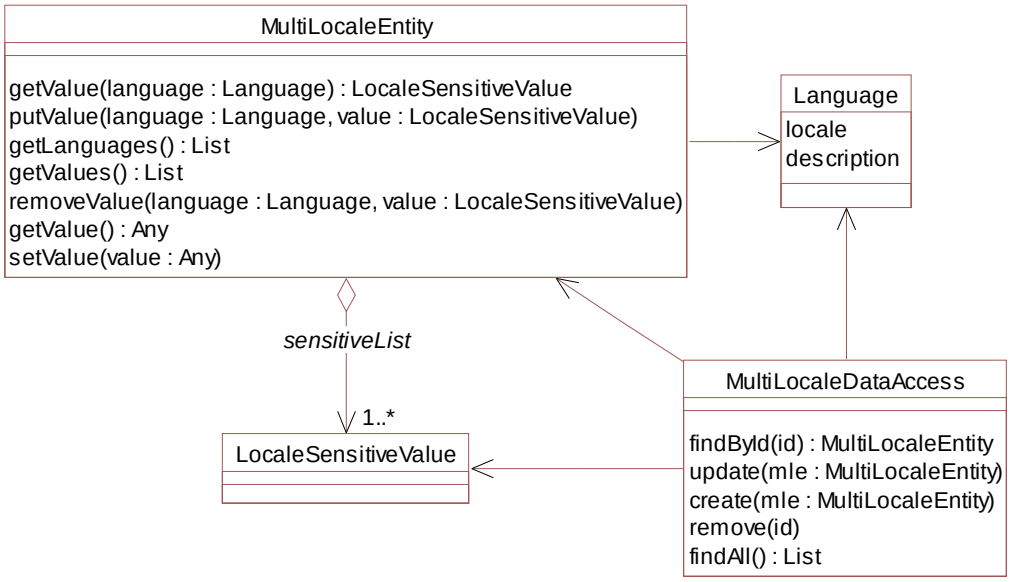


Figura 1 – Diagrama de Classes do Padrão Multi Locale Entity na camada de negócio e integração

A figura 2 apresenta o esquema da camada de dados do padrão *Entidade Multilíngua*. Uma descrição de cada participante é apresentada na seção *Participantes*.

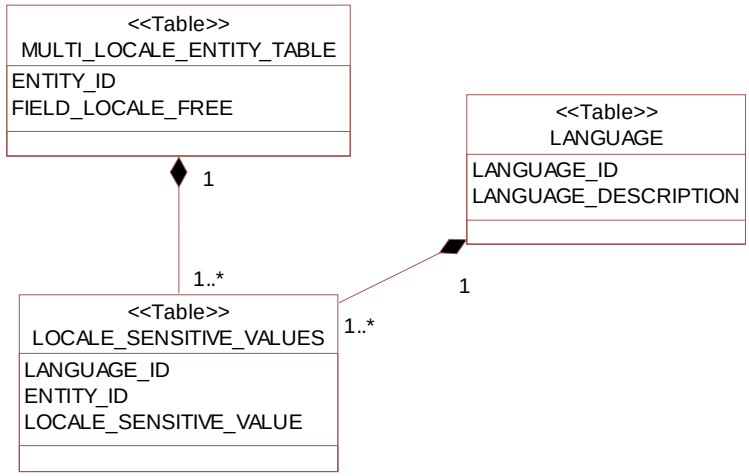


Figura 2 – Esquema do Padrão Multi Locale Entity na camada de dados

6 Participantes

6.1 Camada de Negócio

- **MultiLocaleEntity**
  - Fornece operações para cada atributo para consultar a informação (*getValue*) e outro para atualizar a informação (*putValue*). As operações permitem a passagem da língua como parâmetro
  - Possui uma lista de objetos *LocaleSensitiveValue* que podem ser indexados pela língua (Language)
  - Fornece operação (*removeValues*) para remover um elemento de acordo com a língua
  - Fornece operação (*getLanguages*) para listar as línguas suportadas pela entidade/aplicação

- **LocaleSensitiveValue**
  - Representa o valor dependente da língua
  - Pode ser uma classe complexa (por exemplo, classe que representa o cálculo de imposto dependendo do país) ou tipos primitivos como caracteres e números (por exemplo, descrições e valores monetários)

- **Language**
  - Representa a língua
  - Possui atributo (*locale*) para identificar a língua. Esse atributo identifica a língua propriamente dita e a região. Por exemplo, en_US, representa inglês dos Estados Unidos
  - Possui atributo (*description*) para descrever a língua. Por exemplo, Inglês

6.2 Camada de Integração

- **MultiLocaleDataAccess**
  - Implementa o mecanismo de persistência da entidade sensível à língua
  - Define instruções SQL [2] que fazem o mapeamento da entidade em tabelas com suporte a várias línguas

6.3 Camada de Dados

- **Multi_Locale_Entity_Table**
  - Tabela que possui como chave primária o atributo identificador da entidade (*Entity_Id*) e os atributos que não dependem de língua (*Field_Locale_Free*)
  - Funciona como entidade forte para a tabela fraca *Locale_Sensitive_Values*

- **Locale_Sensitive_Values**
  - Tabela que possui como chave primária o identificador da entidade (*Entity_Id*) e o identificador da língua (*Language_Id*), permitindo armazenar valores diferentes, para línguas diferentes em uma mesma entidade
  - Possui atributos para armazenar dados que dependem da língua (*Locale_Sensitive_Value*)

- **Language**

  - Tabela que armazena as línguas suportadas pela aplicação
  - Possui como chave primária o identificador da língua (*language_id*) e possui um atributo com a descrição da língua (*Language_Description*)

7 Dinâmica

Veja fluxo da consulta para o primeiro acesso de uma entidade multilíngua (ver figura 3):

1. Cliente consulta entidade por seu identificador (*Id*)
2. *MultiLocaleDataAccess* constrói novo objeto *MultiLocaleEntity* e executa consulta por *Id*, recuperando os dados da tabela independente de língua e da tabela sensível a língua. *Para cada linha recuperada*
   2.1 Preenche atributo não-sensível à língua em *MultiLocaleEntity*
   2.2 Adiciona à lista *sensitiveList* o objeto sensível à lista, passando *Language* como parâmetro (*Language* é construído com base no id da língua armazenada na tabela sensível a língua - *LocaleSensitiveValues*)
3. *MultiLocaleDataAccess* retorna objeto *MultiLocaleEntity*
4. Cliente obtém língua do usuário através de algum contexto e obtém valor de um atributo sensível à língua através do método *getValue(contextLanguage)*. Atributos não sensíveis à língua são recuperados através de algum método do tipo *getValue()*
5. O cliente atualiza algum atributo sensível à língua através de algum método *putValue(contextLanguage, newLocaleSensitiveValue)*, passando o novo valor e a língua selecionada como parâmetro
6. O cliente confirma atualização do objeto através do objeto *MultiLocaleEntityDataAccess*
7. *MultiLocaleEntity* executa (dentro de uma única transação)
   7.1 Instrução de atualização em *Multi_Locale_Entity_Table* para os atributos não sensíveis à língua modificados
   7.2 Obtém lista de línguas disponíveis em *MultiLocaleEntity* através do método *getLanguages*
   7.3 Para cada língua da lista, recupera objeto modificado através de *getValue* de *MultiLocaleEntity* e executa instrução de atualização na tabela *Locale_Sensitive_Values*

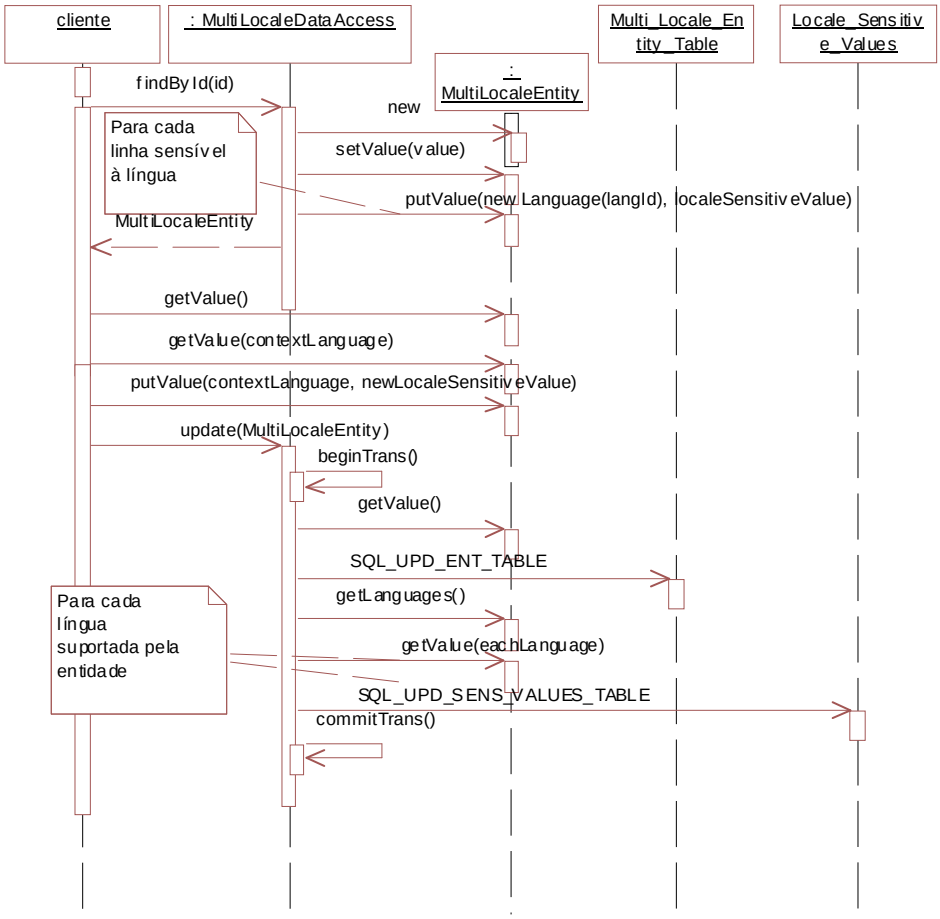


Figura 3: Consulta e atualização de entidade

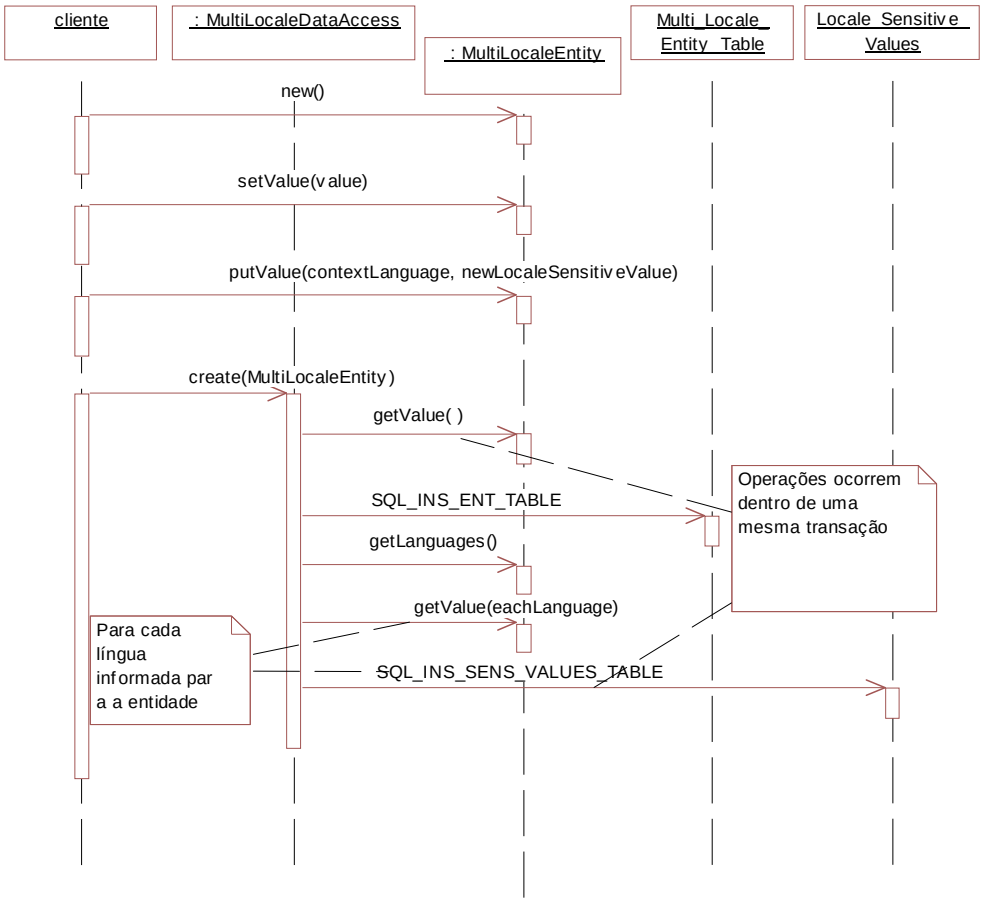


Figura 4: Criação de entidade

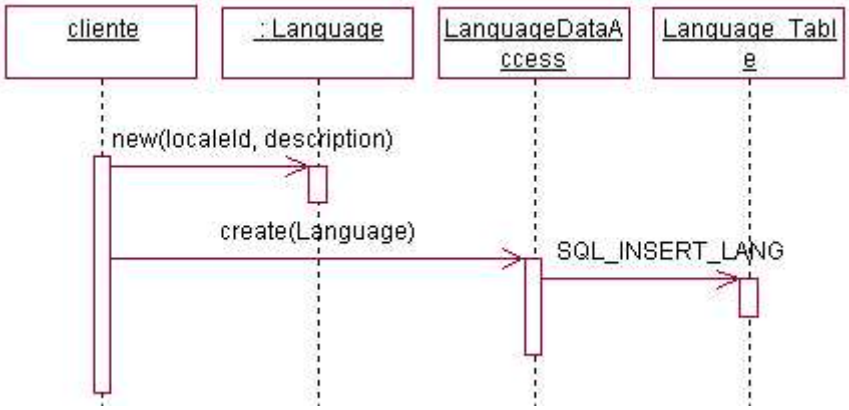


Figura 5: Adicionando uma nova língua

As figuras 4 e 5 apresentam a criação de uma nova entidade e a adição de uma nova língua à aplicação.

8 Conseqüências

8.1 Vantagens

- A classe que representa a entidade (*MultiLocaleEntity*) fornece uma interface simples e bem definida para acessar a informação/atributos de acordo com a língua do contexto da aplicação, facilitando o acesso a informações localizadas e aumentando a usabilidade.
- A configuração de novas línguas pode ser feita via classe *Language* e novas informações (ou atualizações) referentes a uma língua podem ser registradas através da interface *putValue* da classe *MultiLocaleEntity* sem recompilação da aplicação, oferecendo boa manutenibilidade para a aplicação
- O acesso a dados é minimizado através da recuperação e *cache* das informações para cada língua no objeto *MultiLocaleEntity* melhorando o desempenho das operações de localização da aplicação
- A classe *MultiLocaleEntityDataAccess* isola a entidade de negócio do modelo e tecnologia de acesso a dados, fornecendo extensibilidade com relação ao modelo de dados e tecnologia de acesso.
- A separação de tabelas do modelo relacional entre informações sensíveis à língua e informações insensíveis à língua fornece maior racionalização do armazenamento das informações

8.2 Desvantagens

- O *cache* das informações por língua pode gerar maior utilização de memória
- Se os dados da aplicação forem atualizados por outros meios diferentes das classes do padrão, as informações em *cache* podem ficar desatualizadas e inconsistentes com a base de dados
- O padrão gera um número maior de classes e pode gerar maior complexidade para entendimento

9 Variantes

Apesar do padrão dar ênfase ao modelo relacional, o padrão também pode ser utilizado para outros modelos de dados, basta utilizar outras implementações de *MultiLocaleDataAccess* para cada modelo, utilizando instruções de consulta equivalentes.

10 Exemplo

O exemplo exibe parte do código utilizado para implementar uma entidade sensível à língua para um portal na *Web*. Essa entidade representa os tipos de conteúdos que podem ser apresentados no portal. Os dados da entidade são utilizados na página principal do portal para que o usuário possa selecionar qual tipo de conteúdo deseja ver: Esportes, Notícias, Diversões, Tecnologia, Cinema, etc. O portal fornece uma aplicação de *backoffice* que permite que administradores possam criar, remover, atualizar, habilitar e desabilitar tipos de conteúdo.

A entidade possui os seguintes atributos que não dependem de língua: i*d, enabled* e *totalHits*. O atributo *id* é o identificador único do tipo de conteúdo, o atributo *enabled* indica que um tipo de conteúdo pode ser habilitado ou desabilitado. Conteúdos desabilidades não

aparecem no portal. Outro atributo que não depende da língua é *totalHits*. Esse atributo indica o número total de acessos a determinado tipo de conteúdo, independemente da língua ou país.

A entidade também possui os seguintes atributos que são sensíveis à língua: *description* e *priority*. O atributo *description* indica a descrição do tipo de conteúdo em cada língua suportada pelo portal. Dessa forma, o usuário que acessar o portal pode ver os tipos de conteúdo disponíveis na língua mais adequada. O atributo *priority* é um objeto que descreve a ordem (*order*) e cor (*color*) que podem caracterizar um tipo de conteúdo dependendo da língua do usuário. Dessa forma, o administrador do portal pode indicar, por exemplo, que um usuário que seleciona a língua *Português/Brasil* veja o conteúdo Diversão no topo da página e em vermelho, enquanto um usuário com língua *Japonês/Japão* veja Tecnologia no topo com cor vermelha e Diversão no meio da lista e com cor azul.

Na aplicação do padrão *Multi Locale Entity* para a entidade de tipo de conteúddo, temos as classes *ContentType, Priority, ContentTypeDataAccess*. A primeira funciona como *MultiLocaleEntity,* a segunda como *LocaleSensitiveValue* e a terecenira como *MuliLocaleEntityDataAccess*. Para o atributo *description* de *ContentType,* o *LocaleSensitiveValue* é apenas o tipo *String*. Veja na figura 6 que na classe *ContentType* temos as operações *getDescription(language : Language)* e *getPriority(language : Language)* como aplicações da operação *MultiLocaleEntity::getValue(language : Language)* do padrão. Também temos as operações *putDescription(language : Language, value: String)* e *putPriority(language : Language, value: Priority)* como aplicações da operação *MultiLocaleEntity::putValue(language : Language, value : LocaleSensitiveValue)* do padrão. Finalmente temos as operações *removeDescription(language : Language, value: String)* e *removePriority(language : Language, value: Priority)* como aplicações da operação *MultiLocaleEntity::removeValue(language : Language, value : LocaleSensitiveValue)* do padrão.

Para os atributos que não dependem de língua, temos *getEnabled()* e *setEnabled(e : boolean), getTotalHits()* e *setTotalHists(h : Integer)* como aplicações de *MultiLocaleEntity::getValue()* e *MultiLocaleEntity::setValue(v: Any)* do padrão.

As operações *getDescriptions()* e *getPriorities()* são aplicações de *MultiLocaleEntity::getValues().*

As classes do exemplo foram implementadas em *Java*.

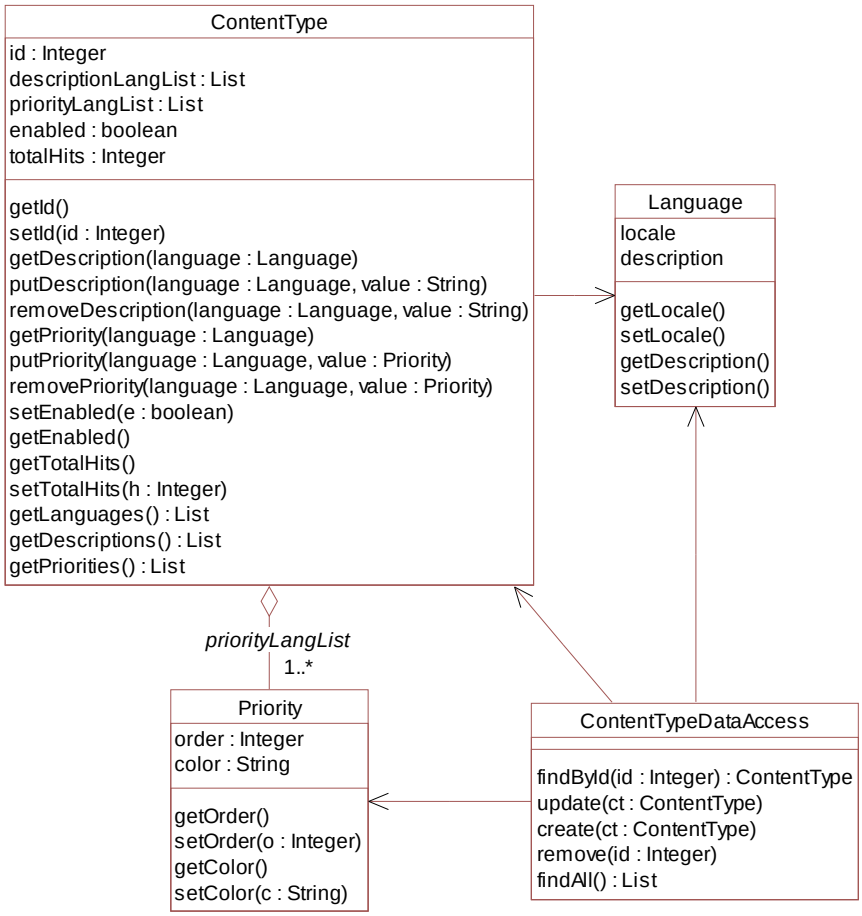


Figura 6: Diagrama de classes para entidade ContentType

As tabelas criadas foram CONTENT_TYPE, CONTENT_TYPE_LOCALE_SENSITIVE_VALUES representado, respectivamente, MULTI_LOCALE_ENTITY_TABLE e LOCALE_SENSITIVE_VALUES. As classes para *Language* e a tabela LANGUAGE também foram criadas. Veja na figura 7 que apenas os campos que não dependem de língua estão na tabela CONTENT_TYPE, os campos sensíveis à língua estão na tabela CONTENT_TYPE_LOCALE_SENSITIVE_VALUES juntamente com a chave primária composta por CONTENT_TYPE_ID e LANGUAGE_ID.

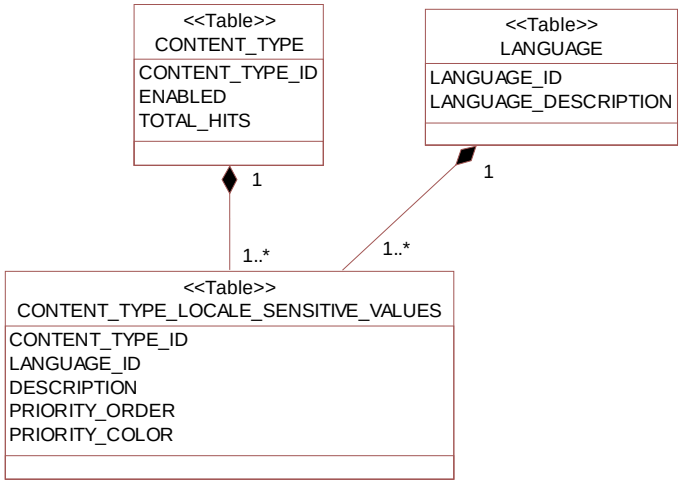


Figura 7: Esquema de dados para entidade ContentType

O trecho de código na Figura 8 apresenta variáveis da classe *ContentTypeDataAccess* que representam as consultas SQL para armazenamento e recuperação da entidade na base de dados:

```
private static final String SQL_FIND_BY_ID = "SELECT CONTENT_TYPE.CONTENT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE.TOTAL_HITS,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.LANGUAGE_ID,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.DESCRIPTION,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_ORDER,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_COLOR FROM CONTENT_TYPE,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONT_TYPE_ID=? AND
CONTENT_TYPE.CONT_TYPE_ID = CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.CONT_TYPE_ID ";

private static final String SQL_FIND_ALL = "SELECT CONTENT_TYPE.CONTENT_TYPE_ID,
CONTENT_TYPE.ENABLED, CONTENT_TYPE.TOTAL_HITS,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.LANGUAGE_ID,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.DESCRIPTION,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_ORDER,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.PRIORITY_COLOR FROM CONTENT_TYPE,
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONTENT_TYPE.CONT_TYPE_ID =
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES.CONT_TYPE_ID ";

private static final String SQL_CREATE_ENT = "INSERT INTO CONTENT_TYPE
(CONT_TYPE_ID, ENABLED, TOTAL_HITS) VALUES (?, ?, ?)";

private static final String SQL_CREATE_LSENS = "INSERT INTO
(CONTENT_TYPE_LOCALE_SENSITIVE_VALUES CONTENT_TYPE_ID, LANGUAGE_ID,DESCRIPTION,
PRIORITY_ORDER, PRIORITY_COLOR) VALUES (?,?,?,?,?)";

private static final String SQL_DELETE_ENT = "DELETE FROM CONTENT_TYPE WHERE
CONTENT_TYPE_ID=?";

private static final String SQL_DELETE_LSENS = "DELETE FROM
CONTENT_TYPE_LOCALE_SENSITIVE_VALUES WHERE CONTENT_TYPE_ID=?";
```

Figura 8 – Consultas de ContentTypeDataAccess

O trecho de código na Figura 9 apresenta o código para recuperar uma lista de todos os tipos de conteúdo. Veja que logo abaixo do comentário "//Verifica se já não está na lista", uma condição é testada para evitar que a instância de *ContentType* seja adicionada à lista, caso ela já tiver sido adicionada. Isso evita que as repetições de informações não sensíveis à língua gerada pela junção das duas tabelas no SQL_FIND_ALL sejam inseridas na lista. Logo

após a condição, o método *putDescription* de *ContentType* é acionado, colocando as informações sensíveis à língua, no caso a descrição do tipo de conteúdo. Depois que todas as informações sensíveis à língua para um mesmo *id* são colocadas na instância, uma nova instância, para novo *id* é criada e o processo se repete.

```
public List findAll() throws DataAccessException {

        Connection conn = null;
        try {
            conn = getDBConnection();
            Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(SQL_FIND_ALL);
            List list = new ArrayList();
            ContentType contentType = null;
            String id;
            while (rs.next()) {
                id = rs.getString("CONT_TYPE_ID");
                //Verifica se já não está na lista
                if (list.size()==0 || !((contentType)list.get(list.size()- 1
                    )).getId().equals(id)) {
                   contentType = new ContentType(id);
                   contentType.setEnabled(rs.getString("ENABLED").equals("YES"));
                   contentType.setTotalHits(rs.getString("TOTAL_HITS"));

                   list.add(ContentType);
                }

                Priority priority = new Priority();
                Priority.setOrder(rs.getInteger("PRIORITY_ORDER"));
                priority.setColor(rs.getString("PRIORITY_COLOR"));

                contentType.setPriority(priority);
               contentType.putDescription(LocaleFormat.parse(rs.getString("LANGUAG
               E_ID")),rs.getString("DESCRIPTION"));
            }
            rs.close();
            stmt.close();
            return list;
        }
        catch (Exception ex) {
            throw DataAccessException.buildException(ex);
        }
        finally {
            closeDBConnection(conn);
        }

    }
```

Figura 9 – Operação findAll de ContentTypeDataAccess

O trecho de código na Figura 10 apresenta o código para inserir um novo tipo de conteúdo no portal. Note que o primeiro SQL (SQL_CREATE_ENT) é executado uma única vez para criar a nova entidade e o segundo SQL (SQL_CREATE_LSENS) é executado para todas as línguas informadas (*contentType.getLanguages*) para a nova entidade, inserindo várias linhas na tabela, para cada novo *id* da entidade sensível à língua. Todas as operações são feitas dentro de uma única transação.

```
public Integer create(ContentType contentType) throws DataAccessException {

        Connection conn = null;
        try {
            conn = getDBConnection();
            int id = getNextId();
            conn.setAutoCommit(false);
            try {
                 PreparedStatement pstmt = conn.prepareStatement(SQL_CREATE_ENT);
                pstmt.setInteger(1,id);
                pstmt.setString(2, (contentType.isEnable()?"YES":"NO"));
                pstmt.setInteger(3, (contentType.totalHits()));

                pstmt.executeUpdate();
                pstmt.close();

                pstmt = conn.prepareStatement(SQL_CREATE_LSENS);

                pstmt.setInt(1,id);

                Locale[] locales = contentType.getLanguages();
                for (int index=0; index<locales.length; index++) {
                    pstmt.setString(2,LocaleFormat.format(locales[index]));
                    pstmt.setString(3,contentType.getDescription(locales[index]));
                   pstmt.setInteger(4,contentType.
                               getPriority(locales[index]).getOrder());
                    pstmt.setString(5,contentType.
                               getPriority(locales[index]).getColor());
                    pstmt.executeUpdate();
                }
                pstmt.close();
                conn.commit();
            }
            catch (SQLException ex) {
                conn.rollback();
                throw new DataAccessException(ex);
            }

            contentType.setId(id);
            return id;
        }
        catch (Exception ex) {
            throw DataAccessException.buildException(ex);
        }
        finally {
            closeDBConnection(conn);
        }

}
```

Figura 10 – Operação create de ContentTypeDataAccess

11 Usos Conhecidos

Por motivos de confidencialidade, mais detalhes dos usos conhecidos abaixo não podem ser fornecidos.

- Um portal Web de acesso a informações de lista telefônica
- Um portal Web de informações sobre restaurantes e gastronomia
- Uma aplicação de Web-TV
- Uma aplicação de correio eletrônico segura na Web

12 Padrões Relacionados

- *Data Access Object* [3]:
  - *Pode ser utilizado para implementação do MultiLocaleDataAccess*
- *ValueObject* [3]:
  - Um *ValueObject* pode representar uma entidade sensível à língua e que deve ser transmitida via rede

Referências

[1] R. Cattel, D. Barry, editors. *The Object Database Standard: ODMG 93*. Morgan Kaufman, 1997.

[2] ISO/IEC JTC1/SC21 N10489, ISO//IEC 9075, Part 2, Committee Draft (CD), *Database Language SQL - Part 2: SQL/Foundation*, ftp://speckle.ncsl.nist.gov/isowg3/dbl/BASEdocs/cd-found.pdf, July, 1996.

[3] http://java.sun.com/blueprints/corej2eepatterns/Patterns/index.html

[4] http://l10n.openoffice.org/

TB-REPP - Padrões de Processo para a Engenharia Reversa baseado em Transformações [*]

Darley Rosa Peres, Alexandre Álvaro, Valdirene Fontanette, Vinicius C. Garcia[♠], Antonio Francisco do Prado
Depto. de Computação, Univ Federal de São Carlos
{darley, aalvaro, valdirene,vinicius,prado}@dc.ufscar.br

Rosana T. V. Braga
ICMC/USP, Univ de São Paulo
rtvb@icmc.usp.br

Resumo

Engenharia Reversa é uma técnica usada para recuperar a informação dos documentos do software e do seu código fonte, visando obter a sua representação em um alto nível de abstração. Deste modo, fica mais fácil a compreensão do sistema.
Considerando que no processo de Reengenharia a Engenharia Reversa é a fase mais problemática, esse artigo propõe um grupo de Padrões de Processo para a Engenharia Reversa Baseada em Transformações (TB-REPP), para obter o projeto do sistema em um alto nível de abstração, assegurando a sua evolução e manutenção, tornando-o mais expressivo e de fácil compreensão.

Abstract

Reverse Engineering is a technique used to recover information from software documents and even from source code, aiming to obtain its representation in a higher abstraction level. In that way, it can make easier the system understanding.
Considering that in the Re-engineering process, Reverse Engineering is the most problematic phase, this paper proposes a Transformation-Based Reverse Engineering Process Patterns group (TB-REPP), to obtain the system project in a high abstraction level, assuring its evolution and maintenance, making it more expressive and easy to understand.

1. Introdução

Sistema de software é um artefato evolutivo e requer constantes modificações, seja para corrigir erros, melhorar desempenho, adicionar novas funcionalidades ou até mesmo para adaptá-lo às novas tecnologias de desenvolvimento de software.

A dificuldade em atualizar esses sistemas de software tem motivado pesquisadores a encontrar novas soluções para facilitar e reduzir os custos de sua manutenção, pois a falta de

[♠] Financiado pela Fundação de Amparo à Pesquisa do Estado da Bahia (Fapesb).

uma documentação consistente que permita um melhor entendimento do domínio da aplicação é constante na grande maioria dos sistemas.

Atualmente, existe um grande número de empresas que continuam trabalhando com sistemas implementados em linguagens de programação antigas já em desuso, cujas manutenções são árduas e custosas. Esses sistemas, denominados legados, ainda são de muita utilidade aos seus usuários e muitas vezes, sua reconstrução, usando técnicas modernas de desenvolvimento de software, podem ser a solução para sua reutilização sem ter que construir um novo sistema. Normalmente, os sistemas legados não possuem documentação ou quando possuem, a mesma está desatualizada, devido às inúmeras alterações e adaptações neles implementadas, e nem sempre a pessoa que desenvolveu o sistema é a responsável pela sua manutenção. Fatores como estes tornam difícil e de alto custo a manutenção, fazendo com que os desenvolvedores até abandonem o sistema.

Num processo de Reengenharia, a fase mais problemática fica a cargo da Engenharia Reversa, pois a obtenção de uma documentação e da recuperação de um projeto orientado a objetos partindo de um código legado é mais difícil do que aplicar a Engenharia Avante com o projeto já recuperado e todo documentado. Levando em consideração essa dificuldade, é proposto um conjunto de Padrões de Processo para a Engenharia Reversa baseada em Transformações, cuja contribuição é a obtenção de um projeto recuperado do sistema legado em um alto nível de abstração, representado na notação *UML* [6,7], garantindo sua evolução e sua manutenibilidade, tornando-o mais expressivo e de fácil entendimento. Esses Padrões utilizam-se de transformações que além de agilizar o processo de Engenharia Reversa, facilitam a obtenção da documentação do projeto.

Transformação de software consiste em gerar novas descrições dos programas numa mesma linguagem ou em outras linguagens. Pode ser usado tanto para implementar um sistema, a partir de especificações em alto nível de abstração, quanto para reimplementar um software existente a partir de seu código fonte, com o objetivo de atualizá-lo para ser executado em outras plataformas computacionais. Pode compreender não só a mudança de código, mas também uma mudança de paradigmas de programação, interfaces de usuário e arquiteturas de banco de dados.

Os padrões existentes em cada passo do processo são apresentados na Tabela 1, juntamente com suas descrições.

Nome Padrão	**Descrição**
Passo Identificar	
Definir hierarquia Chama/Chamado	Identificar no código legado a hierarquia de chamadas das unidades de programas.
Identificar Casos de Uso	Identificar casos de uso do código legado, e armazená-los como fatos na base de conhecimento.
Identificar Supostas Classes e Supostos Atributos	Identificar no código legado, supostas classes e supostos atributos que darão origem a classes e atributos no projeto final.
Identificar Supostos Métodos	Identificar fatos sobre supostos métodos e armazená-los na base de conhecimento.
Identificar Relacionamentos	Identificar os supostos relacionamentos e seus tipos e armazená-los na base de conhecimentos
Identificar Cardinalidades	Identificar as cardinalidades dos relacionamentos e armazená-las

	como fatos na base de conhecimento.
Passo Organizar	
Criar Supostas Classes e Supostos Atributos	Através dos fatos armazenados na base de conhecimento criar arquivos contendo as supostas classes e supostos atributos.
Alocar Supostos Métodos nas Supostas Classes	Alocar os supostos métodos que estão armazenados como fatos na base de conhecimento em suas supostas classes.
Passo Recuperar	
Criar Especificações das Classes e dos Atributos	Criar as especificações na linguagem de modelagem para as classes e seus atributos.
Criar Especificações dos Métodos	Criar as especificações dos métodos na linguagem de modelagem em suas respectivas classes.
Criar Casos de Uso	Gerar as especificações dos Casos de Uso na linguagem de modelagem.
Criar Diagrama de Seqüência	Criar as especificações em linguagem de modelagem para o diagrama de seqüência, seguindo o fluxo de controle do código legado organizado e dos casos de uso.
Criar Especificações dos Relacionamentos e Cardinalidades	Gerar as especificações dos relacionamentos e suas cardinalidades na linguagem de modelagem, de acordo com os fatos identificados.

Tabela 1. Catálogo dos Padrões de Processos.

Os Usos Conhecidos dos padrões apresentados estão descritos e apresentados no final do artigo.

2. Passo Identificar

O objetivo deste passo é identificar no código legado os elementos que compõem o pseudo Modelo Orientado a Objetos (OO). O Engenheiro de Software, utilizando-se de transformações de software que extrai essas informações sobre a estrutura do código legado, identificando de forma semi-automática elementos tais como, supostas classes, seus supostos atributos, supostos métodos, supostos relacionamentos, supostos casos de uso e a hierarquia de chamadas do código legado, e armazena-os como fatos numa Base de Conhecimento. Esses fatos auxiliarão na organização do código, segundo os princípios da Orientação a Objetos.

Para facilitar a identificação desses elementos foi adotado um "*Metamodelo*", segundo a notação *UML* [8], que representa os modelos Orientados a Objetos de um sistema, conforme mostra a Figura 1.

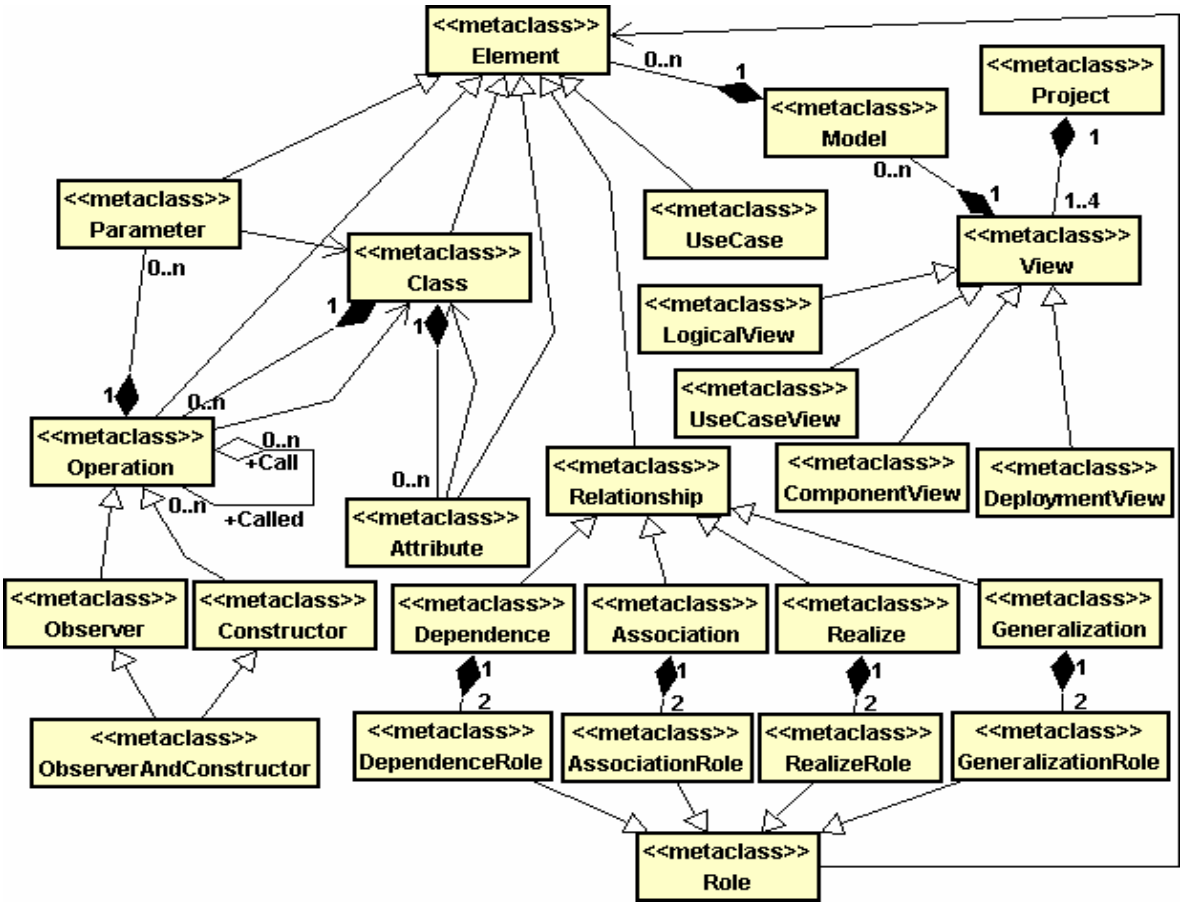


Figura 1. Metamodelo das especificações Orientadas a Objetos na notação UML

2.1. Definir Hierarquia de Chamadas (CHAMA/CHAMADO)

Intuito:

- Determinar o fluxo de execução das unidades de programa do sistema legado.

Problema:

- Há uma necessidade de se conhecer a hierarquia dos programas/módulos do sistema legado para identificar a seqüência de execução dessas unidades de programas.

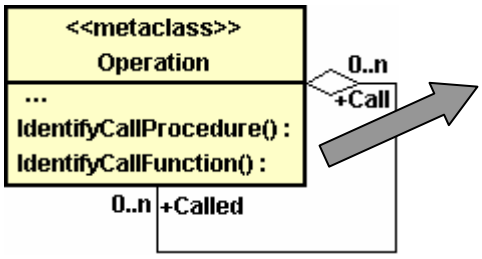
Influências:

- Ausência de documentação, exceto o código fonte (programas).
- A existência de grande quantidade de arquivos de programas e de dados nos sistemas legados, dificultando o seu entendimento.

Solução:

Através de transformações, com o auxílio do Engenheiro de Software, obter uma relação de unidades de programas (programas, procedimentos e funções) CHAMA/CHAMADO [9], fazendo uma referência cruzada entre essas unidades de programas, ou seja, identificando as unidades de programas por elas utilizadas (chamadas) e as unidades de programas que as utilizam (chamadoras).

A Figura 2 mostra o auto-relacionamento "*Call Called"* da metaclasse "*Operation"*, utilizados na geração dos fatos sobre a hierarquia de chamadas. A figura mostra também a transformação para geração dos fatos.




```
TRANSFORM  IdentifyCallProcedure
LHS: {{dast domain.procedureRole
 ProcedureCallProgram([[name_TargetProgram]]);
}}
POST-MATCH: {{dast txt.decls
...
}}
TEMPLATE T_ hierarchy
RHS: {{dast txt. decls
          [decls_CallerProgram]] call
          [[decls_TargetProgram]]
}}
```

Figura 2. Metaclasse *CallCalledHierarchy* do Metamodelo *OO*

A Figura 3 mostra um exemplo que ilustra a relação CHAMA/CHAMADO, onde as informações obtidas são baseadas na metaclasse "*Operation"* do metamodelo OO.

Na relação CHAMA/CHAMADO, se um procedimento ou função é chamado, registra-se também a unidade de programa em que ele se encontra. Por exemplo, a relação CHAMA/CHAMADO da Figura 3 mostra que, em uma unidade de programa Clipper chamado "Catproc.prg" tem-se os procedimentos "Inicial", "Logotipo" e "Janela", que estão sendo chamados por uma outra unidade de programa chamada "Menuprin.prg". Essas informações são importantes, pois ajudam a identificar uma seqüência dos programas a serem submetidos aos transformadores. Neste caso, a ordem para submeter os programas aos transformadores será do "Catproc.prg" e por último o "Menuprin.prg".

CHAMA	CHAMADO			
Menuprin	Inicial (p)	<Catproc.prg>	Modos2 (prg)	
	Logotipo (p)	<Catproc.prg>	Elimios (prg)	
	Janela (p)	<Catproc.prg>	Insecve (prg)	
	Ostela1 (prg)		Altcve (prg)	
	Ostela2 (prg)		Elicve (prg)	
	Modos1 (prg)		Indexa (prg)	
Ostela1	Logotipo (p)	<Catproc.prg>	Impos1 (prg)	<Catproc.prg>
	Aviso (p)	<Catproc.prg>	Buscli (f)	<Catproc.prg>
	Telalim (p)	<Catproc.prg>	Coluna (f)	<Ostela1.prg>
	Limpa (p)	<Catproc.prg>	Busvei (f)	<Modos1.prg>

Figura 3. Relação CHAMA/CHAMADO

Padrões Relacionados:

- Este padrão é a entrada para os padrões Identificar Casos de Uso, utilizado neste passo e para os padrões Criar Casos de Uso e Criar Diagrama de Seqüência, utilizados no passo Recuperar.
- Relaciona-se com o padrão ***Iniciar Análise dos Dados*** proposto por [14], que propõe a construção de uma tabela de Programas x Arquivos com objetivo de se saber o relacionamento entre eles. Já na relação CHAMA/CHAMADO o relacionamento é entre os módulos para se obter o fluxo de execução do sistema.

2.2. Identificar Casos de Uso

Intuito:

- Identificar os casos de uso do sistema legado e armazená-los na base de conhecimento.

Problema:

- Os Casos de Uso deverão ser identificados através da inspeção do código legado, que muitas vezes é implementado em uma linguagem procedural e não Orientada a Objetos.

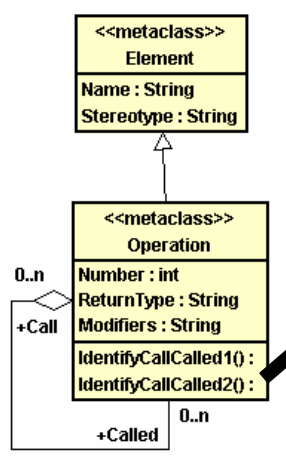
Influências:

- A falta de documentação para a identificação dos atores e Casos de Uso.
- A estrutura do código de sistemas legados procedimentais.

Solução:

Baseando-se na hierarquia de chamadas, e com o auxílio do Engenheiro de Software, aplicar transformações que deverão identificar e armazenar na base de conhecimento, fatos sobre os atores, que representam qualquer entidade que interage com o sistema, os casos de uso e a interação entre eles. Em seguida, através da interação do Engenheiro de Software, que deverá apontar o início de um caso de uso como sendo a chamada que inicia a execução de um determinado cenário significativamente importante para o sistema, como por exemplo, a chamada de uma unidade de programa do menu principal do sistema.

A Figura 4 exemplifica a transformação de identificação dos fatos dos supostos Casos de Uso, Atores e suas interações, que serão armazenadas na base de conhecimento.




```
TRANSFORM IdentifyCallCalled2
LHS: {{dast clipper.call_func_proc
   'DO' [[iden chamado]]
}}

POST-MATCH: {{dast txt.decls
...
fprintf(stderr," ** USE CASE IDENTIFICADO **");
fprintf(stderr,"O ATOR deve ser informado !");
cin >> ator;
fprintf(stderr,"O USE CASE deve ser informado !");
cin >> usecase;
...
}
   KBAssertIfNew("usecase_diagram([[ator]], [[usecase]])");
...
}}
```

Figura 4. Transformação para a identificação de supostos Casos de Uso

O resultado disso é o particionamento da hierarquia de chamadas em Casos de Uso.

Padrões Relacionados:

- Esse padrão tem como entrada o padrão Definir Hierarquia de Chama/Chamado, e é utilizado como entrada no padrão Criar Casos de Uso no passo Recuperar.
- Relaciona-se com o padrão ***"Obter Cenário"*** proposto por [14], cujo objetivo é obter os cenários do sistema através da análise das interfaces do sistema em operação, gerando a tabela de cenários do sistema.

2.3. Identificar Supostas Classes e Supostos Atributos

Intuito:

- Identificar no código legado, supostas classes e supostos atributos e armazená-los como fatos na Base de Conhecimento.

Problema:

- Diferentes formas de acesso a Banco de Dados e arquivos nos diferentes domínios do código legado e falta de uma estruturação dos dados.
- A desorganização do código legado dificulta bastante o trabalho de identificação de classes e atributos.

Influências:

- Reconhecer classes/objetos e seus relacionamentos a partir do código legado é difícil.
- Utilizar transformações para reconhecer fatos que poderão dar origem às respectivas classes e atributos.

Solução:

Utilizando-se de transformações de identificação, identificar as supostas classes e seus supostos atributos considerando-se os comandos de abertura e acesso aos arquivos ou banco de dados, comandos relacionados com as estruturas de dados, globais ou locais e as interfaces de interação do sistema com o ambiente externo.

O Engenheiro de Software deverá considerar as seguintes possibilidades:

- Sistemas que utilizam arquivos de dados:
  - ***Supostas Classes***
    Caso o sistema utilize arquivos de dados, cada arquivo de dados dá origem a uma suposta classe persistente da camada Banco de dados e cada estrutura de dados dá origem a uma suposta classe da camada Regras de Negócio. As telas de interação com o ambiente externo dão origem a supostas classes da camada de Interface.
  - ***Supostos Atributos***
    Caso o sistema utilize arquivos de dados, cada campo do arquivo de dados dá origem a um suposto atributo desse arquivo de dados.

- Sistemas que utilizam banco de dados:
  - ***Supostas Classes***
    Caso o sistema utilize banco de dados, cada tabela do banco de dados dá origem a uma suposta classe persistente da camada Banco de dados e cada estrutura de dados dá origem a uma suposta classe da camada Regras de Negócio. As telas de interação com o ambiente externo dão origem a supostas classes da camada de Interface.
  - ***Supostos Atributos***
    Caso o sistema utilize banco de dados, cada campo da tabela do banco de dados dá origem a um suposto atributo dessa tabela.

As supostas classes e supostos atributos identificados pelas transformações são armazenados como fatos na Base de Conhecimento pelas próprias transformações. A estrutura destes fatos deverá ser organizada de acordo com as propriedades que definem um atributo. Essas propriedades são encontradas em forma de meta-atributos das metaclasses *"Class"* e *"Attribute"*, do metamodelo OO, como mostra a Figura 5. É ilustrados também um exemplo de transformação implementada para geração de fatos das supostas classes e supostos atributos e o armazenamento dos mesmos na Base de Conhecimento.

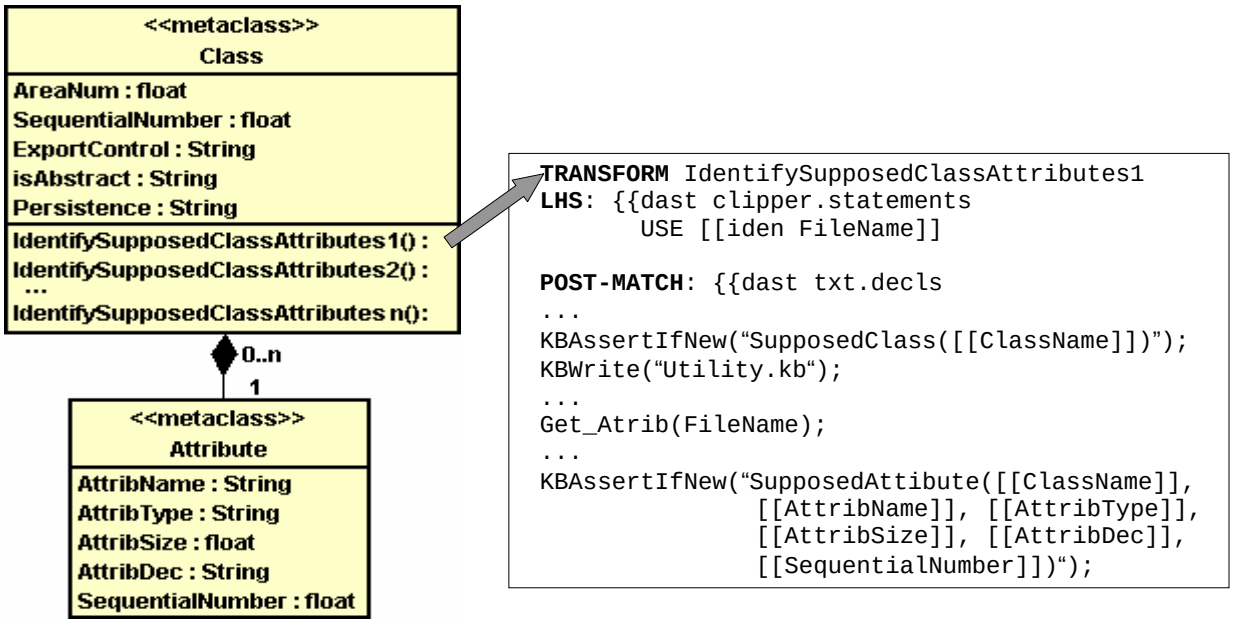


Figura 5. Metaclasse *Class* e *Attribute* do Metamodelo *OO*

Os meta-atributos *"Name", "Stereotype", "AttribName", "AttribType", "AttribSize", "AttribDec" e "SequentialNumber"* foram identificados na metaclasse *"Attribute"*. Deve-se verificar a existência de estruturas correspondentes para cada um desses meta-atributos no paradigma do código legado. Caso exista, a estrutura do fato do suposto atributo deverá conter esse dado.

Além disso, é criada uma suposta classe Principal cujo objetivo é alocar trechos que não podem ser alocados nas supostas classes de Negócio, como por exemplo, rotinas que iniciam o sistema.

Para ambos os casos, utilizando arquivos de dados ou banco de dados, as variáveis globais encontradas no código dão origem a atributos privados da suposta classe Principal.

Padrões Relacionados:

- Este padrão é a entrada para o padrão Criar Supostas Classes e Atributos, utilizado no passo Organizar.
- Relaciona-se com o padrão ***"Criar Visão OO dos Dados"*** proposto por [14], que tem como objetivo criar uma visão orientada a objetos dos dados, obtendo um Diagrama de Pseudo-Classes.

2.4. Identificar Supostos Métodos

Intuito:

- Identificar supostos métodos e armazená-los na base de conhecimento.

Problema:

- A dificuldade em identificar em sistemas legados, seus procedimentos, funções ou blocos de comandos.
- Anomalias podem ocorrer com comandos que consultam e/ou comandos que alteram a estrutura de dados.

Influências:

- Na maioria dos sistemas legados a estruturação do código não é modular, os códigos estão embaralhados em um procedimento principal, tornando quase impossível a identificação dos métodos.

Solução:

Identificar no código legado e armazenar na base de conhecimento, por meio das transformações, os supostos métodos a partir das unidades de programas que podem ser procedimentos, funções ou blocos de comandos executados pelo disparo de um evento de interface ou pela chamada de outra unidade de programa, respeitando o escopo, dependência de dados e visibilidade do código legado. Os supostos métodos são classificados em [9]:

- construtores (c), quando alteram uma estrutura de dados, e
- observadores (o), quando somente consultam a estrutura de dados.

Um suposto método que faz referência à somente um arquivo de dados é relacionado como suposto método da suposta classe identificada para esse arquivo. Se o suposto método não consulta nem modifica nenhum arquivo de dados, ele é relacionado com uma suposta classe de interface.

Para a identificação de um suposto método, as transformações deverão conter formas para verificar todas as características que um método necessita para ser representado na notação UML, de acordo com o metamodelo. Essas características são encontradas em forma de atributos da metaclasse *"Operation"*, conforme mostra a Figura 6.

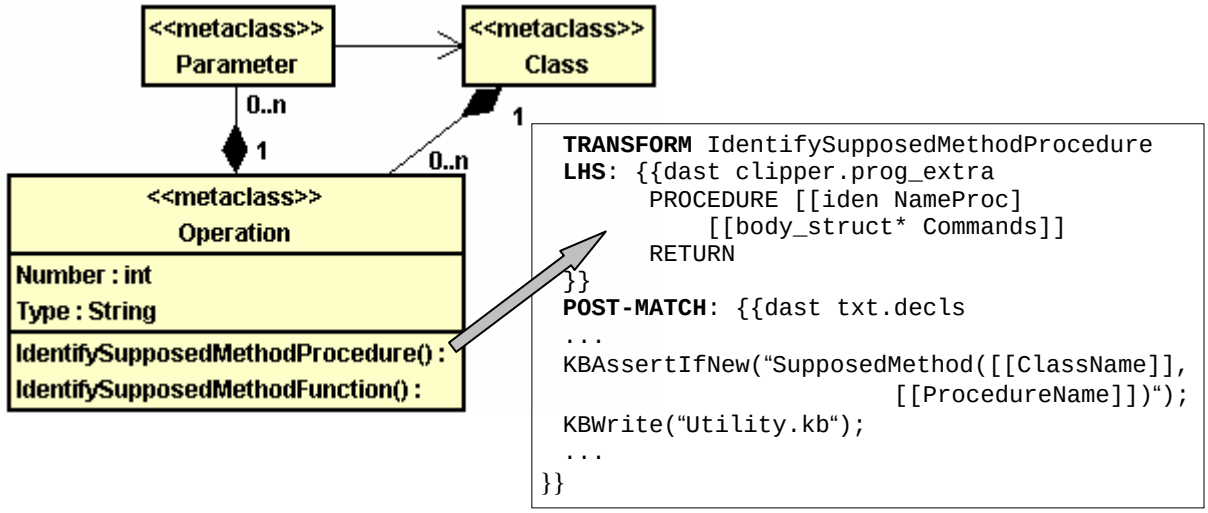


Figura 6. Metaclasse *Operation* do Metamodelo *OO*

Padrões Relacionados:

- Este padrão é a entrada para o padrão Alocar Supostos Métodos nas Supostas Classes, utilizado no passo Organizar.
- Relaciona-se com o padrão ***"Tratar Anomalias"*** proposto por [14], que tem como objetivo analisar as Descrições de Use Cases para tratar as anomalias, definindo, assim, os possíveis métodos das pseudo-classes (candidatas a classe) do sistema, obtidas pelo padrão Criar Visão OO dos Dados.

2.5. Identificar Relacionamentos

Intuito:

- Identificar os relacionamentos, seus tipos e armazená-los como fatos na Base de Conhecimento.

Problema:

- Pela falta de documentação existente, não se sabe como os arquivos de dados do sistema legado estão relacionados entre si.

Influências:

- Sistemas implementados em linguagens como Clipper, Cobol, entre outras, em geral não utilizam banco de dados relacional, no qual os relacionamentos entre as tabelas são explícitos.
- A partir da inspeção dos dados e da análise do código fonte é possível identificar o relacionamento dos dados no sistema legado.

Solução:

Os supostos relacionamentos são reconhecidos através de variáveis e comandos, embutidos nos transformadores, que manipulam dados de um arquivo ou de uma tabela de um banco de dados. É verificado se um campo de um arquivo ou tabela, identificado como chave, tem seus valores consultados e atribuídos a uma variável, e posteriormente o valor dessa variável é armazenado em campos de outro arquivo ou tabela ou é utilizado para fazer busca em registros de outros arquivos ou tabelas. São armazenadas informações na Base de Conhecimento, como: o nome do arquivo, tabela de origem e de destino, o nome do relacionamento na origem e no destino.

É necessária a interação do Engenheiro de Software para informar os tipos dos relacionamentos.

Um relacionamento pode ser do tipo:

- ***"é parte de"***, se um arquivo ou tabela complementa outro arquivo ou tabela,
- ***"é um"***, se um arquivo ou tabela compartilha a estrutura de outro arquivo ou tabela, ou
- ***"está associado"***, se existe navegabilidade entre os arquivos ou tabelas relacionados.

Utiliza-se de transformação para reconhecer e armazenar na Base de Conhecimento esses supostos relacionamentos e através de interações com o Engenheiro de Software informar o tipo de relacionamento mais apropriado para o suposto relacionamento reconhecido durante as transformações, conforme mostra a Figura 7, usando o metamodelo.


```
TRANSFORM
IdentifySupposedRelationship1
LHS: {{dast clipper.statements
     SEEK [[expr_num field]]
}}
POST-MATCH: {{dast txt.decls
...
Get_Atrib(FileName);
...
fprintf(stderr," ** RELAC IDT **");
fprintf(stderr,"O tipo do
relacionamento deve ser informado");
fprintf(stderr,"\n %s e parte de %s
?", class, classcorrente);
cin >> decisao;
if ((strcmp(decisao, "S") == 0)) {
     sprintf(rel_tipo, "\"%s\"","é
     parte de");
}
...
KBAssertIfNew("Relationship([[elemen
t1]],[[element2]], [[relationship]]
,[[cardinality]]");
KBWrite("UtilityRss.kb");
...
}}
```

Figura 7. Transformação para identificação de supostos relacionamentos

Padrões Relacionados:

- Este padrão é a entrada para o padrão Criar Especificações dos Relacionamentos e Cardinalidades, utilizado no passo Recuperar.
- Este padrão está relacionado com o padrão ***"Identificar Relacionamentos"*** proposto por [14]: que são identificados através da tabela Entidades x Chaves e trechos do código fonte que auxiliaram na elaboração dessa tabela.

2.6. Identificar Cardinalidades

Intuito:

- Identificar as cardinalidades dos relacionamentos.

Problema:

- Com a pouca documentação existente dos sistemas legados, se faz necessário analisar as diferentes estruturas de dados a fim de identificar o grau das cardinalidades.

Influências:

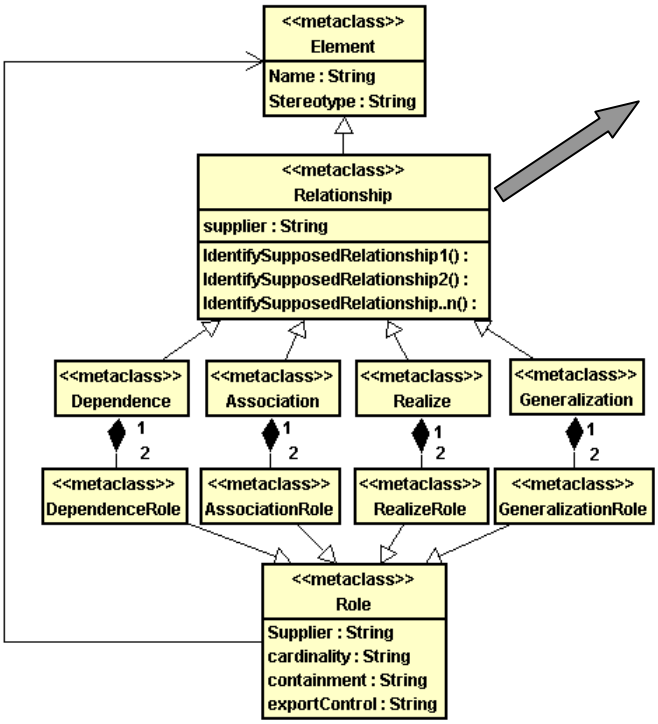
- Fazer uma análise em arquivos ou tabelas não normalizadas.
- Interação do Engenheiro de Software para validar as cardinalidades.

Solução:

Para determinar a cardinalidade de um relacionamento, é necessário analisar os campos dos arquivos ou tabelas que dão origem ao relacionamento e consultar as informações dos campos desses arquivos ou tabelas, registro a registro, para identificar o número de ocorrências de um mesmo valor armazenado.

Se um dado valor no campo de um arquivo ou tabela, ocorrer repetidas vezes significa que a cardinalidade máxima do arquivo ou tabela é ***"n"***.

No entanto, as informações obtidas dos arquivos ou tabelas podem não ser suficientes para definir as verdadeiras cardinalidades de um relacionamento, necessitando da interação do Engenheiro de Software. Essa interação poderá ser feita durante as transformações. A Figura 8 mostra uma transformação que permite ao Engenheiro de Software inserir as cardinalidades dos relacionamentos. O Engenheiro de Software interage para confirmar os valores dessas cardinalidades e atualizar os fatos de relacionamentos (*Relationship*).




```
TRANSFORM IdentifyCardinality1
LHS: {{dast clipper.statements
      SEEK [[expr_num field]]
}}
POST-MATCH: {{dast txt.decls
...
Get_Atrib(FileName);
...
fprintf(stderr,"Arquivos de dados:
existe o relacionamento: ");
fprintf(stderr,"%s está relacionado
com %s %s", classe, destino, classcc);
fprintf(stderr,"%s está relacionado
com %s %s", classcc, origem,classe);
fprintf(stderr,"está correto? ");
cin >> decisao;
if ((strcmp(decisao, "S") == 0)) {
sprintf(query, "Relationship(*a, *b,
*c, *d)", cardinality);}
...
KBAssert("Relationship([[element1]],
        [[element2]],[[relationship]],
          [[cardinality]])");
     KBWrite("UtilityRss.kb");
  ...
  }}
```

Figura 8. Transformação para identificação de cardinalidades

Padrões Relacionados:

- Este padrão é a entrada para o padrão Criar Especificações dos Relacionamentos e

Cardinalidades, utilizado no passo Recuperar.

3. Passo Organizar

O objetivo deste passo é organizar o código legado, em classes, atributos e métodos.

Com base nas informações identificadas no passo anterior (Identificar) armazenadas na Base de Conhecimento, utilizando de transformações apropriadas que deverá segmentar o código legado, organizando-o em classes, com seus atributos e métodos, produzindo um código intermediário, ainda na mesma linguagem do código legado.

Para atingir este objetivo o Engenheiro de Software deverá seguir os padrões que estão detalhados a seguir:

3.1. Criar Supostas Classes e Supostos Atributos

Intuito:

- Criar supostas classes e supostos atributos.

Problema:

- É difícil reconhecer supostas classes e supostos atributos em um sistema legado procedimental. Uma classe pode ser constituída de *n* programas do código legado.

Influências:

- Os fatos armazenados na Base de Conhecimento auxiliam o Engenheiro de Software na construção das supostas classes e atributos.

Solução:

As supostas classes identificadas no passo anterior darão origem a classes, através das transformações de organização. Para cada classe deve-se criar um arquivo com o mesmo nome da classe, onde serão alocados todos os seus atributos e métodos.

Os supostos atributos identificados no passo anterior devem originar atributos. Portanto, tem-se a interação do Engenheiro de Software para informar se cada um destes supostos atributos será realmente um atributo da classe.

Esta tarefa é realizada visando garantir a integridade nas informações pertinentes aos atributos, e diminuir a possibilidade de criação automática de atributos que não são necessários. Como vimos no passo Identificar, os fatos de supostos atributos são oriundos dos campos das tabelas, de arquivos de dados ou de bancos de dados do código legado. Existem recursos de implementação como os contadores, dentre outros, que não precisam ser recuperados como atributos no projeto do código legado, então o Engenheiro de Software poderá, durante essa interação, impedir a criação destes atributos, caso seja necessário.

A Figura 9 mostra o programa “Clientes.prg” organizado de acordo com os fatos identificados e armazenados na Base de Conhecimento, no passo anterior, Identificar.

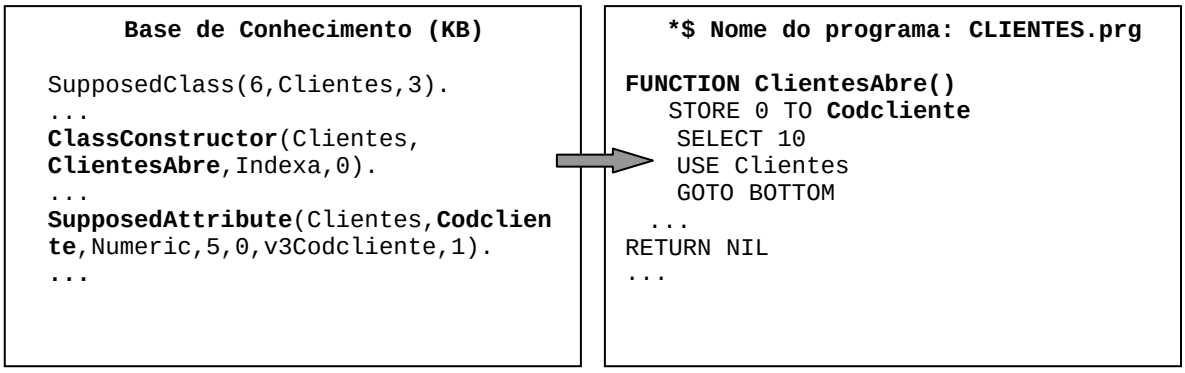


Figura 9. Organização da Classe Clientes de acordo com os fatos armazenados na KB

Padrões Relacionados:

- Este padrão tem como entrada o padrão Identificar Supostas Classes e Atributos, do passo Identificar, e é utilizado como entrada para o padrão Criar Especificações das Classes e dos Atributos no passo Recuperar.

3.2. Alocar Supostos Métodos nas Supostas Classes

Intuito:

- Criar os supostos métodos nas supostas classes criadas.

Problema:

- A desorganização do código legado com relação às funções, *procedures* e métodos, que futuramente pertencerão a apenas uma classe, dificulta a alocação dos supostos métodos.

Influências:

- A forma ou maneira como os supostos métodos acessam os dados influenciam na sua alocação.
- A identificação feita anteriormente pelo transformador e o armazenamento dos fatos na Base de Conhecimento, facilita a correta alocação de métodos nas supostas classes, através das transformações.

Solução:

O objetivo desta tarefa é alocar nos arquivos das classes, os seus respectivos supostos métodos identificados no passo, Identificar, corrigindo-se anomalias da seguinte forma:

Os supostos métodos são classificados em construtores (c), quando alteram a estrutura de dados, e observadores (o), quando somente consultam as estruturas de dados. Assim,

quando um suposto método refere-se a mais de uma suposta classe ele deve ser alocado usando os seguintes critérios [9]:

- Se construtor de uma suposta classe e observador de outra (oc), será alocado na suposta classe que faz referência como construtor;
- Se observador de uma suposta classe e construtor de várias outras classes (oc+), será alocado na primeira suposta classe que faz referência como construtor;
- Se observador de mais de uma suposta classe e construtor de apenas uma (o+c), será alocado na primeira suposta classe que faz referência como observador.

O Engenheiro de Software poderá utilizar um Sistema de Transformação para realizar esta tarefa. As transformações deverão consultar a Base de Conhecimento (KB) para obter informações sobre os supostos métodos e alocá-los na workspace da suposta classe a qual esse suposto método pertence.

A Base de Conhecimento é consultada para obter informações sobre os supostos métodos, e as anomalias não solucionadas pelas transformações são resolvidas pelo Engenheiro de Software, que pode tomar decisões, interagindo com o Sistema de Transformação durante a aplicação das transformações.

Padrões Relacionados:

- Este padrão tem como entrada o padrão Identificar Supostos Métodos, do passo Identificar, e é utilizado como entrada para o padrão Criar Especificações dos Métodos no passo Recuperar.

4. Passo Recuperar

O objetivo deste passo é recuperar as especificações numa linguagem de Modelagem, obtendo assim o Projeto Orientado a Objetos Recuperado do Código Legado.

O Engenheiro de Software, com o auxílio de transformações e de um Sistema Transformacional, parte do Código Legado Organizado, obtido no passo Organizar, para gerar essas especificações do projeto, descritas na linguagem de modelagem alvo. Depois de geradas as especificações, estas podem ser importadas pelo Engenheiro de Software numa ferramenta CASE para poder visualizar o projeto recuperado graficamente.

Os padrões utilizados para a geração das especificações do projeto são apresentados a seguir.

4.1. Criar Especificações das Classes e dos Atributos

Intuito:

- Gerar especificações na linguagem de modelagem para as classes e atributos obtidos no passo anterior.

Problema:

- As informações sobre as classes e atributos foram identificadas e organizadas, mas ainda precisam ser convertidas para uma linguagem de modelagem, na qual requer regras sintáticas.

Influências:

- O trabalho de conversão do código legado para o código OO, normalmente, é repetitivo, e está sujeito a erros. Esse padrão visa a automatização do processo de conversão das classes e seus respectivos atributos.

Solução:

Através de transformações, criar as especificações na linguagem de modelagem MDL (*Modelling Domain Language*), para as classes e seus atributos. Outras linguagens de modelagem poderiam ser utilizadas, como: Notação do Coad/Yourdon [25], OMT [26], Booch [24], etc. Para realizar esta atividade deve-se:

- Para cada classe, criar a especificação correspondente na linguagem de modelagem*;*
- Para cada atributo da classe, criar a especificação correspondente na linguagem de modelagem;

Cada unidade de programa do sistema legado, organizada como uma classe, dá origem a sua especificação na linguagem de modelagem. O mesmo acontece para cada atributo da unidade de programa reconhecido, dando origem a sua respectiva especificação na linguagem de modelagem. A Figura 10 mostra uma parte do transformador que reconhece as unidades de programa (funções, procedimentos, programas) do sistema legado, para gerar as especificações de classes e de seus atributos na linguagem MDL.

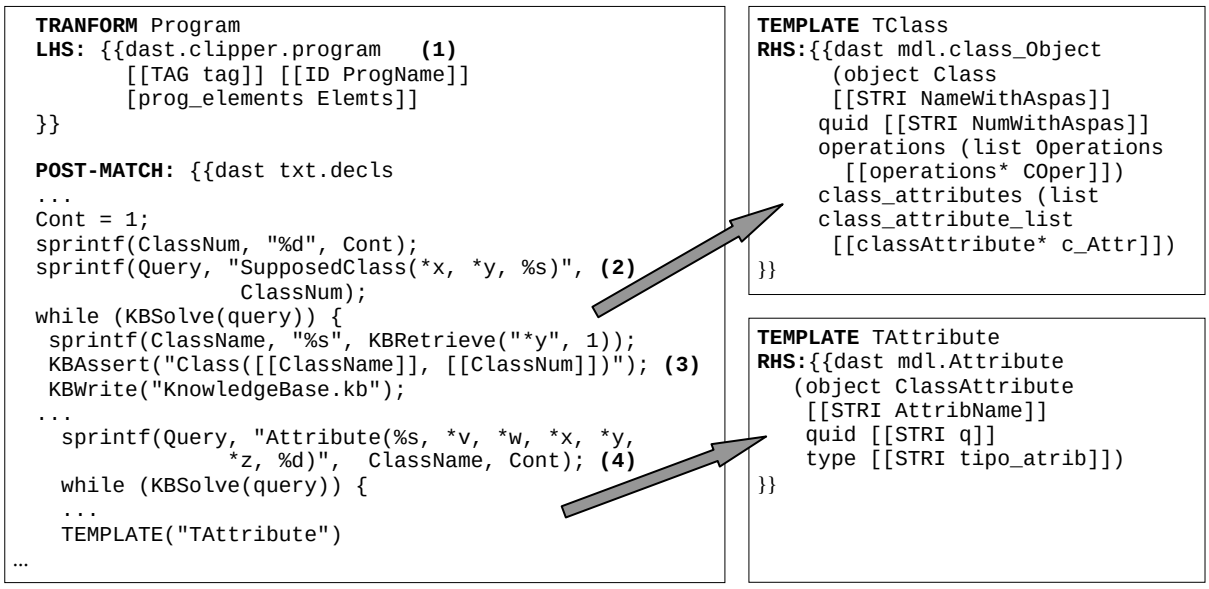


Figura 10. Transformação que gera a especificação na linguagem de modelagem para a classe e seus atributos

Padrões Relacionados:

- Este padrão tem como entrada o padrão Criar Supostas Classes e Supostos Atributos, do passo Organizar.
- Relaciona-se com os padrões "***Definir as Classes***" e "***Definir os Atributos***" proposto por [14], que apenas consolidam diversos diagramas já obtidos numa apresentação segundo a visão de Orientação a Objetos.

4.2. Criar Especificações dos Métodos

Intuito:

- Gerar as especificações dos métodos na linguagem de modelagem.

Problema:

- Com os supostos métodos já identificados e armazenados na base de conhecimento, é preciso pesquisar na linguagem de modelagem a estrutura de um método, para que se possa gerar suas especificações neste formato, seguindo suas regras sintáticas.
- A inclusão do corpo do método já na linguagem alvo da reimplementação, na especificação do método, completa o futuro projeto recuperado do código legado.

Influências:

- O trabalho de conversão do código legado para o código OO, normalmente, é repetitivo e está sujeito a erros. Esse padrão visa a automatização do processo de conversão dos métodos.

Solução:

Construir transformadores para criar as especificações das assinaturas dos métodos e transformar os corpos dos métodos diretamente para uma linguagem de programação alvo da reimplementação.

Assim como é feito para as classes e seus respectivos atributos, a especificação dos métodos também é obtida com o auxílio de um Sistema Transformacional. A Figura 11 mostra, como exemplo, a criação das especificações de um método na linguagem de modelagem.

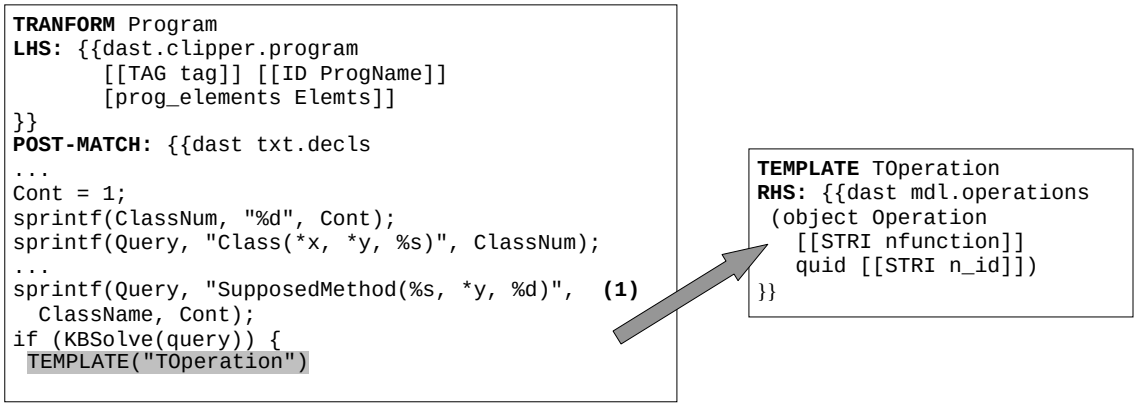


Figura 11. Transformação que gera a especificação na linguagem de modelagem para um método

No exemplo da Figura 11, para cada fato *"SupposedMethod"* é criada a sua correspondente especificação na linguagem de modelagem (1) através da *TEMPLATE "TOperation"*.

Padrões Relacionados:

- Este padrão tem como entrada o padrão Alocar Supostos Métodos nas Supostas Classes, do passo Organizar.
- Relaciona-se com o padrão ***"Definir Métodos"*** proposto por [14], que obtém os métodos das classes no Diagrama de Classes em construção por meio da descrição do Use Case correspondente.

4.3. Criar Casos de Uso

Intuito:

- Utilizar transformações para criar os casos de uso.
- Identificar os fatos sobre os supostos casos de uso armazenados na base de conhecimento para a geração das especificações correspondentes na linguagem de modelagem.

Problema:

- Os casos de uso já foram identificados e armazenados na base de conhecimento como fatos, mas ainda precisam ser convertidas na linguagem de modelagem, respeitando suas regras sintáticas.

Influências:

- O trabalho de conversão do código legado para o código OO, normalmente, é repetitivo e está sujeito a erros. Esse padrão visa a automatização do processo de conversão dos fatos aramazenados em Base de Conhecimento em Modelos de Caso de Uso.

Solução:

Criar as especificações dos Casos de Uso a partir dos fatos recuperados da Base de Conhecimento.

Para cada suposto Caso de Uso já identificado, deve-se gerar especificações correspondentes na Linguagem de Modelagem, observando as seguintes recomendações:

- Para cada Caso de Uso identificado, criar as respectivas especificações na linguagem de modelagem;
- Para cada ator identificado, criar especificações de classe com "stereotype" pré-definido para a identificação de atores na linguagem de modelagem;

A Figura 12 mostra, como exemplo, a criação das especificações de um Use Case através da *TEMPLATE T_UseCase*, a partir de um fato *UseCase* da Base de Conhecimento.

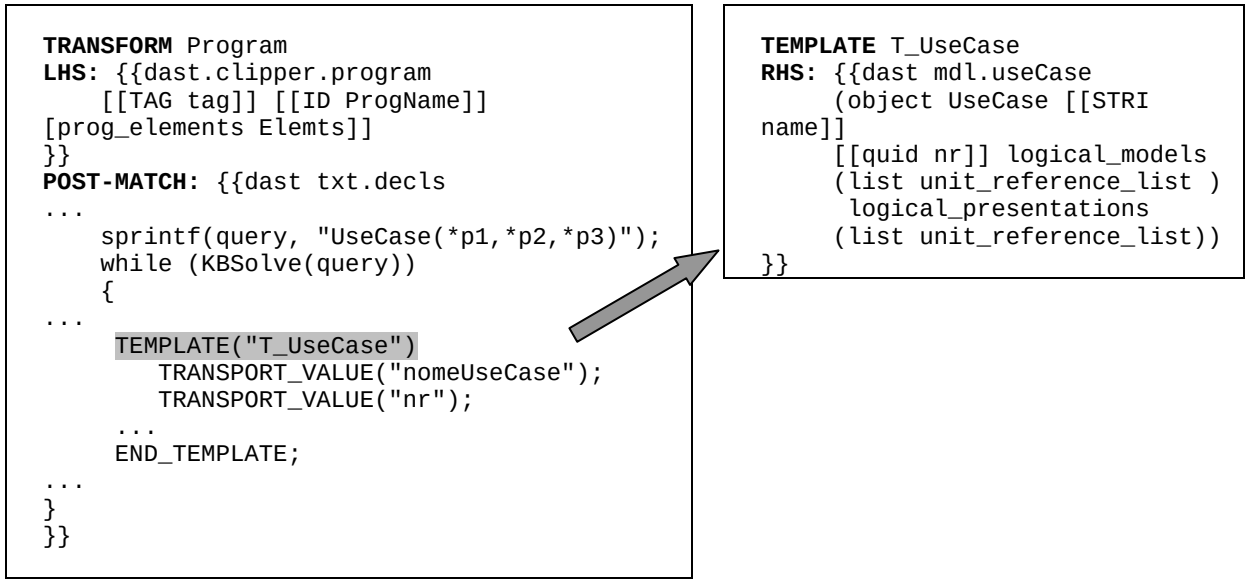


Figura 12. Transformação de recuperação das especificações de supostos Casos de Uso

Padrões Relacionados:

- Este padrão tem como entrada os padrões Identificar Casos de Uso e Definir hierarquia chama/chamado, do passo Identificar.
- Relaciona-se com o padrão ***"Construir Diagramas de Use Cases"*** proposto por [14], que constroem os diagramas de casos de uso do sistema a partir da tabela de cenários do sistema elaborada pelo padrão Obter Cenários [14]. Este padrão é executado em conjunto com o padrão "***Interview During Demo"*** proposto por [13], entrevistar o usuário durante o sistema em Operação.

4.4. Criar Diagrama de Seqüência

Intuito:

- Identificar o fluxo de controle do código legado organizado e dos casos de uso para a geração do diagrama de seqüência.

Problema:

- O fluxo identificado no passo, Identificar, representa o fluxo entre as unidades de programas, e não um fluxo entre os objetos, portanto representa um diagrama de seqüência procedural.

Influência:

- O trabalho de conversão do código legado para o código OO, normalmente, é repetitivo e está sujeito a erros. Esse padrão visa a automatização do processo de conversão dos fatos aramazenados em Base de Conhecimento em Diagramas de Sequencia.

Solução:

Baseado nos Casos de Uso, para cada seqüência de chamada de cada caso de uso cria-se um diagrama de seqüência.

Cada diagrama de seqüência corresponde a um curso, Normal ou Alternativo, do caso de uso. O fluxo de controle do Código Legado Organizado, definido pela seqüência de chamadas das unidades de programas, é mantido através das conexões de mensagens entre os objetos.

Padrões Relacionados:

- Este padrão utiliza-se dos padrões Definir hierarquia chama/chamado e Identificar Casos de Uso, do passo Identificar, para realizar sua tarefa.
- Relaciona-se com o padrão ***"Construir Diagramas de Seqüência"*** proposto por [14], que objetiva a construção dos Diagramas de Seqüência, a partir de cada descrição de Use Case, obtida pelo padrão Elaborar Descrição de Use Cases.

4.5. Criar Especificações dos Relacionamentos e Cardinalidades

Intuito:

- Gerar as especificações dos relacionamentos e suas cardinalidades na linguagem de modelagem, baseando-se nos fatos armazenados na base de conhecimento.

Problema:

- Conhecer as regras sintáticas das especificações que representam os relacionamentos e suas cardinalidades na linguagem de modelagem.

Influência:

- Os fatos armazenados na Base de Conhecimento, identificados pelos padrões Identificar Cardinalidades e Identificar Relacionamentos, auxiliam na correta especificação dos relacionamentos e cardinalidades.

Solução:

Criar as especificações dos relacionamentos com suas respectivas cardinalidades, baseado nos fatos armazenados na Base de Conhecimento.

- Para cada suposto relacionamento já identificado, deve-se gerar especificações correspondentes na linguagem de modelagem, acoplando-se suas respectivas cardinalidades também já identificadas;

Assim, são criadas, para cada relacionamento, as especificações na linguagem de modelagem contendo informações da classe de origem e classe de destino, com suas cardinalidades.

A Figura 13 mostra, por exemplo, uma transformação que reconhece na Base de Conhecimento um fato *Relationship* de uma associação e gera sua respectiva especificação na linguagem de modelagem MDL através da *TEMPLATE T_Association*.

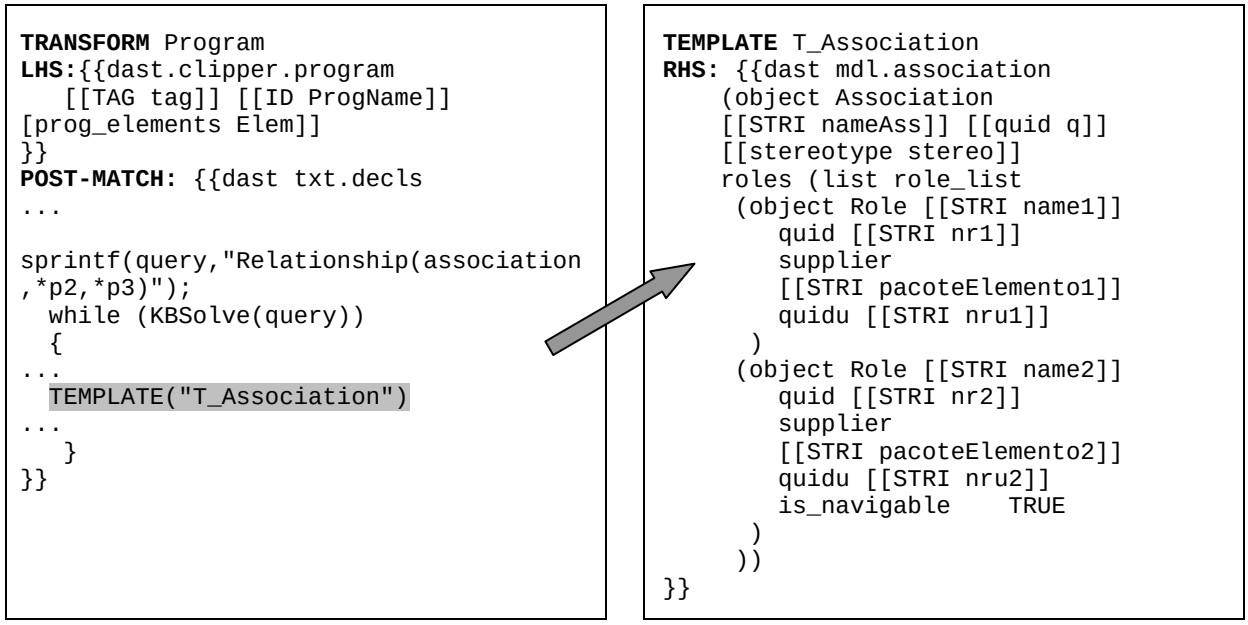


Figura 13. Transformação de recuperação das especificações de um relacionamento

Padrões Relacionados:

- Este padrão tem como entrada o padrão Identificar Relacionamentos e Identificar Cardinalidades, do passo Identificar.

5. Usos Conhecidos dos Padrões Propostos:

Finalmente, o item **Usos Conhecidos** é explicitado a seguir, por ser comum a todos os padrões.

Os conceitos aqui apresentados foram utilizados nos processos de engenharia reversa realizados em [15, 16, 17, 18, 19, 20, 21, 22], sem pertencerem a um grupo de Padrões.

Em tais estudos e experiências foram utilizados dois mecanismos para auxiliá-los no processo de Engenharia Reversa: o ST Draco-PUC [1, 11] e a ferramenta MVCase [2]. A Figura 14 exemplifica todo o processo com as técnicas e mecanismos utilizados no processo de Engenharia Reversa.

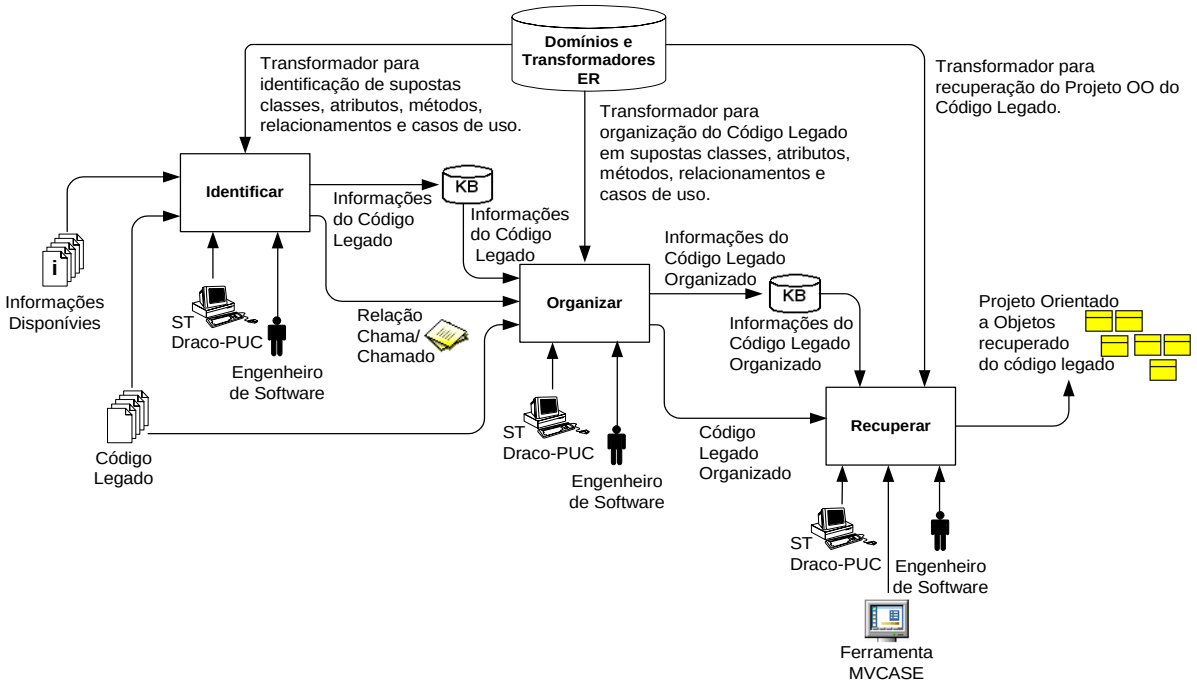


Figura 14. Padrões de Processo para a Engenharia Reversa.

6. Agrupando os Padrões

Após a visão geral dos padrões descritos nas seções anteriores, a Figura 15 mostra a interação destes padrões, provendo um conjunto de Padrões de Processo para a Engenharia Reversa baseada em Transformações.

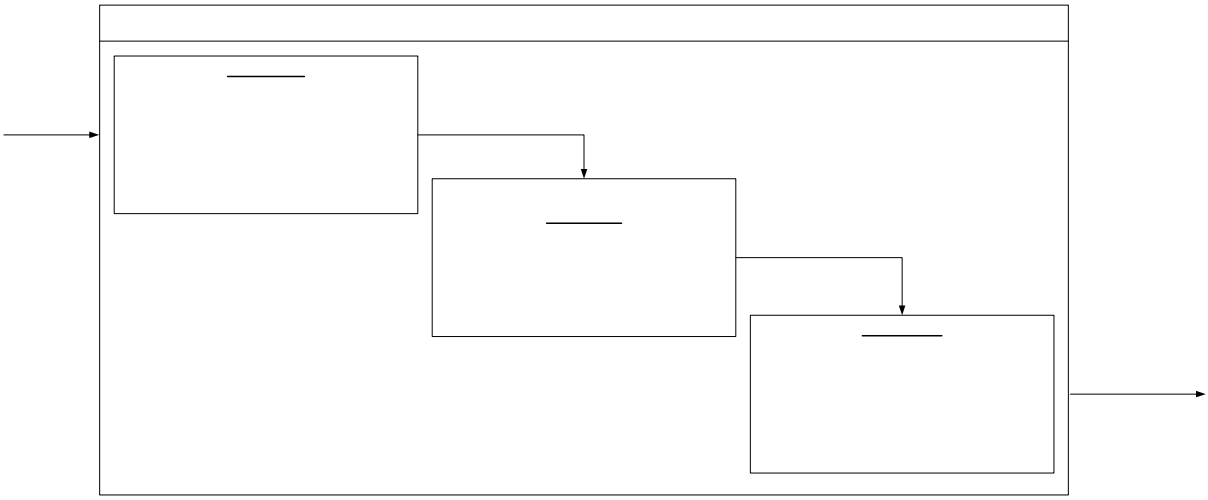


Figura 15. Interação dos Padrões de Processo para a Engenharia Reversa.

Na Engenharia Reversa, o Engenheiro de Software parte do *Código Legado* e suas informações disponíveis, para obter o Projeto Orientado a Objetos Recuperado do Código Legado e as Especificações MDL do Projeto Orientado a Objetos. O projeto Orientado a Objetos é representado por modelos gráficos e textuais, por exemplo, usando a notação UML.

As especificações do Projeto Orientado a Objetos são descritas na linguagem de modelagem MDL. Estas descrições em MDL permitem recuperar o projeto Orientado a Objetos e dar continuidade ao processo de reengenharia.

7. Conseqüências

O maior problema no uso dos padrões apresentados está no fato do Engenheiro de Software ter um vasto conhecimento sobre o Sistema Transformacional Draco-PUC e sobre a construção de transformações, já que a maior parte das tarefas serão realizadas por elas.

É necessário também grande conhecimento sobre as linguagens do domínio origem e alvo do processo de Reengenharia para a construção correta das transformações.

Agradecimentos

Os autores gostariam de agradecer a Prof.ª Drª. Rosana Teresinha Vaccare Braga pelas suas orientações durante o processo de s*hepherding*.

Referências

[1] Leite, J.; Sant'Anna, M.; Freitas, F.; **Draco-PUC: A Technology Assembly For Domain Oriented Software Development.** Proceedings of ICSR94 (International Conference on Software Reuse), IEEE Press, 1994.

[2] Almeida, E., S., Lucrédio, D., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. **MVCase Tool: An Integrating Technologies Tool for Distributed Component Development (in portuguese).** In SBES'2002, 16th Brazilian Symposium on Software Engineering, Tools Session.

[3] D'Souza, D., F., Wills, A., C., 1999. **Objects, Components, and Frameworks with UML, The Catalysis Approach,** Addison-Wesley. USA.

[4] Gamma, E., et al., 1995. **Elements of Design Patterns: Elements of Reusable Object Oriented Software**, Addison-Wesley.

[5] Werner, C.M.L.; Braga, R. M.M. **Desenvolvimento Baseado em Componentes.** XIV Simpósio Brasileiro de Engenharia de Software SBES2000 Minicursos e Tutoriais – pg. 297-329 – 2-6 de Outubro, 2000.

[6] Booch, G. et al. **The Unified Modeling Language.** User Guide. USA: Addison Wesley, 1999.

[7] Fowler R, M. **UML Destilled. Applying the Standard Object Modeling Language.** England:Addison Wesley, 1997.

[8] **Unified Modeling Language 1.4 specification**. Available at site Object Management Group. URL: http://www.omg.org/technology/documents/formal/uml.htm. Consulted in February, 2003.

[9] Penteado, R.D. **Um Método para Engenharia Reversa Orientada a Objetos.** São Carlos-SP, 1996. Tese de Doutorado. Universidade de São Paulo. 251p.

[10] Neighbors, J.M., **Software Construction Using Components**, Doctoral dissertation, Information and Computer Science Dept. University of California, Irvine, 1980.

[11] Neighbors, J.M. **The Draco approach to Constructing Software from Reusable Components**. IEEE Transactions on Software Engineering. v.se-10, n.5, pp.564-574, September, 1984.

[12] Almeida, E., S., Bianchini, C., P., Prado, A., F., Trevelin, L., C., 2002. **MVCase: An Integrating Technologies Tool for Distributed Component-Based Software Development.** In APNOMS'2002, The Asia-Pacific Network Operations and Management Symposium,Poster Session. Proceedings of IEEE.

[13] Demeyer, S.; Ducasse, S.; Nierstrasz, O., **"A Pattern Language for Reverse Engineering".** Proceedings of the 5th European Conference on Pattern Languages of Programming and Computing, (EuroPLOP'2000), Andreas Ruping(Ed.), 2000.

[14] Recchia, E. L.; Penteado, R. – **FaPRE/OO: Uma Família de Padrões para Reengenharia Orientada a Objetos de Sistemas Legados Procedimentais.** Artigo apresentado no SugarloafPLoP2002 – The Second Latin American Conference on Pattern Languages of Programming – Agosto/2002, Itaipava – RJ.

[15] Novais, E. R. A.; **Reengenharias de Software Orientadas a Componentes Distribuídos.** São Carlos/SP, 2002, Dissertação de Mestrado, Universidade Federal de São Carlos.

[16] Bossonaro, A. A.; **Estratégia de Reengenharia de Software usando Transformações**. Disponível em URL: http://www.recope.dc.ufscar.br.

[17] Jesus, E. S. **Engenharia Reversa de Sistemas Legados Usando Transformações**, 2000 - Dissertação de Mestrado, Ciências da Computação, UFSCAR - Universidade Federal de São Carlos.

[18] Fukuda, A. P. **Refinamento Automático de Sistemas Orientados a Objetos Distribuídos.** São Carlos/SP, 2000. Dissertação de Mestrado. Universidade Federal de São Carlos.

[19] Sametinger, J. **Software Engineering with Reusable Components**. Springer-Verlag, 1997.

[20] Garcia, V. C., Fontanette, V., Perez, A. B., Prado, A. F., Sant'Anna, M.; **RHAE/CNPQ - Projeto: Reengenharia de Software Usando Transformações (RST)**. Processo número: 610.069/01-2.

[21] Nogueira, A. R.; **Transformação de DataFlex Procedural para Visual DataFlex Orientado a Objetos reusando um Framework.** São Carlos/SP, 2002, Dissertação de Mestrado. Universidade Federal de São Carlos.

[22] Prado, A.F. **Estratégia de Engenharia de Software Orientada a Domínios**. Rio de Janeiro-RJ, 1992. Tese de Doutorado. Pontifícia Universidade Católica. 333p.

[23] Abrahão, S.M., Prado, A.F; Sant'anna, M.; **A Semi-Automatic Approach for Building Web-Enabled Aplications from Legacy.** Submitted on 4 IEEE international software engineering Standards Symposium – Curitiba, Brasil, May 1999.

[24] Booch, G., Rumbaugh, J., Jacobson, I; **The Unified Modeling Language Reference Manual**, Addison-Wesley, 1999.

[25] Coad, P., Yourdon, E. **Object-Oriented Design**. Yourdon Press/Prentice-Hall, Englewood Cliffs, NJ, 1991.

[26] Rumbaugh, J., Blaha, M., Premerlani, W. F. Eddy, and W. Lorensen. **Object-oriented modeling and design**. Prentice-Hall, 1991.

PATI-MVC: Padrões MVC para Sistemas de Informação

Gabriela T. De Souza*
gabi@atlantico.com.br
Instituto Atlântico

Carlo Giovano S. Pires*
cgiovano@atlantico.com.br
Instituto Atlântico

Márcio de Oliveira Barros
marcio@cos.ufrj.br
Universidade Federal do Rio de Janeiro

Resumo

Sistemas de informação são baseados em manutenção e consulta de dados. Usualmente, estes sistemas têm funcionalidades simples, mas estas funcionalidades se repetem várias vezes. O objetivo dos padrões PATI-MVC é apresentar soluções para o projeto de funcionalidades de manutenção de dados em sistemas de informação.

Abstract

Information systems are based on registering, searching, and maintaining data. Usually, they have simple functionality, but these functions are repeated several times within an information system. The aim of PATI-MVC patterns is to present design solutions for data maintaining in information systems.

1. Introdução

Em sistemas de informação, as funcionalidades são muito concentradas em cenários de uso baseados em manutenção e consulta de dados, gerando vários problemas recorrentes durante o desenvolvimento do sistema. Esses problemas são comumente relacionados a aspectos de apresentação, interação e tratamento de eventos de usuário; separação entre aspectos de apresentação, regras de negócio e persistência de dados. Neste artigo serão apresentados os padrões CRUD-MVC e Registro com Busca que são baseados no modelo CRUD e no padrão de arquitetura MVC (Modelo-Visão-Controle) [3] e tem o objetivo de fornecer soluções para problemas recorrentes em sistemas de informação.

O modelo CRUD (*Create, Retrieve, Update, Delete*) é aplicado em cenários de consulta e manutenção de entidades em sistemas de informação. Esses cenários são

* Este trabalho foi suportado pelo Instituto Atlântico (*www.atlantico.com.br*).

tipicamente construídos com base em operações de criação (ou inclusão) de novos dados em um repositório, localização (ou consulta de dados), atualização e exclusão dos dados. Os dados mantidos pelas operações CRUD estão relacionados a uma ou mais entidades de negócio. Uma forma bastante comum para armazenar as entidades é através de tabelas em bancos de dados relacionais.

O padrão de arquitetura MVC determina a separação de responsabilidades de uma aplicação em componentes de modelo, visão e controle. O modelo é formado por entidades de negócio que representam os dados do sistema. Componentes de visão (telas gráficas ou páginas HTML, por exemplo) têm o objetivo de apresentar as informações pertencentes ao modelo e capturar eventos de usuário. Componentes de controle tratam os eventos capturados, notificam e coordenam componentes de modelo. Os componentes de controle fazem a ponte entre componentes de visão e do modelo.

Os padrões CRUD-MVC e Registro com Busca participam de uma família de padrões dedicada a construção de sistemas de informação. O padrão CRUD-MVC apresenta os relacionamentos e interações entre classes do modelo, visão e controle para realização de operações CRUD e suporte para outras operações de negócio. O padrão Registro com Busca concentra-se nas classes da visão, indicando padrões de interação com usuário de acordo com a complexidade do objeto e cenário de negócio.

2. Padrão CRUD-MVC

2.1 Contexto

Sistemas de informação requerem funcionalidades de negócio implementadas através de interface humano-computador (IHC), componentes para tratamento de regras de negócio e acesso a dados para operações CRUD e operações de negócio.

2.2 Problema

Como tratar funcionalidades recorrentes de criação, consulta, atualização e exclusão de dados em sistemas de informação considerando aspectos de apresentação e tratamento de eventos, regras de negócio e persistência?

2.3 Forças

- Deve fornecer baixo acoplamento, facilitando a troca de mecanismos de interface e acesso a dados
- Deve permitir reuso de estrutura e comportamentos de componentes de IHC, e tratamento de eventos CRUD

2.4 Solução

A idéia básica da solução é criar uma camada de classes abstratas que capturam a estrutura, comportamento e colaborações genéricas entre modelo, visão e controle e criar várias implementações (para cada cenário de uso CRUD) herdando as características da camada abstrata e definindo as características concretas da aplicação. O projetista deve utilizar o padrão definindo, as características gerais de interface, negócio e persistência na camada abstrata.

As classes abstratas de visão determinam operações para tratamento de eventos CRUD com enfoque em questões de interface com usuário, por exemplo, a operação *tratarExclusao* pode sempre exibir uma interface de diálogo para confirmar a operações antes de acionar o controlador. Outros eventos de interface com usuário e outras características que possam ser reutilizadas também são de responsabilidade das classes abstratas de visão. As classes abstratas de controle permitem a fabricação de entidades, tratamento de eventos CRUD e também suportam outras operações genéricas que o projetista possa definir. O tratamento de eventos CRUD nas classes de controle é voltado para o fluxo de negócio, ao contrário do tratamento de eventos CRUD nas classes de visão. A classe abstrata de entidade permite definir as operações CRUD propriamente ditas ou outras operações genéricas de entidades. As classes concretas implementam as operações definidas na camada abstrata e complementam comportamentos e características específicas de negócio.

2.5 Estrutura

O padrão apresenta uma camada de classes abstratas para visão, modelo e controle. As classes abstratas utilizam métodos de fabricação definidos de acordo com o padrão *Factory Method* [2], e assim, permitem que as subclasses concretas determinem as implementações das classes abstratas. Essa estrutura permite que classes do nível abstrato possam implementar as interações entre visão, modelo e controlador e as operações CRUD que são reutilizadas por todas as classes concretas da aplicação. Os relacionamentos do padrão MVC existem no nível das classes abstratas (para tratamento de eventos CRUD) e no nível das classes concretas (para tratamento de eventos de negócio específicos).

A classe de visão notifica *ControladorCRUD* sobre eventos de criação, alteração, exclusão e consulta de entidades de negócio. Por sua vez, *ControladorCRUD* é uma classe de controle que define mecanismos para a criação da entidade de negócio e operações reutilizáveis para o tratamento dos eventos CRUD sobre uma interface genérica para uma entidade de negócio. *Entidade* é uma classe de modelo que define a interface genérica de entidade de negócio, com contratos para as operações CRUD.

Para cada uma das classes genéricas, podem existir várias implementações concretas de acordo com as diversas entidades de negócio do sistema. As implementações concretas especializam as classes genéricas, definindo as características e comportamentos concretos de IHC (*ComponenteIHCConcreto*), fabricando controladores concretos (*ControladorConcreto*) que criam entidades concretas (*EntidadeConcreta*) e implementam tratamentos de outros eventos de negócio (que não operações CRUD). As entidades concretas definem os atributos (informação), implementam as operações CRUD e outras operações de negócio. A Figura 1 apresenta o diagrama de classes do padrão e seus componentes.

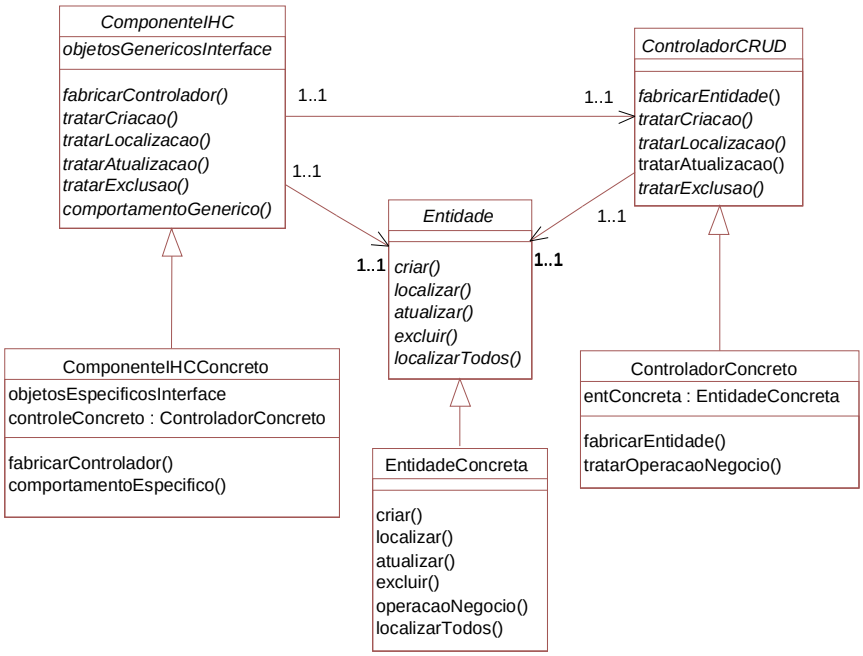


Figura 1 – Diagrama de classes do Padrão CRUD-MVC

2.6 Participantes

- **ComponenteIHC**
  - Pode ser formado por vários componentes de interface com usuário
  - Define as características de interação com usuário (*objetosGenericosInterface*) presente em todo componente de IHC de um sistema. Pelo menos, comandos para criação, localização, atualização e exclusão. Pode definir outras características a serem reutilizadas (por exemplo, ajuda e comando para fechar/ocultar interface)
  - Define comportamento existente em toda interface de usuário (*comportamentoGenerico*). Por exemplo, controle de acesso a operações e alertas
  - Fornece método *Factory* [2] *fabricarControlador* para criação do objeto de controle e armazena referência através de interface *ControladorCRUD*
  - Fornece métodos genéricos para tratamento dos eventos de interface com usuário (*tratarCriacao*, *tratarLocalizacao*, *tratarLocalizacao*, *tratarExclusao*). Os métodos de evento notificam seu observador (*ControladorCRUD*) das ações tomadas pelo usuário. Os métodos genéricos permitem definir ações de pré-processamento e pós-processamento das operações de criação, localização, atualização e exclusão. Por exemplo, antes de solicitar exclusão, a interface pode exibir um diálogo de confirmação de operação (pré-processamento de exclusão)

- **ComponenteIHCConcreto**
  - Define os campos de interface com o usuário para visualização e preenchimento das informações da entidade de negócio. Define também como os campos serão sincronizados com os atributos de *EntidadeConcreta*

- Define as características específicas de interação com usuário para a entidade de negócio: comandos para ações de negócio, pastas para manter relacionamentos com outras entidades e notificação de eventos de negócio para o *ControladorConcreto*
- Implementa o método *fabricarControlador*, criando *ControladorConcreto* para implementar *ControladorCRUD*

- **ControladorCRUD**
  - Fornece método *Factory* [2] *fabricarEntidade* para criação da entidade de negócio
  - Define as operações para tratamento de eventos: *tratarCriacao, tratarLocalizacao, tratarAtualizacao, tratarExclusao* que operam sobre a interface genérica (*Entidade*) implementada pelo objeto criado pelo método *fabricarEntidade*

- **ControladorConcreto**
  - Implementa método *Factory* [2] *fabricarEntidade* para criação da entidade de negócio (*EntidadeConcreta*)
  - Define as operações para tratamento de eventos de negócio

- **Entidade**
  - Define interface genérica para entidade de negócio com operações para criação, localização, atualização e exclusão
  - Pode ser composta por objetos que concentram a informação e objetos para acesso ao repositório de dados. No entanto, fornece interface única para o controlador. Três padrões podem ser combinados para gerar a entidade de negócio: *Data Access Object* [1], *ValueObject* [1] e Fachada [2]. Assim, utiliza-se um *ValueObject* representando a informação, um *Data Access Object* responsável por realizar a persistência do *ValueObject* e um objeto fachada para coordenar o *ValueObject* e o *Data Access Object*, expondo para os clientes uma interface de negócio unificada: atributos, métodos do ciclo de vida – criar, atualizar, excluir, consultarPorId, consultarTodos e métodos de negócio. A fachada seria a *Entidade* propriamente dita

- **EntidadeConcreta**
  - Implementa a interface de *Entidade* para os métodos de criação, localização, atualização e exclusão
  - Define os atributos que são sincronizados com os campos de *ComponenteIHCConcreto*
  - Define e implementa outros métodos de pesquisa
  - Define e implementa métodos de negócio utilizados por *ControladorConcreto*

2.7 Dinâmica

A seqüência de eventos abaixo descreve a dinâmica do padrão:

1. O usuário aciona algum comando CRUD em *ComponenteIHCConcreto;*
2. Se evento de atualização ou inclusão, a informação é sincronizada de *ComponenteIHCConcreto* para *EntidadeConcreta;*

3. O comportamento (métodos de tratamento) herdado de *ComponenteIHC* é executado, realizando algum pré-processamento e enviando uma notificação sobre o evento para o *ControladorConcreto*;
4. Em *ControladorConcreto*, o método adequado para tratamento do evento (*tratarCriacao*, *tratarLocalizacao*, *tratarAtualizacao*, ou *tratarExclusao*) é executado pelo código herdado de *ControladorCRUD*, executando o método CRUD adequado em *EntidadeConcreta* (através da interface de *Entidade*);
5. O método CRUD de *EntidadeConcreta* é executado e o controle retorna para o código herdado de *ComponenteIHC*, permitindo pós-processamento;
6. Se evento de localização, a informação é sincronizada de *EntidadeConcreta* para *ComponenteIHCConcreto*.

Os eventos de negócio específicos das classes concretas são tratados de maneira similar, utilizando, no entanto, código implementado nas classes concretas. A dinâmica do padrão é exemplificada na figura 2. A figura 2 apresenta o tratamento para o evento *criar*. Os outros eventos ocorrem de maneira análoga.

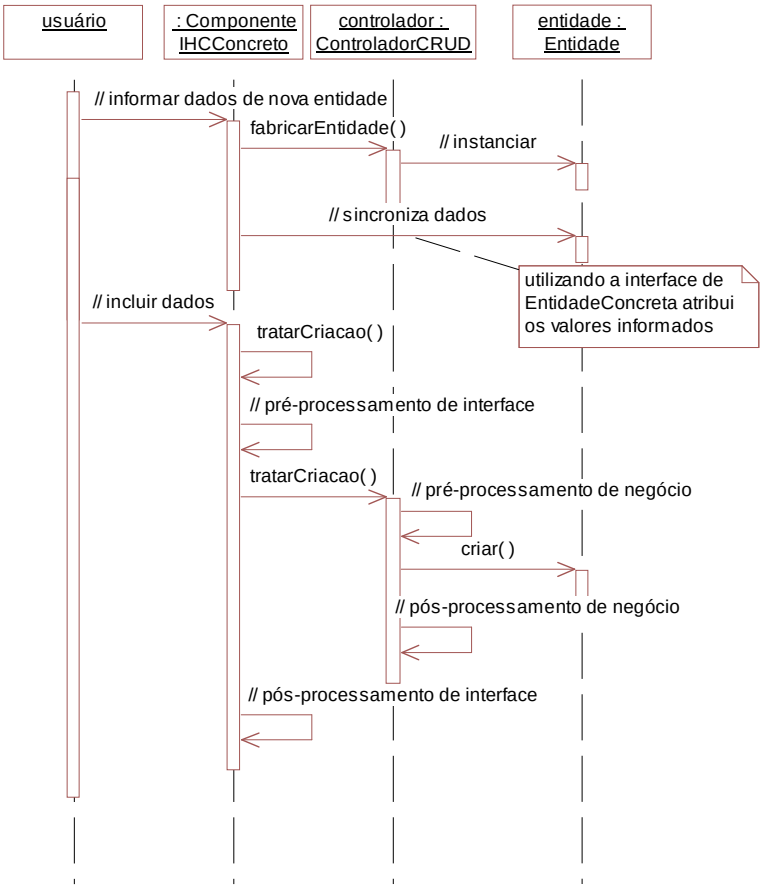


Figura 2: Diagrama de seqüência para o subfluxo de evento criar

2.8 Conseqüências

O padrão CRUD-MVC oferece os seguintes benefícios:

- A classe *ComponenteIHC* permite padronização e reuso de interface de usuário e de tratamento de eventos CRUD através de herança;

- Separação de código entre eventos CRUD e outros eventos de negócio;
- Permite a rápida extensão/manutenção de recursos de interface de usuário e tratamento de eventos através da centralização em classes abstratas que servem de base para o restante das classes da aplicação;
- Permite independência da interface de usuário com relação ao mecanismo de persistência através da interface de persistência de entidade;
- Permite a troca de componentes de IHC sem afetar o restante do código.

As seguintes desvantagens merecem ser mencionadas:

- Apesar de separar aspectos de apresentação, controle, regras de negócio e persistência, o número de classes da aplicação pode aumentar bastante com o uso do padrão. Se essa separação não é muito crítica na aplicação, as variantes 1 e 2 podem ser consideradas;
- Com a utilização do padrão a complexidade da solução torna-se maior, envolvendo relacionamento e comunicação entre vários objetos.

2.9 Padrões relacionados

- *Factory Method* [2]:
  - Utilizado na criação de objetos de Entidade e de Controle
- *MVC* (Modelo-Visão-Controle):
  - Utilizado para fornecer baixo acoplamento entre os componentes da aplicação
- *Data Access Object* [1], *ValueObject* [1] e Fachada [2]:
  - Esse três padrões são utilizados para construção de *Entidade*. Os padrões *Data Access Object* (DAO) e *ValueObject* são bastante utilizados em aplicações J2EE, mas podem ser utilizados em qualquer tipo de linguagem orientada a objetos

2.10 Variantes

De acordo com as características da linguagem utilizada na implementação do sistema ou requisitos não-funcionais, o padrão pode ser utilizado com as seguintes variantes:

1. Componente IHC e Controlador unificados

A unificação dessas duas classes aumenta o acoplamento, mas simplifica o projeto, reduz o número de classes e ainda continua fornecendo características como reutilização do tratamento de eventos e padronização dos componentes de interface e comportamento. Nesse caso, os tratamentos dos eventos com relação à interface e regras de negócio estariam juntos na classe *ComponenteIHC* e as respectivas classes concretas também seriam unificadas. Algumas regras de negócio do controle poderiam eventualmente passar para a entidade.

2. Simplificação da *Entidade*

A separação da entidade em fachada, *ValueObject* e DAO oferece uma boa separação entre operações de negócio, operações de acesso e leitura de dados e persistência, mas aumenta o número de classes do sistema. Se essa separação não for importante para a aplicação, pode-se utilizar apenas uma classe de entidade que tem os atributos com

seus métodos para leitura e escrita, métodos de negócio relacionados aos dados e código para persistência e comunicação com o repositório de dados todos juntos.

3. Utilização do padrão *Observer*
Algumas linguagens e *frameworks* (como Java *Swing*) trazem suporte natural para o padrão *Observer* [2]. Em outras linguagens a implementação do padrão pode ser mais complicada. No entanto, o uso do padrão *Observer* pode sofisticar a implementação do padrão MVC, diminuindo o acoplamento entre visão, controlador e modelo. Dois tipos de estratégias MVC podem ser adotados com o padrão *Observer*: Modelo Ativo ou Passivo. No modelo Ativo, a classe *Entidade* seria observada por *ComponenteIHC* através da interface *Observable* e qualquer mudança na *Entidade* levaria a uma notificação de *ComponenteIHC* através da interface *Observer*. Com o modelo passivo, o *ControladorCRUD* é que seria observado por *ComponenteIHC*, e quando fizesse alterações na *Entidade* notificaria *ComponenteIHC*.

2.11 Usos Conhecidos

Por motivos de confidencialidade, mais detalhes dos usos conhecidos abaixo não podem ser fornecidos.

- Sistema Imobiliário
- Sistema de controle de Processos Jurídicos
- Sistema de Administração de Recursos Humanos
- Sistema de CallCenter
- Sistemas Bancários

3. Padrão Registro com Busca

3.1 Contexto

Utilizado para as principais entidades de negócio, com muitos campos e/ou relacionamentos. Em geral, essas entidades são objetos complexos com muitos atributos e relacionamentos. O enfoque do cenário é de uma busca eficiente seguida de visualização e edição de uma entidade.

3.2 Problema

Como criar componentes de cadastro e manutenção de entidades de negócio complexas, atendendo a requisitos de interface humano-computador, permitindo a reutilização de interação com usuário e estrutura comuns aos vários tipos de entidades complexas existentes em uma aplicação de sistema de informação?

3.3 Forças

- Deve suportar reuso de características de interação com usuário, recorrentes na manutenção de entidades complexas
- Necessita de manipulação individual do objeto, mas com busca eficiente de entidades de negócio

- Deve permitir fácil visualização de dados e relacionamentos de uma entidade complexa
- Deve fornecer baixo acoplamento entre interfaces de usuário para entrada e consulta de dados, tratamento de eventos CRUD e entidade de negócio

3.4 Solução

Utilizar classe *Registro* para criação, atualização e exclusão de entidades, permitindo que uma única entidade seja visualizada e editada por vez. A classe *Registro* apresenta a informação e relacionamentos da entidade. Ela também fornece um mecanismo de localização eficiente, colaborando com a classe *Busca*. A classe *Busca* fornece critérios de pesquisa e lista de resultados. Além de buscar a entidade complexa, ela pode ser utilizada para buscar entidades para preenchimento de relacionamentos.

O padrão *Registro com Busca* utiliza o padrão *CRUD-MVC* (ver seção 2) para definir como as classes de modelo, visão e controle podem se relacionar. Dessa forma, o padrão mantem seu foco na interação com usuário para manutenção de entidades complexas e utiliza a estrutura de *CRUD-MVC* para fornecer características como extensibilidade e baixo acoplamento. As classes *Registro* e *Busca* funcionam de acordo com a classe *ComponenteIHC*. As classes *RegistroConcreta* e *BuscaConcreta* funcionam de acordo com a classe *ComponenteIHCConcreto*. As classes de controle e modelo são utilizadas também de acordo com o padrão *CRUD-MVC*. Assim, o padrão *Registro com Busca* apresenta uma solução de interação com usuário para cenários de manutenção de entidades complexas e utiliza o padrão *CRUD-MVC* para estruturar suas classes.

3.5 Estrutura

A Figura 3 apresenta o diagrama de classes do padrão e seus componentes. As classes de controle e modelo não são apresentadas e seguem o padrão *CRUD-MVC*.

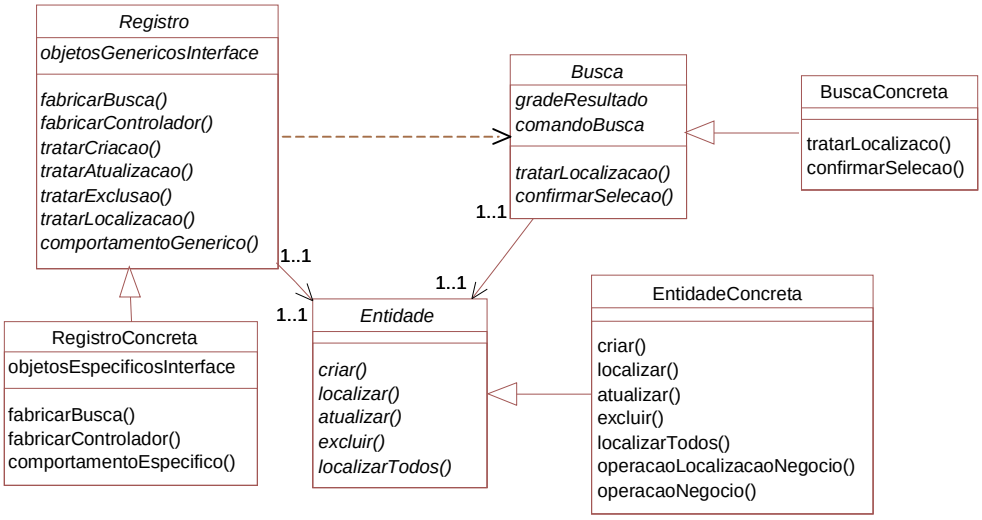


Figura 3 – Diagrama de Classes do Padrão Registro com Busca

3.6 Participantes

- **Registro**
  - Define as características de interação com usuário presente em toda classe para manutenção de entidades complexas. Fornece botão para acionar uma interface para realizar pesquisas elaboradas sobre a entidade complexa
  - Permite a edição, criação e exclusão de apenas uma entidade complexa por vez.
  - Fornece método *Factory* [2] *fabricarBusca* para criar uma instância de classe de *BuscaConcreta*. Após a criação o objeto registro exibe a tela de busca

- **Busca**
  - Fornece uma interface gráfica genérica para preenchimento de critérios de pesquisa e de apresentação de resultados
  - Fornece botão para ação de pesquisa e grade para exibição de resultados
  - Executa comunicação genérica com o objeto *Registro* para seleção e apresentação da entidade de negócio

- **RegistroConcreta**
  - Define os campos para visualização e preenchimento das informações da entidade de negócio complexa e como os campos serão sincronizados com os atributos de *EntidadeConcreta*
  - Define as características específicas de interação com usuário para a entidade de negócio: comandos para ações de negócio, pastas para manter relacionamentos com outras entidades e notificação de eventos de negócio para o *ControladorConcreto*. As pastas de entidades relacionadas indicam relacionamentos *um-para-muitos* da entidade de negócio para entidades dependentes
  - Implementa o método *fabricarControlador*, criando *ControladorConcreto* para implementar *ControladorCRUD*

- **BuscaConcreta**
  - Define os critérios concretos para a pesquisa
  - Define os campos que são apresentados na grade de resultados
  - Faz sincronização da coleção de entidades com a grade
  - Pode ser utilizada para buscar entidades para preenchimento de campos de relacionamentos da entidade complexa

3.7 Dinâmica

Os eventos de CRUD se comportam de acordo com o padrão *CRUD-MVC*. O detalhe ocorre no evento de localização. Para localizar e visualizar uma entidade, o seguinte fluxo é executado:

1. Usuário dispara comando de localização;
2. O código herdado de *Registro* é executado e o método *fabricarBusca* é executado, utilizando a implementação fornecida por *RegistroConcreta*. Em seguida à criação do objeto de busca, o mesmo é disponibilizada para o usuário;

3. O usuário informa os parâmetros da pesquisa e solicita a localização;
4. O objeto *BuscaConcreta* notifica o *ControladorConcreto* para tratamento do evento de pesquisa;
5. *O ControladorConcreto* utiliza um método de negócio de pesquisa, *localizar(params)*, da *EntidadeConcreta*, disponiblizando uma coleção de entidades para *BuscaConcreta;*
6. *BuscaConcreta* exibe alguns dados das entidades encontradas e o usuário seleciona a entidade que deseja visualizar ou editar;
7. *BuscaConcreta* disponibiliza a entidade selecionada para *RegistroConcreta* que exibe seus dados.

Em seguida, o usuário pode editar ou excluir a entidade. A criação pode ser feita a qualquer momento. A dinâmica do padrão é exemplificada na figura 4.

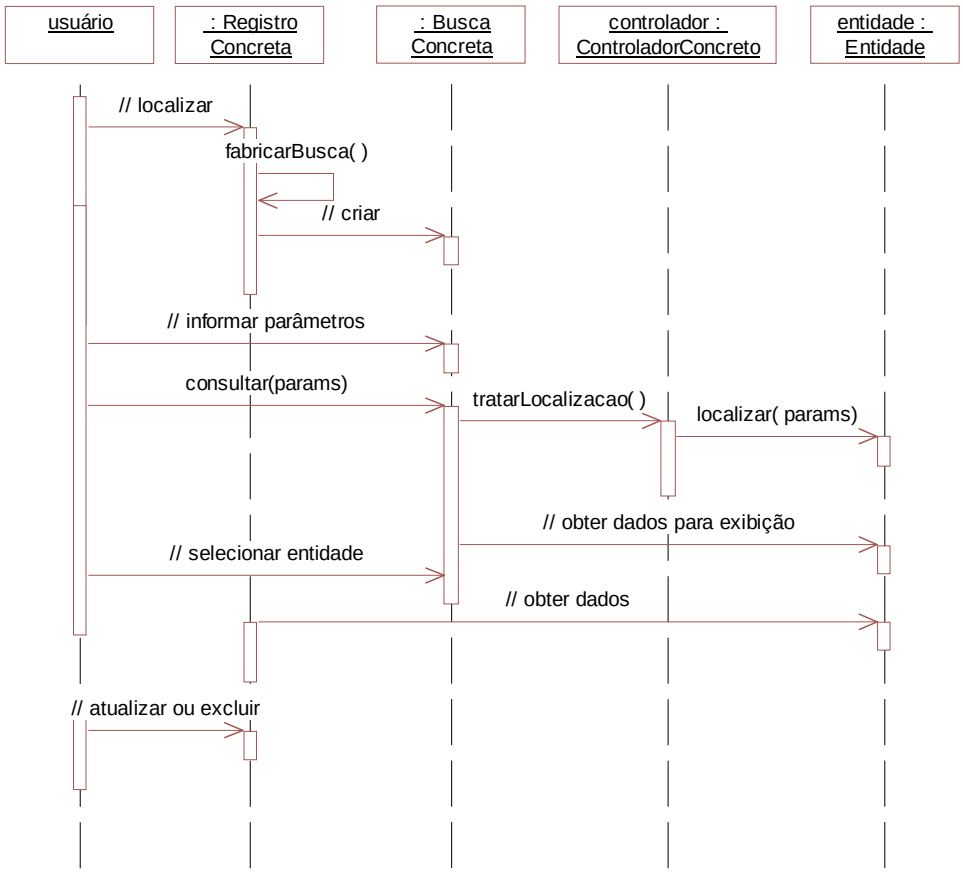


Figura 4: Diagrama de seqüência do padrão Registro com Busca

3.8 Conseqüências

O padrão Registro com Busca oferece os seguintes benefícios:

- Fornece mecanismo eficiente de busca e visualização/edição de uma única entidade, aumentando a eficiência do sistema. Economiza recursos do sistema evitando alocação em memória de coleções de entidades complexas que consomem muitos recursos;

- Permite maior usabilidade do sistema, fornecendo para o usuário uma interface adequada para visualização e edição de entidade complexa;
- O reuso de *Registro* através de herança fornece interface de usuário padronizada para registro de entidades de negócio complexas;
- Permite reuso de tratamento de eventos CRUD e interação com objeto de busca.

As seguintes desvantagens merecem ser mencionadas:

- Apesar de separar aspectos de apresentação, controle, regras de negócio e persistência, o número de classes da aplicação pode aumentar bastante com o uso do padrão;
- Com a utilização do padrão a complexidade da solução torna-se maior, envolvendo relacionamento e comunicação entre vários objetos.

3.9 Padrões relacionados

- *Factory Method* [2]:
  - Utilizado na criação de objetos de Entidade e de Busca
- *CRUD-MVC* (ver seção 2):
  - Utilizado para definir interação entre as classes do modelo, visão e controle

3.10 Variantes

O padrão *Registro com Busca* também pode ser utilizado sem a estrutura do *CRUD-MVC*, considerando apenas seu principal foco, a interação com usuário para manutenção de entidades complexas. Essa situação pode ser aplicada quando a aplicação não possui fortes requisitos de extensibilidade e baixo acoplamento.

3.11 Usos Conhecidos

- Ver *CRUD-MVC*

4. Agradecimentos

Este trabalho foi suportado pelo Instituto Atlântico.

Os autores agradecem aos responsáveis pelo processo de revisão, em especial ao Márcio de Oliveira Barros, pelas contribuições realizadas no aprimoramento do artigo.

5. Referências

[1] D. Alur, J. Crupi, D. Malks. *Core J2EE Patterns As melhores práticas e estratégias de design*, Editora Campus, 2002.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Michael. *Pattern-Oriented Software Architecture*, John Wiley & Sons, 2001.

Padrões Arquiteturais e de Projeto para a Modelagem de Usuários baseada em Agentes

Ismênia Ribeiro de Oliveira, Rosario Girardi

Universidade Federal Maranhão (UFMA) – GESEC/DEINF
Av. dos Portugueses, s/n, Campus do Bacanga – São Luís – MA – Brasil
ism_oliveira@yahoo.com.br, rgirardi@deinf.ufma.br

Abstract

Most software systems do not have the ability to satisfy heterogeneous user needs. Each user has particular knowledge level, needs, abilities and preferences. In that context, there is a need for building systems adaptable to each user type or group of users with common characteristics. Because of its advantages to approach software complexity, the agent-based development paradigm is being used for the development of applications that can adapt to different types of users. Pattern reuse in this development paradigm makes possible an optimization in the time and cost of development and an increase in the quality of software. This article approaches user modeling as a mean of producing adaptation effects in a system and proposes solutions for frequent user modeling problems in the form of agent-based architectural and detailed software patterns.

Resumo

A maioria dos sistemas de software não tem a habilidade para satisfazer as necessidades heterogêneas dos seus usuários. Cada usuário tem níveis de conhecimento, necessidades, habilidades e preferências bastante variadas. Nesse contexto surge a necessidade de construir sistemas que se adaptem a cada tipo de usuário ou grupo de usuários com características comuns. Por causa de suas vantagens para abordar a complexidade do software, o paradigma de desenvolvimento de software orientado a agentes está sendo utilizado para o desenvolvimento de aplicações que se adaptam a diferentes tipos de usuários. A reutilização de padrões no desenvolvimento de sistemas multiagente possibilita uma otimização no tempo e no custo do desenvolvimento e um aumento na qualidade do software. Este artigo aborda a modelagem de usuários como forma de produzir efeitos de adaptação em um sistema e propõe soluções recorrentes para problemas recorrentes de projetos de modelagem de usuários na forma de padrões arquiteturais e de projeto baseados em agentes.

1. Introdução

É possível adaptar sistemas de acordo com o conhecimento ou experiência do usuário, histórico de ações anteriores, propriedades cognitivas (estilo de aprendizado, personalidade), objetivos e planos (intenções) do usuário, seus interesses e preferências.

Um importante componente dos sistemas interativos adaptáveis é a habilidade para modelar os usuários do sistema. A modelagem de usuários é essencial em sistemas que tentam adaptar seu comportamento aos usuários para interagir de forma mais inteligente e individualizada.

Um grande desafio encontrado na modelagem de usuários é tratar a heterogeneidade dos usuários de software. Uma solução para este problema seria projetar e implementar a modelagem dos usuários e a adaptabilidade dos sistemas através de utilização da abordagem de agentes. Seriam usados agentes especializados e

individualizados para cada usuário ou grupos de usuários, como agentes de interface agentes de modelagem e agentes de adaptação.

Um agente é uma entidade autônoma, que possui um sistema interno de tomada de decisões, agindo sobre o mundo e sobre outros agentes que o rodeiam e por fim, que é capaz de funcionar sem a necessidade de algo ou alguém para guiá-lo. A analogia feita com agentes no mundo real nos leva a conceituar um agente como uma entidade ativa, sempre ao lado do usuário e que possui conhecimentos específicos sobre um determinado domínio. De posse de bases de conhecimento e de mecanismos de raciocínio, os agentes devem ser capazes de reconhecer situações em que devem se ativar, sem que o usuário perceba, ou seja, de forma transparente ao usuário [32].

Segundo Wooldridge e Jennings [32], a construção e a especificação da estrutura e funcionamento de um agente genérico pode ser realizada segundo três tipos de arquiteturas:

- **Arquiteturas deliberativas:** segue a abordagem clássica da Inteligência Artificial, onde os agentes contêm um modelo simbólico do mundo, explicitamente representado, e cujas decisões (ações) são tomadas via raciocínio lógico, baseadas em casamento de padrões e manipulações simbólicas;
- **Arquiteturas reativas**: a arquitetura reativa é aquela que não inclui nenhum tipo de modelo central e simbólico do mundo e não utiliza raciocínio complexo e simbólico. Baseia-se na proposta de que um agente pode desenvolver inteligência a partir de interações com seu ambiente, não necessitando de um modelo pré-estabelecido;
- **Arquiteturas híbridas:** a arquitetura híbrida mistura componentes das arquiteturas deliberativas e reativas com o objetivo de torná-la mais adequada e funcional para a construção de agentes.

Essas considerações feitas sobre agentes nos levam a crer que a modelagem de usuários seria melhorada e facilitada caso fosse implementada com agentes inteligentes, além da melhora na eficiência e eficácia do sistema, através dos mecanismos de adaptação.

Este artigo aborda a modelagem de usuário como forma de produzir efeitos de adaptação em um sistema. A partir desse estudo são propostas soluções recorrentes para problemas recorrentes de projetos de modelagem de usuários, ou seja, padrões arquiteturais e de projeto especializados na modelagem de usuário e adaptação de sistemas.

A **Figura 1** mostra, em uma rede semântica, o relacionamento entre os padrões para esses sistemas. O padrão *Sociedade multiagente para sistemas adaptativos* justifica a escolha da abordagem de agentes para sistemas adaptativos. A organização dos agentes no sistema multiagente é descrita pelo padrão arquitetural *Camadas multiagente para sistemas adaptativos/adaptáveis* que distribui os agentes em camadas conforme as tarefas que serão executadas pelos mesmos. O detalhamento dos agentes que compõem as camadas é descrito pelos padrões *Interface, Modelagem e Adaptação*. Os padrões *Modelagem* e *Adaptação,* por sua vez, são derivados do padrão *Deliberativo.* Já o padrão *Interface* é derivado do padrão *Reativo*.

Neste artigo são apresentados os padrões: *Sociedade multiagente para sistemas adaptativos, Camadas multiagente para sistemas adaptativos, Deliberativo, Reativo* e *Modelagem*. A Tabela 1 apresenta, de forma sucinta os padrões propostos, classificando-os segundo a dependência ou independência do domínio da aplicação e segundo o tipo de padrão.

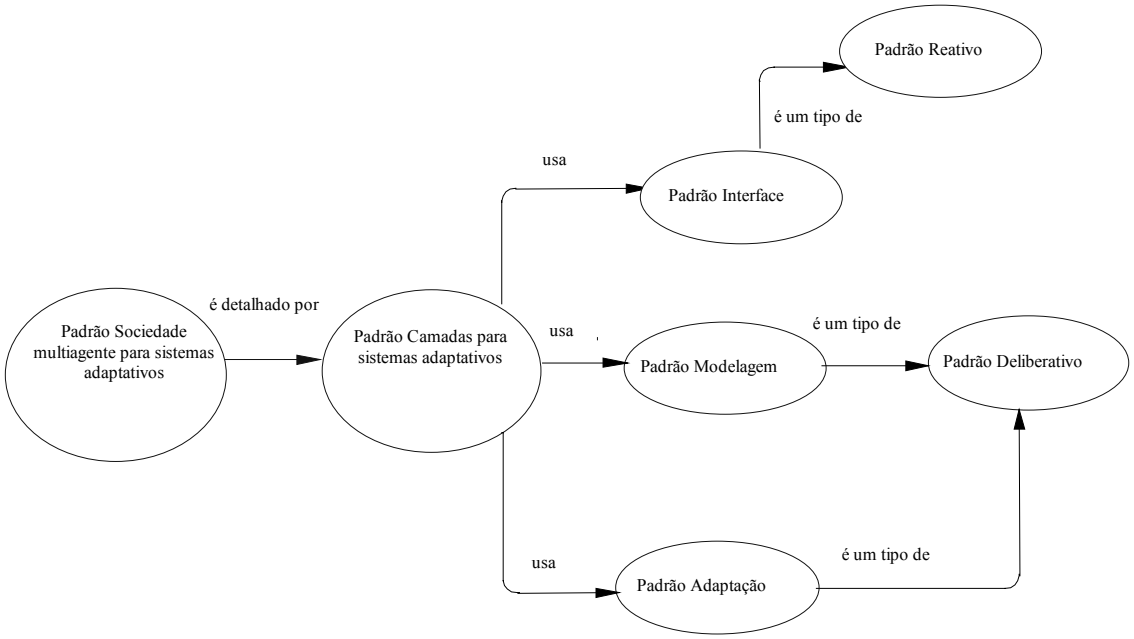


Figura 1 Rede semântica mostrando o relacionamento entre os padrões propostos

Tabela 1 Padrões para o projeto de sistemas adaptativos/adaptáveis

Padrões baseados em Agentes				
Classificação		**Problema**	**Solução**	**Padrão**
Padrões Dependentes de Domínio	**Padrão Conceitual**	Como estruturar uma comunidade de agentes para construir modelos de usuários e se adaptar a diferentes tipos de usuários ou grupos de usuários?	O padrão agrupa os agentes em três camadas, de acordo com as principais funcionalidades exercidas por sistemas adaptativos/adaptáveis: camada de interface; camada de modelagem e camada de adaptação,	Camadas Multiagente para Sistemas Adaptativos/Adaptáveis
	Padrão Arquitetural	Como construir e organizar sistemas que modelam características e preferências de usuários de forma a adaptar-se às suas necessidades?	Projetar e implementar sistemas que modelam características e preferências de usuários e produzem adaptação através de utilização da abordagem de agentes.	Sociedade multiagente para Sistemas Adaptativos/adaptáveis
	Padrões de projeto detalhado	Como criar e manter modelos de usuários em sistemas adaptativos?	A solução envolve a criação de um agente responsável por construir e manter modelos de usuários.	Modelagem
Padrões Independentes de Domínio		Como projetar um agente para que possa raciocinar sobre um problema de forma que o possibilite atingir metas pró-ativamente?	O agente deve ser do tipo deliberativo, ou seja, ele deve possuir modelos simbólicos internos de do ambiente no qual ele está inserido e de si mesmo e raciocinar sobre esses modelos de forma que o possibilite criar um plano de ações para atingir suas metas.	Deliberativo
		Como projetar um agente para apenas reagir a estímulos do ambiente no qual está inserido?	O agente deve ser do tipo reativo, ou seja, ele não tem um modelo interno do ambiente no qual está inserido, o agente deve agir usando um comportamento estímulo/resposta.	Reativo

2. Padrão Sociedade multiagente para sistemas Adaptativos

Contexto

Os sistemas onde o usuário pode iniciar, propor, selecionar e produzir a adaptação ou também deixar que o sistema execute algumas dessas funções, são chamados de sistemas adaptáveis. Os sistemas que executam todas as funções acima citadas, de forma automática, são chamados de sistemas adaptativos. O objetivo geral de tais sistemas é prover seus usuários com conteúdo atualizado, subjetivamente interessante, com informação pertinente, num tamanho e profundidade adequados ao contexto e em correspondência direta com o *modelo do usuário*.

O modelo de usuário é uma fonte de conhecimento que contém informações, explícitas ou implicitamente adquiridas, de todos os aspectos relevantes ao usuário para serem utilizados no processo de adaptação de uma aplicação de software.

As principais áreas de aplicação dos sistemas adaptativos são:

- Comércio eletrônico [31];
- Interfaces adaptativas [20];
- Turismo [13];
- Recuperação de informação [23] [22];
- Educação [3].

Problema

Como construir e organizar sistemas que modelam características e preferências de usuários de forma a adaptar-se às suas necessidades?

Forças

As características de autonomia, habilidade social, reatividade, pró-atividade e capacidade de aprendizagem, próprias dos agentes, provêm mecanismos apropriados para atingir os objetivos dos sistemas adaptativos/adaptáveis que utilizam técnicas de modelagem de usuários. Freqüentemente, os agentes estão aptos a adaptar-se aos seus ambientes e exibem algum grau de inteligência, apesar dessas características não serem obrigatórias.

As características dos agentes podem ser associadas com os principais objetivos da utilização de modelos de usuários em sistemas adaptativos, refletindo assim as razões para a utilização da tecnologia de agentes em tais sistemas, como pode ser visto na Tabela 2.

Tabela 2 Padrões para o projeto de sistemas adaptativos/adaptáveis

Objetivos dos sistemas adaptativos/adaptáveis	Características dos agentes
Melhorar a eficiência e eficácia da interação (facilitar e acelerar interações)	**Reatividade** – através da interação com o ambiente, os agentes reagem rapidamente a mudanças no ambiente e apresentam ao usuário o que ele quer ver de forma a refletir o seu perfil.
Tornar os sistemas complexos mais usáveis	**Habilidade social**- os agentes cooperam e interagem com outros agentes para resolver tarefas complexas tornando os sistemas mais simples do ponto de vista do usuário
Ajudar o usuário a encontrar informação	**Autonomia e pró-atividade**– o usuário pode delegar tarefas (objetivos) para que os agentes de software as executem autonomamente.
Prever o comportamento futuro do usuário	**Capacidade de aprendizagem**- os agentes podem aprender com o comportamento do usuário e fazer inferências sobre o seu comportamento esperado.

Solução

Modelar a aplicação como uma sociedade de agentes. Seriam usados agentes especializados para cada usuário ou grupos de usuários, como agente de interface, agente de modelagem e agente de adaptação.

O processo de modelagem de usuários e o processo de personalização de aplicações podem ser divididos em três tarefas maiores que podem ser executadas por diferentes componentes do sistema:

- A *aquisição* consiste na identificação de informações sobre usuários, como características, preferências, necessidades e fazer essas informações acessíveis ao componente de adaptação da aplicação;
- A *representação* consiste na representação dos conteúdos dos modelos de usuários e modelos de adaptação, apropriadamente, em um sistema formal, permitindo o seu acesso e posterior processamento e formular suposições sobre usuários e/ou grupos de usuários, seus comportamentos e seu ambiente;
- A *produção* consiste na geração da adaptação do conteúdo, da apresentação, da modalidade e da estrutura, baseados nos modelos de usuários, nos modelos de adaptação e nos modelos do domínio.

As tarefas de aquisição e produção podem ser atribuídas ao agente de interface, a tarefa de representação e manutenção de modelos de usuários pode ser atribuída ao agente de modelagem e o agente de adaptação pode ser responsável pela representação e manutenção de modelos de adaptação, como pode ser visualizado na Figura 2. Durante o processo de modelagem de usuários, de acordo com a Figura 2., o agente de interface colhe dados sobre o usuário; esses dados são processados e representados em um modelo de usuário pelo agente de modelagem. Na execução do processo de adaptação, o agente de adaptação produz modelos de adaptação baseados no modelo de usuário, esses modelos são utilizados pelo agente de interface para personalizar a aplicação.

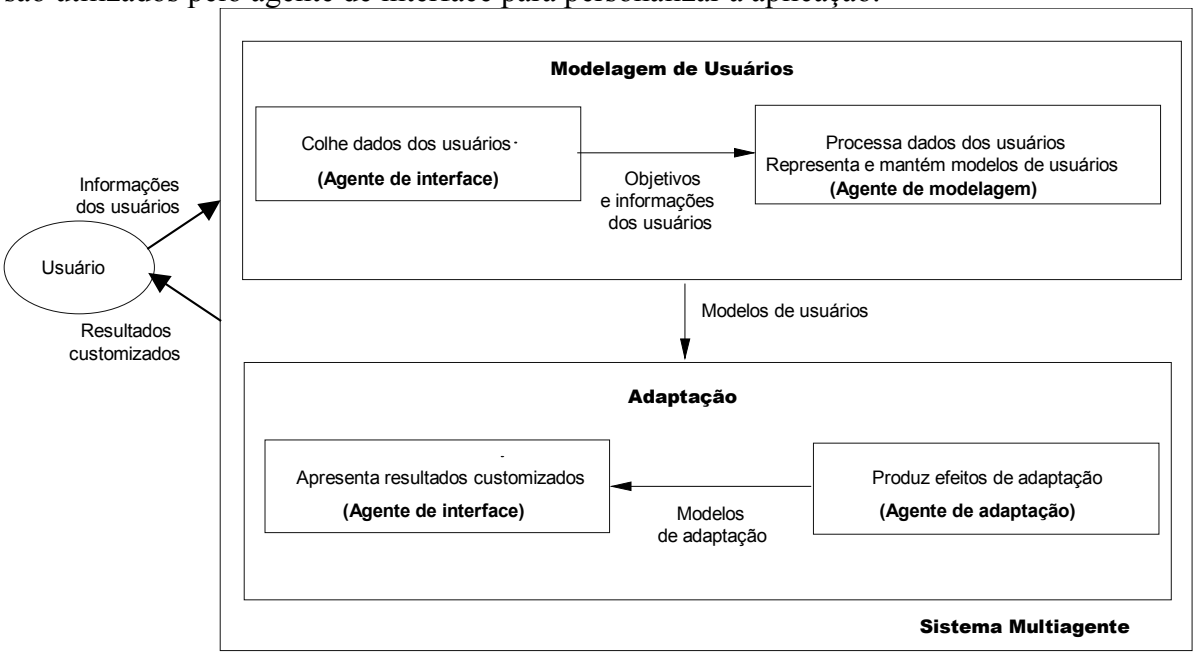


Figura 2 O processo da modelagem de usuários e adaptação de sistemas utilizando agentes

Padrões relacionados

A solução arquitetural, descrita neste sistema de padrões, para estruturar os agentes dos sistemas adaptativos/adaptáveis é descrita no padrão chamado Camadas multiagente para sistemas adaptativos/adaptáveis.

Usos conhecidos

Bylund [6] e Baldassin [2] propõem o uso de agentes para oferecer serviços personalizados aos usuários, distribuindo tarefas entre esses agentes

3. Camadas Multiagente para Sistemas Adaptativos/adaptáveis

Contexto

Nos sistemas adaptativos/adaptáveis, baseados na tecnologia de agentes, é preciso projetar soluções para organizar os agentes de maneira que possam satisfazer as necessidades e preferências de seus usuários.

Problema

Como estruturar um sistema multiagente apto a adaptar-se às necessidades de diferentes tipos de usuários ou grupos de usuários?

Forças

A organização dos agentes em camadas, segundo a similaridade de responsabilidades tem todas as vantagens (flexibilidade, facilidade de manutenção, de entendimento e de extensão) descritas pelo padrão camadas [5].

Solução

A solução (Figura 3), baseada no uso de técnicas de adaptação e modelagem de usuários, envolve a definição de um critério para a divisão das responsabilidades entre agentes distribuídos em camadas, de acordo com o padrão camadas descrito por Silva Júnior [29], uma extensão do padrão Camada proposto por Buschmann [5]. Os sistemas adaptativos/adaptáveis que usam a modelagem de usuários são compostos basicamente por uma camada de interface, uma camada de modelagem e uma camada de adaptação, sendo que podem ser adicionadas novas camadas de acordo com o tipo de serviço que o sistema adaptativo/adaptável irá prover**.**

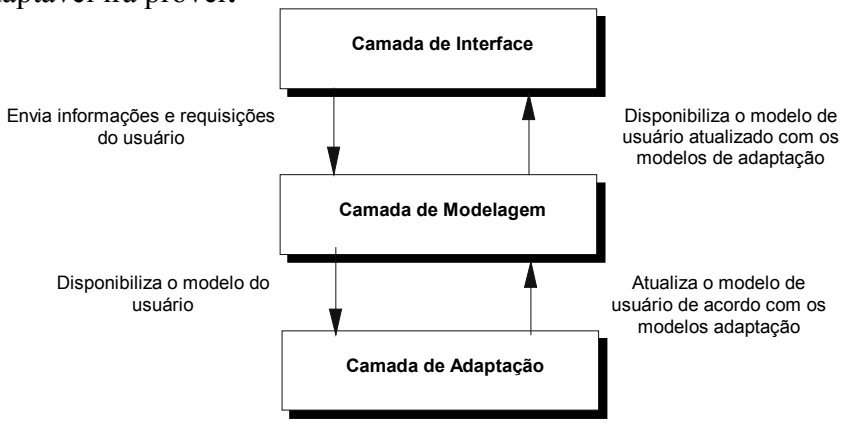


Figura 3. Padrão camadas multiagente para a modelagem de usuários

Um grupo de agentes está associado a cada camada. Cada camada é estruturada como segue.

Camada de interface

A camada de interface consiste em um grupo de agentes de interface onde cada agente está associado a um usuário ou grupo de usuários com necessidades similares para auxiliá-lo(s) em suas tarefas. As principais responsabilidades dos agentes dessa camada são:

- ·Apresentar uma interface inicial, caso o usuário ainda não possua um modelo de usuário, ou personalizada, baseando-se no modelo de usuário corrente;
- ·Receber requisições dos usuários;
- ·Colher informações sobre usuários, tais como características dos usuários, comportamento durante o uso do computador ou do sistema, através do monitoramento do uso do computador ou requisitando essas informações diretamente ao usuário;

- Caso seja adotado um sistema de retroalimentação, fazer perguntas a respeito dos resultados para possíveis melhoras na eficácia do sistema;
- ·Apresentar informações aos usuários de forma personalizada (respostas e explicações).

Camada de modelagem

A camada de modelagem consiste em um grupo de agentes de modelagem. Cada agente de modelagem provê serviços para um agente de interface. As principais responsabilidades dos agentes dessa camada são:

- Processar a informação provida pelo usuário, através de um agente de interface e atualizar ou criar modelos de usuários, se eles ainda não existirem, de acordo com esta informação;
- ·Formular metas ou inferir suposições sobre usuários ou grupo de usuários com base nos modelos de usuários;
- ·Reformular consultas, tarefas e metas dos usuários de acordo com os modelos de usuários e disponibilizá-los para o agente de adaptação correspondente.

Camada de adaptação

A camada de adaptação consiste em um conjunto de agentes de adaptação. Cada agente de adaptação provê serviços para o agente de modelagem. As principais responsabilidades dos agentes dessa camada são as seguintes:

- Construir, representar e manter modelos de adaptação, de acordo com os modelos de usuários;
- ·Atualizar os modelos de usuários de acordo com os modelos de adaptação.

Padrões relacionados

- Para o projeto dos agentes da camada de interface pode ser utilizado padrão Interface.
- Para o projeto dos agentes da camada de modelagem pode ser utilizado o padrão Modelagem.
- Para o projeto dos agentes da camada de adaptação pode ser utilizado o padrão Adaptação.

Usos conhecidos

A organização em camadas é típica das arquiteturas multiagente para filtragem e recuperação de informação, tais como as arquiteturas RETSINA [28], AMALTHAEA [22] e ABARFI [9], bem como as arquiteturas de sistemas Web Adaptativos, como a arquitetura genérica descrita por Sharma [27]. Este padrão é inspirado nas soluções arquiteturais propostas por tais sistemas.

4. Padrão Modelagem

Contexto

Nos sistemas adaptativos/adaptáveis, baseados na tecnologia de agentes, é preciso projetar soluções para representar os usuários através da construção de modelos de usuários que especificam suas características, suas preferências e os seus interesses.

Problema

Como criar e manter modelos de usuários que serão utilizados como referência para que a aplicação ofereça serviços personalizados aos seus usuários?

Forças

Os usuários diferem em seus níveis de conhecimento, necessidades e preferências. O tema da adaptabilidade é fornecer informação sob medida de acordo com as características dos usuários, por exemplo, adaptando o estilo de apresentação durante a interação com o usuário. Qualquer abordagem para apresentar informação sob medida envolve a criação e manutenção de modelos de usuários que devem estar aptos a refletir as mudanças nas características desses usuários no decorrer do tempo.

A Modelagem de usuários é necessária, pois através dela se pode sintetizar as características e habilidades de usuários ou grupo de usuários com o fim de oferecer serviços personalizados.

Solução

A solução envolve a criação de um agente de modelagem do tipo deliberativo, responsável por de construir e manter modelos de usuários com base nas informações fornecidas por um agente de interface através da interação com o usuário.

O agente de modelagem recebe informações sobre os usuários do agente de interface, e os classifica como pertencentes a um ou mais grupos, chamados de estereótipos; introduz as características específicas de tais grupos no modelo individual de cada usuário; faz inferências de novas hipóteses sobre os usuários com base nos fatos iniciais.

Algumas das técnicas mais comuns para a representação de modelos de usuários são: redes bayesianas [21], lógica fuzzy, redes neurais, redes de Petri [18], representação baseada em lógica [25], estereótipos [34], representação vetorial [22] e ontologias [11].

Além da representação dos modelos de usuários, o agente de modelagem é responsável pela manutenção da consistência dos modelos verificando as novas informações fornecidas pelo agente de interface e/ou pelo agente de adaptação, comparando com informações anteriores. Dentre as técnicas mais comuns para fazer a manutenção de modelos de usuários podemos destacar a lógica fuzzy (teoria de conjuntos difusos), a teoria de probabilidades, algoritmos genéticos [18].

Padrões relacionados

O agente de modelagem pode ser definido como um agente do tipo deliberativo que tem suas características descritas pelo padrão Deliberativo.

Os modelos de usuários são utilizados pelo agente de adaptação para construir modelos de adaptação para cada usuário ou grupo de usuários. As características do agente de adaptação são descritas pelo padrão Adaptação

Usos conhecidos

Este padrão está sendo utilizado na construção de agentes de modelagem que utilizam técnicas de aquisição implícitas do modelo baseadas em algoritmos genéticos [30] no contexto dos projetos de pesquisa MaAE e JURÍDICAS [15].

5. Padrão Deliberativo

Contexto

Em sistemas multiagente, onde existem tarefas complexas, é preciso projetar agentes que estejam aptos a executar essas tarefas, exibindo comportamento direcionado a metas, tomando iniciativas e decisões através de raciocínio e/ou adaptando-se às mudanças no ambiente.

Problema

Como projetar um agente para que possa raciocinar sobre um problema de forma que o possibilite atingir metas pró-ativamente dentro do contexto no qual está inserido?

Forças

Tendo conhecimento, sobre o seu ambiente e usando mecanismos de raciocínio para encontrar uma solução adequada, agentes deliberativos são capazes de executar tarefas complexas.

Solução

O agente deve ser do tipo deliberativo, ou seja, ele deve possuir modelos simbólicos internos de raciocínio do ambiente no qual ele está inserido e de si mesmo. Ele raciocina sobre esses modelos para criar um plano que lhe permite atingir suas metas. O padrão deliberativo é estruturado por cinco módulos organizados verticalmente e horizontalmente: módulo de comunicação, módulo de ação, módulo de raciocínio, módulo de sensores e módulo de conhecimento. A estrutura do padrão deliberativo está representada na Figura 4.

Através do *módulo sensores*, o agente percebe do ambiente e atualiza seu conhecimento com estas percepções para refletirem o estado corrente do ambiente. Percepções são informações que o agente pode receber do mundo real, informações de recursos e interações com outros usuários. A camada sensitiva também recebe mensagens de outros agentes da sociedade.

No *módulo de raciocínio*, baseado nos seus conhecimentos, o agente elabora metas e planos para atingir essas metas.

No *módulo de ação* o agente atua no ambiente executando as ações do plano gerado pelo módulo de raciocínio. A ação executada é armazenada no módulo de conhecimento para refletir os efeitos das ações no ambiente. O módulo de ação serve o módulo de comunicação. Durante a execução de um plano, um agente pode determinar a necessidade de cooperação com outros agentes, requerendo informações ou pedindo para executar ações. Esta cooperação é suportada pelo módulo de comunicação.

No módulo de comunicação, o agente envia e processa mensagens.

O *módulo de conhecimento* é um repositório onde o agente armazena informações sobre o estado corrente do ambiente através do histórico das percepções do agente e ações executadas. Esta camada contém recursos e comportamento do agente e também conhecimento sobre as habilidades dos outros agentes da sociedade.

Os módulos de sensores, de raciocínio, de ação e de comunicação têm acesso direto ao módulo de conhecimento para que possam executar suas respectivas funcionalidades.

Cada módulo pode ser construído através da composição de um grupo de padrões para prover a funcionalidade de cada módulo.

O ciclo de execução do agente consiste em: no módulo de sensores, a percepção é usada para atualizar o conhecimento do agente. No módulo de raciocínio, este conhecimento é usado para determinar, dinamicamente, as metas correntes do agente para as quais um plano de ações foi construído, de acordo com as capacidades e comportamento do agente no módulo de conhecimento. O plano é então executado no módulo de ação. Quando um plano necessita de ações que o agente não está apto a executar porque elas não

fazem parte das suas capacidades e comportamentos, o módulo de comunicação determina a necessidade de cooperação com outros agentes. Então mensagens são enviadas para agentes capazes de ajudá-lo a cumprir o plano; as mensagens de outros agentes são recebidas através do módulo de sensores, depois então o ciclo completo é repetido. O módulo de conhecimento é atualizado a cada novo ciclo pelos demais módulos.

1- Recebe mensagens e percebe mudanças no ambiente

2- Registra percepções e mensagens recebidas

3- Percepções ou mensagens recebidas

4- Consulta capacidades do agente

5- Plano de ações

6- Ações não suportadas pelo agente

7- Registra e consulta possíveis colaborações com outros agentes

8- Envia mensagens para outros agentes

9- Atua no ambiente

10- Registra ações realizadas

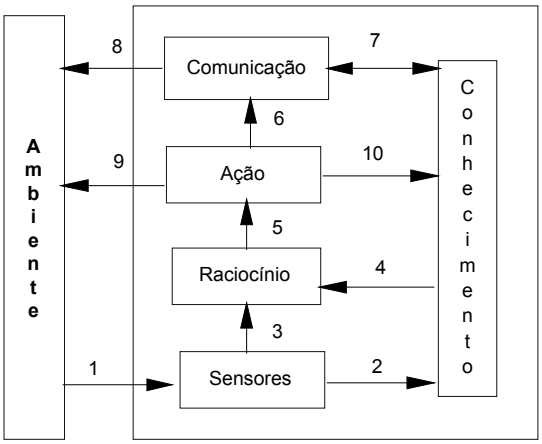


Figura 4 Estrutura do Padrão Deliberativo

Padrões relacionados

Os agentes deliberativos são amplamente utilizados em arquiteturas multiagente que utilizam raciocínio complexo [32], Kendall [17] propõe um padrão alternativo, chamado Agente deliberativo que difere do padão proposto neste trabalho principalmente em dois aspectos. Primeiro, o padrão agente deliberativo sugerido por Kendall acrescenta as camadas de mobilidade e tradução, funcionalidades das quais fazemos abstração atualmente no nosso trabalho. Segundo, o nosso padrão propõe a base de conhecimento como camada que interage verticalmente com todos os outros módulos do agente deliberativo

Usos conhecidos

Os ambientes de desenvolvimento de sistemas multiagente, como AgentBuilder [1] e Zeus [10] [33] e JADE [16] usam diferentes arquiteturas genéricas para a construção de agentes deliberativos. Eles provêm funcionalidades semelhantes às providas pelas camadas do padrão deliberativo que nós propusemos. Porém, a forma como os módulos interagem é diferente. No agente genérico do AgentBuilder e JADE/JESS, os módulos interagem diretamente sem considerar uma hierarquia de camadas. O agente genérico do Zeus é organizado em camadas, no entanto o conhecimento do agente está distribuído em várias camadas.

6. Padrão Reativo

Contexto

Em sistemas multiagente, onde existem tarefas que demandam respostas rápidas durante a sua execução, é necessário projetar agentes que estejam aptos a reagir a estímulos do seu ambiente, mas que não utilizem raciocínio complexo e que não tenham conhecimento sobre o ambiente no qual estão inseridos.

Problema

Como projetar um agente para apenas reagir a estímulos do ambiente no qual está inserido ou a mensagens de outros agentes quando ele não tem conhecimento sobre esse ambiente e nem pode aprender a partir dele?

Forças

O uso deste padrão é indicado em aplicações que demandam respostas rápidas requerendo que os agentes reajam a estímulos e atuem no seu ambiente. Mesmo não exibindo comportamento racional, os agentes estruturados de acordo com o padrão reativo podem exibir algum tipo de comportamento inteligente através da interação com outros agentes em sistemas multiagente.

Solução

O agente deve ser do tipo reativo, ou seja, ele não deve ter um modelo interno do ambiente no qual está inserido, o agente deve agir usando um comportamento estímulo/resposta de acordo com o estado corrente do ambiente ao qual está integrado. A solução do padrão reativo estrutura um agente em quatro módulos: comunicação, ação, regras e sensores. A estrutura do padrão reativo está representada na Figura 5.

1- Recebe mensagens e percebe mudanças no ambiente

2-Seleciona regra de acordo com as percepções

3- Ações para serem executadas

4- Ações não suportadas pelo agente

5- Envia mensagens para outros agentes

6- Atua no ambiente

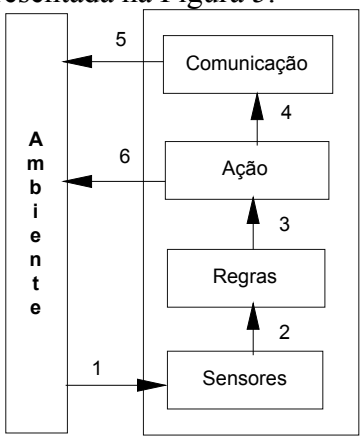


Figura 5 Estrutura do Padrão Reativo

Através do *módulo de sensores*, o agente percebe o ambiente. Percepções são informações que o agente pode receber do mundo real, informações de recursos e interações com usuários. O módulo de sensores também recebe mensagens de outros agentes da sociedade. O módulo de sensores serve o módulo de regras.

O *módulo de regras* é composto de um conjunto de regras de condição-ação da forma "**se** <condição> **então** <ação>". Baseado nos estímulos percebidos pelo módulo de sensores uma regra condição-ação é escolhida. O módulo de regras serve o módulo de ação.

No *módulo de ação*, uma ação é selecionada para ser executada.

Durante a execução de uma ação, o agente determina a necessidade de cooperação com outros agentes para requerer informações ou pedir a execução de uma ação, o módulo de comunicação suporta a cooperação com outros agentes.

No *módulo de comunicação*, o agente envia e processa mensagens.

Cada módulo pode ser contruído através da composiçao de um grupo de padrões para prover a funcionalidade de cada módulo.

O ciclo de execução do agente consiste em: no módulo de sensores, um estímulo é percebido. No módulo de regras, este estímulo é usado para determinar a ação que o agente irá executar no módulo de ação. Se para a realização da ação selecionada o agente necessita cooperar com outros agentes, o módulo de comunicação é ativado. Então mensagens são enviadas para os agentes capazes de cooperar; as respostas dos outros agentes são recebidas através do módulo de sensores, então o ciclo completo se repete.

Padrões relacionados

Kendall [17] propõe um um padrão alternativo, chamado Agente reativo, para estruturar agentes reativos similar ao desenvolvido neste trabalho, sendo que o o padrão sugerido por Kendall acrescenta as camadas de mobilidade e tradução, funcionalidades das quais fazemos abstração atualmente no nosso trabalho.

Usos conhecidos

Este tipo de solução para a estruturação dos agentes foi originalmente proposta na arquitetura de classificação de Brooks [4]. Kulkarni [19] criou um sistema de comportamento reativo chamado ReBa que pode disparar diferentes ações em reposta a eventos de dispositivos. Dentre os ambientes de desenvolvimento de software e frameworks multiagente que temos utilizado [1] [33] [16], apenas o JADE [16] provê uma arquitetura genérica para o desenvolvimento de agentes reativos.

7. Agradecimentos

Este trabalho é apoiado pelo CNPq.

Os autores agradecem aos responsáveis pelo processo de revisão, em particular ao Jerffeson Souza, pelas contribuições realizadas no melhoramento da descrição dos padrões propostos neste trabalho.

8. Bibliografia

1 "AgentBuilder User's guide" (2003), Reticular Systems, external documentation, http://www.agentbuilder.com/.

2 BALDASSIN, Alexandro, RIZZO, Ivan, MALTEMPI, Marcus (2002) "Uma Abordagem Baseada em Agentes para Filtragem de Correspondências Eletrônicas". Revista Eletrônica de Iniciação Científica (REIC). Vol. II, n° 4.

3 BEAUMONT, I. (1994) "User Modelling in the Interactive Anatomy Tutoring System ANATOM-TUTOR". User Modelling and User-Adapted Interaction 4(1) 21 –45.

4 BROOKS, R (1986) "A robust layered control system for a mobile robot". IEEE Journal of Robotics and Automation, 2 (1): 14-23.

5 BUSCHMANN, F., MEUNIERS, R., ROHNERT, H., SOMMERLAD, P., STAL, M. A (1996) "System of Patterns. Pattern-Oriented Software Architecture", Wiley.

6 BYLUND, Markus, WAERN, Annika (1997) "Adaptation Agents: Proving Uniform Adaptations in Open Service Architectures", In: Proc. of 3rd ERCIM Workshop on UI for All.

7 COSSENTINO, Massimo, BURRAFATO, Piermarco, LOMBARDO, Saverio, SABATUCCI, Luca (2002) "Introducing Pattern Reuse in the Design of Multi-Agent Systems". AITA'02 workshop at NODe02 - 8-9.

8 DEUGO, Dwight, WEISS, Michael, KENDALL, Elizabeth (2001) "Reusable Patterns for Agent Coordination". Publicado como capítulo 14 do livro: Omicini, A., Zambonelli, F., Klusch, M.,Tolksdorf, R., Coordination of Internet Agents: Models, Technologies, and Applications, Springer.

9 DINIZ, Alessandra (2001) "Uma Arquitetura baseada em Agentes para a Recuperação e Filtragem da Informação". Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão.

10 ERICEIRA, Bruno (2001) “Desenvolvimento de Sistemas Multiagente Utilizando a Ferramenta Zeus”. Monografia do Curso de Bacharel em Ciência da Computação, Departamento de Ciências da Computação, Universidade Federal do Maranhão, São Luís-MA.
11 FARIA, Carla, RIBEIRO, Ismênia, GIRARDI, Rosario (2003) "Especificação de uma Ontologia Genérica para a Construção de Modelos de Usuários", Anais da Terceira Jornada Iberoamericana de Engenharia de Software e Engenharia do Conhecimento (JIISIC 2003, Valdivia, Chile.
12 FERREIRA, Steferson Lima Costa, GIRARDI, Rosario (2002) “Arquiteturas de Software baseadas em Agentes: do Nível Global ao Detalhado”, Revista Eletrônica de Iniciação Científica da SBC.
13 FINK, J, KOBSA, A. (2002) “User Modeling in Personalized City Tours”. Artificial Intelligence Review 18(1), 33-74.
14 GIRARDI, Rosario (2001) “Agent-Based Application Engineering”, In: Proc. of 3rd International Conference on Enterprise Information Systems (ICEIS 2001).
15 GIRARDI, Rosario (2002) “Reuse in Agent-based Application Development”, In: 1st International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS´2002), International Conference on Software Engineering, Orlando, Florida.
16 JADE, Java Agent Development framework (2003). http://jade.cselt.it
17 KENDALL, Elizabeth (1997) “The Layered Agent Pattern Language”, In: Proc. of Pattern Languages of Programming (PLOP'97).
18 KOBSA, Alfred (1999)“Personalised hypermedia presentation techniques for improving online customer relationships”, GMD Report 66.
19 KULKARNI, A (2002) "A Reactive Behavioral System for the Intelligent Room”. Master's thesis, Massachusetts Institute of Technology, Cambridge, MA, http://citeseer.nj.nec.com/kulkarni02reactive.html. Accessed in May 2003.
20 LANGLEY, P (1999) “User Modeling in Adaptive Interfaces”, In: J Kay (ed.) UM99 User Modeling: Proceedings of the Seventh International Conference Springer 357-370.
21 MISLEVY, RJ, GITOMER, DH (1996) “The Role of Probability-based Inference in an Intelligent Tutoring System”, User Modeling and User-Adapted Interaction 5(3-4) 253-282.
22 MOUKAS, Alexandros and MAES, Pattie (1996) “Amalthaea: Information Discovery and Filtering using a Multiagent Evolving Ecosystem”, Proceedings of the Conference on Practical Applications of Agents and Multiagent Technology.
23 NICK, Z, THEMIS, P. (2001) “Web Search Using a Genetic Algorithm”. IEEE Internet Computing, 5(2), 18-26.
24 OLIVEIRA, Ismênia Ribeiro de (2001) “Uma Análise de Padrões de Projeto para o desenvolvimento de Software baseado em agentes”. Monografia do Curso de Bacharel em Ciência da Computação, Departamento de Ciências da Computação, Universidade Federal do Maranhão, São Luís-MA.
25 POHL, Wolfgang, (1999) “Logic-Based Representation and Reasoning for User Modeling Shell Systems”, User Modeling and User-Adapted Interaction 9: 217-282, Kluwer Academic Publishers. Printed in the Netherlands.
26 RUSSELL, S, NORVIG, P (1995) “Artificial Intelligence: A Modern Approach”. Prentice-Hall.
27 SHARMA, Amit (2001) “A Generic Architecture for User Modeling Systems and Adaptive Web Services”, In: Workshop on E-Business & the Intelligent Web. (IJCAI, 2001).
28 SHEHORY, Onn and SYCARA, Katia (2000), “The Retsina Communicator”. In Proceedings of Autonomous Agents, Poster Session.
29 SILVA JUNIOR, Geovanne Bezerra da (2003) “Padrões Arquiteturais para o Desenvolvimento de Aplicações Multiagente”. Dissertação (Mestrado em Ciência da Computação) – Curso de Pós-Graduação em Engenharia de Eletricidade, Universidade Federal do Maranhão.
30 SOBRINHO, Antonio Carlos (2003) “Uma Análise dos Algoritmos Genéticos e suas Aplicações em Sistemas de Acesso à Informação”. Monografia do Curso de Bacharel em Ciência da Computação, Departamento de Ciências da Computação, Universidade Federal do Maranhão, São Luís-MA.

31 STRACHAN, L., ANDERSON, J., SNEESBY, M, EVANS, M (1997) “Pragmatic User Modelling in a Commercial Software System”, In: User Modeling: Proceedings of the Sixth International Conference, UM97, páginas 189-200. Springer, Vienna, New York.
32 WOOLDRIDGE, Michael, JENNINGS, Nicholas (1994) “Intelligent Agents: Theory and Practice”, Knowledge Engineering Review.
33 ZEUS DOCUMENTATION, disponível em: http://193.113.209.147/projects/agents.htm, acesso em julho, 2002.
34 ZHANG, Xiangmin (2003) “Discriminant Analysis as a Machine Learning Method for Revision of User Stereotypes of Information Retrieval Systems”, In: Proceedings of UM'03, 9th International Conference on User Modeling Pennsylvania, USA.

Um Padrão para Gerenciamento de Redes

Calebe de Paula Bianchini [1], Eduardo Santana de Almeida [2], Diogo Sobral Fontes [3], Rossana Maria de Castro Andrade [4]

[1] Departamento de Computação – Universidade Anhembi Morumbi (UAM)
R. Quatá, 203 – São Paulo – SP – Brasil – CEP 04.546-041
Fone/Fax: +55 11 3847.3159

[2] Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Av. Prof. Luis Freire, s/n – Recife – PE – Brasil – CEP 50.740-540 – C.P. 5871

[3] Departamento de Computação – Universidade Paulista (UNIP) – Compus de Assis
R. Myrtes S. Conceição, 301 – C. Nelson Marcondes – Assis – SP – Brasil – CEP 19815-050

[4] Departamento de Computação – Universidade Federal do Ceará (UFC)
Campos do Pici, Bloco 910 – Fortaleza – CE – Brasil – CEP 60455-760

[1] calebe@netsite.com.br, [2] esa2@cin.ufpe.br,
[3] sobral.terra@terra.com.br, [4] rossana@ufc.br

Resumo

A crescente descentralização dos recursos computacionais e a necessidade de gerenciamento de sistemas distribuídos e heterogêneos têm motivado os pesquisadores na construção de ferramentas que auxiliam os administradores de redes, em grande parte das suas tarefas, através do monitoramento automático dos dispositivos. Dentre as técnicas para atingir estes objetivos, destacam-se as que utilizam agentes de software. As tarefas, sinalizadas por eventos e muitas vezes repetitivas, dos diferentes domínios de aplicações, incluindo o das aplicações distribuídas, são delegadas aos agentes de software programados em uma base de conhecimento. Assim, esse trabalho apresenta um padrão de arquitetura para o desenvolvimento de agentes de software inteligentes, em um sistema multi-agentes, que auxiliam o gerenciamento de redes. Esse padrão é denominado GeR (Gerenciamento de Redes).

Abstract

The increasing decentralization of computational resources and the need of distributed and heterogeneous systems management are the motivation for researches to construct tools that assist network managers, in great part of their tasks through automatic monitoring devices. Among the techniques to achieve these goals, the ones that use software agents stand out. The tasks, which are signalized by events and many times repetitive, of different application domains, including distributed application, are assigned to the software agents programmed in a knowledge base. Thus, this work presents an architecture pattern called GeR for developing intelligent software agents that assist in the network monitoring.

1 Contexto

A evolução e a expansão das redes de computadores abriram caminho para o desenvolvimento de novas aplicações nesta área da computação.

Além disso, os avanços na tecnologia aplicada a redes de computadores têm enfatizado a investigação de agentes de *software* inteligentes como um paradigma promissor no desenvolvimento de *softwares* complexos e distribuídos. De um modo geral, agentes de *software* são vistos como um conjunto de objetos complexos, com capacidade de tomar decisões diante uma situação. De fato, objetos e agentes apresentam pontos de similaridade, porém o desenvolvimento de *software* orientado a agentes propõe outros desafios para a Engenharia de *Software* a partir do momento em que agentes necessitam de um nível maior e mais complexo de abstração [1]. Já a inteligência de um agente pode variar de acordo com a capacidade de tomar decisões, as formas de representação dessa inteligência (linguagens simbólicas, etc..), a interatividade com outros agentes, as decisões e ações visando um objetivo, entre outras características que definem o grau de inteligência de um agente.

Esses avanços, juntamente com a expansão das redes de computadores, têm provocado grande necessidade de melhorias no gerenciamento de redes para facilitar e agilizar as tarefas dos seus administradores. Para garantir essas novas exigências, é necessário melhorar a qualidade dos serviços, principalmente os relacionados com gerenciamento de redes [2, 3]. A utilização de agentes de *software* inteligentes facilita a construção de modelos dinâmicos, personalizados para melhorar a qualidade dos serviços no monitoramento de redes, prevendo eventos ou aumentando o tempo de resposta aos eventos ocorridos.

2 Motivação

A crescente descentralização dos recursos computacionais e a necessidade de gerenciamento de sistemas distribuídos e heterogêneos têm motivado os pesquisadores na construção de sistemas que auxiliam os administradores de redes, em grande parte das suas tarefas, através do monitoramento automático dos dispositivos.

Essa melhoria tem sido alcançada com o uso de agentes de *software* inteligentes auxiliando a análise dos dados coletados de cada dispositivo. Além dos recursos que o próprio protocolo de gerenciamento oferece (*traps* [3], *Event Mibs* [4], *Expression Mibs* [5], etc), vários trabalhos vêm propondo técnicas e estratégias para o gerenciamento de redes utilizando agentes [6,7,8,9,10].

3 Problema

Apesar das recentes e constantes pesquisas na área de gerenciamento de redes, ainda há carência de técnicas e ferramentas que suportem tanto o desenvolvimento quanto a utilização de agentes de *software* inteligentes em sistemas de gerenciamento.

Em um lado, diversas técnicas têm sido utilizadas pelos pesquisadores na busca de soluções para os problemas do gerenciamento de redes [11, 3], dada a variedade de plataformas de *hardware* e *software* utilizada, envolvendo diferentes dispositivos conectados, e requerendo cada vez mais dedicação e atenção dos administradores para um bom gerenciamento [2,3]. Assim, existe uma grande preocupação em atingir os serviços de gerenciamento [11,3], procurando automatizar grande parte das tarefas dos administradores.

Já no outro lado, pesquisas envolvendo o estado da arte de agentes de *software* seguem duas diferentes linhas: engenharia de *software* orientado a agentes [12], e engenharia de *software* orientado a objetos para sistemas multi-agentes [13]. Na primeira linha, pesquisadores afirmam veementemente que sistemas multi-agentes são muito mais complexos que os orientados a objetos, sendo que o desenvolvimento tradicional não consegue capturar a complexidade de sistemas multi-agentes. Por outro lado, na segunda linha, pesquisadores

propõem a integração de agentes na orientação a objetos, além de defenderem que agentes e objetos são abstrações complementares.

4 Forças

- É necessário minimizar esforços no desenvolvimento dos sistemas de gerenciamento e dos agentes de *software* inteligentes, concentrando-se nas soluções dos problemas de gerenciamento de redes usando agentes.
- Permitir a integração de diversas formas de representação de inteligência e de definição de comportamento dos agentes em um único sistema de gerenciamento.

5 Solução

O objetivo do GeR é propor um conjunto de componentes para a construção de sistemas de gerenciamento de rede. Desse modo, o Padrão para Gerenciamento de Redes – Padrão GeR – permite:

- integrar diferentes tecnologias, como UML [14,15,16], SNMP [2,3], CORBA [17,18] e Java [19], para a construção de sistemas de gerenciamento de rede [2,3];
- cobrir o ciclo de vida do desenvolvimento de agentes de *software* inteligentes, para serem aplicados em sistemas inteligentes de gerenciamentos, desde a fase de especificação até a fase de implementação, culminando na sua utilização e definição de seu comportamento; e
- utilizar diversas estratégias para conduzir e definir o comportamento do agente, como, por exemplo, linguagem lógica, redes neurais, lógica difusa, filtros de verificação, entre outras técnicas de construção e definição de sua inteligência.

Com esse propósito, GeR oferece dois componentes básicos:

- **device**, que implementa as especificações do protocolo SNMP; e
- **agent**, que analisa as configurações coletadas dos dispositivos.

O componente **device** se comunica com o dispositivo remoto, coletando as configurações descritas pela Mib. O **agent** se conecta ao componente **device** através de suas interfaces, recuperando os valores das configurações. A análise é baseada nesses valores, e seu comportamento é descrito na base de conhecimento através de cláusulas lógicas.

GeR sugere que os gerenciamentos de redes e agentes de *software* sejam definidos e construídos separadamente, mantendo sua independência, mas com interfaces bem definidas para sua relação. Essa modularidade permite a reutilização dos componentes em aplicações de outros domínios, como por exemplo, *e-commerce*, agentes de busca, educação a distância, bioinformática, entre outros, e a utilização de diversas técnicas de construção e definição de sua inteligência. Além disso, também permite que apenas partes do padrão sejam reutilizadas, tanto no desenvolvimento de sistemas inteligentes de gerenciamento de redes, como também no desenvolvimento e definição de agentes de *software* inteligentes.

Na solução do GeR, utilizamos o método Catalysis para o desenvolvimento dos componentes, cobrindo suas fases, desde o levantamento de requisitos até a implementação. Para a utilização dos agentes, utilizou-se a plataforma DSAP, que fornece toda uma infraestrutura para o desenvolvimento, criação, execução e mobilidade desses agentes. Dessa forma, grande parte do esforço que seria gasto na construção do agente é direcionada para a definição de seu comportamento e inteligência.

6 Estrutura

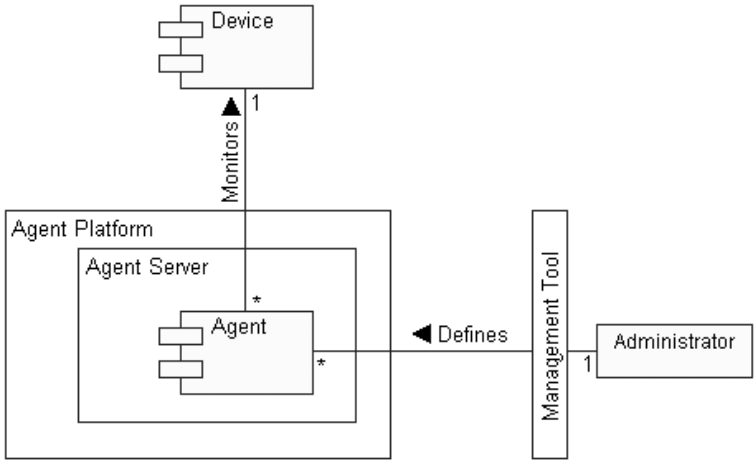


Figura 1. Estrutura do Padrão

7 Participantes

Administrator
- interage com a ferramenta definindo o gerenciamento dos dispositivos, criando agentes de *software* e descrevendo seu comportamento na base de conhecimento.

Management Tool
- oferece recursos para a interação entre o administrador e o sistema de gerenciamento e com os agentes de *software* [6].

Device
- são os dispositivos gerenciáveis, descritos através de suas MIBs, definidos pelo administrador e gerenciados pelos agentes.

Agent
- implementa um conjunto de classes e os seus relacionamentos [20], para a instanciação de agentes inteligentes.

Agent Server
- fornece infra-estrutura e serviços necessários às aplicações distribuídas e multiplataformas em sistemas multi-agentes [21,22]

Agent Platform
- define um ambiente para desenvolvimento de aplicações de diversos domínios e um conjunto escalável de servidores [21,22].

8 Conseqüências

O GeR oferece os seguintes benefícios:
- **Modularidade**: o Padrão permite separar os aspectos de desenvolvimento de agentes e de gerenciamento de redes; e

- **Reutilização**: através dos aspectos de modularidade oferecidos, os desenvolvedores podem reutilizar os componentes, diminuindo a redundância de código.

Mesmo com as vantagens listadas acima, as seguintes desvantagens podem ser apresentadas:

- Conhecimento de outros paradigmas [23,24]: o comportamento dos agentes utilizado no monitoramento é descrito em uma linguagem lógica [25,26,27], obrigando o administrador a conhecer esse paradigma.

9 Implementação

Para a implementação do GeR, utilizou-se as seguintes abordagens:

9.1 Catalysis

Catalysis [14] é um método de desenvolvimento de *software* Orientado a Objetos que utiliza Componentes Distribuídos, Padrões de Projetos e *Frameworks* para projetar e construir Sistemas de Negócio, que devem suportar tecnologias orientadas a objetos, utilizando Java, CORBA ou DCOM (*Distributed Component Object Model*) [28]. Sua notação é baseada na UML [15] e fundamenta-se em três princípios: abstração, precisão e componentes "*plug-in*". O princípio **abstração** orienta o desenvolvedor na busca dos aspectos essenciais do sistema, dispensando detalhes que não são relevantes para o contexto do sistema. O princípio **precisão** tem como objetivo descobrir erros e inconsistências na modelagem e o princípio **componentes "*plug-in*"** usa o reuso de componentes para construir outros componentes [14,16].

Catalysis apresenta modelos precisos e um processo de desenvolvimento completo e sistemático, permitindo aos desenvolvedores partirem da análise e especificação do domínio da aplicação e chegarem ao código, dividindo o sistema em componentes e identificando ao longo do processo os elementos de reutilização [14].

O processo de desenvolvimento de *software* em Catalysis é divido em três níveis: Domínio do Problema, Especificação dos Componentes e Projeto Interno dos Componentes.

O componente *agent* foi desenvolvido utilizando os níveis Catalysis, como se segue:

Domínio do Problema.

Neste nível é dado ênfase no entendimento do problema, isto é, especifica-se "o que" o sistema deve atender para solucionar o problema. Nesta fase, utilizam-se técnicas de entrevista coletiva e informal com o usuário, que são documentados com *Storyboards* [16] e *MindMaps* [16]. Essas representações identificam a ligação entre os termos obtidos na entrevista, definindo assim, uma terminologia do domínio do problema.

Uma vez definida a terminologia do domínio do problema, pode-se criar Modelos de Colaboração [16] e de Casos de Uso [16], que representam os atores e suas interações com o sistema.

A Figura 2 apresenta o Domínio do Problema. Os requisitos identificados foram especificados em Modelos de Colaboração, representando a coleção de ações e os objetos participantes. Em seguida, estes Modelos foram refinados em Modelos de Casos de Uso.

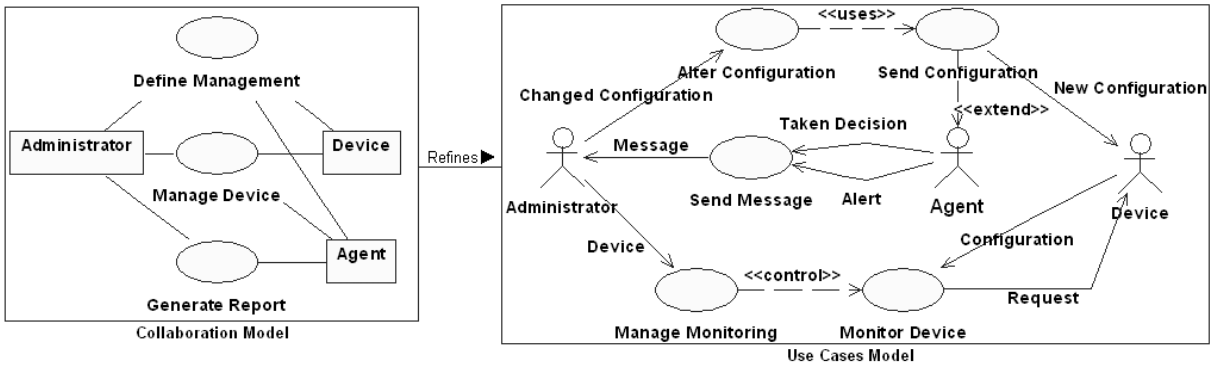


Figura 2. Domínio do Problema

Especificação de Componentes.

Este nível descreve o comportamento externo do sistema de uma forma não ambígua, tendo início com o mapeamento dos diagramas obtidos no Domínio do Problema para o Modelo de Tipos [16] que especifica o comportamento dos objetos. Esse modelo mostra os atributos e as operações dos tipos de objetos, sem se preocupar com a implementação. A partir do Modelo de Tipos constroem-se Diagramas de Seqüência [15] que têm por objetivo mostrar os cenários de execução das operações ao longo do tempo. Outras técnicas, como Diagramas de Estados [15], podem ser utilizadas para a especificação dos componentes. A Figura 3 apresenta o Diagrama de Estados para o componente *agent*.

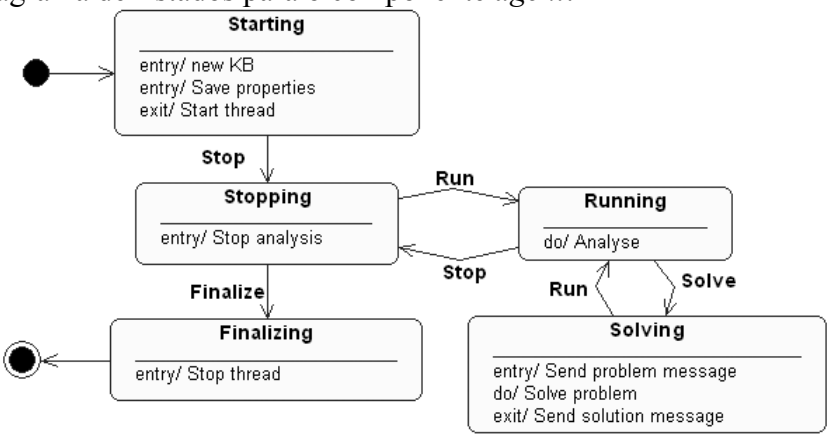


Figura 3. Diagrama de Estados

Projeto Interno de Componentes.

Neste nível define-se "como" serão implementados os requisitos especificados para os componentes, preocupando-se com sua distribuição. O Projeto Interno dos Componentes começa com a definição do Modelo de Classes [15] do Sistema, mostrando as classes com seus atributos, operações e relacionamentos. Esse modelo é derivado do Modelo de Tipos, obtido no nível anterior. Utiliza-se o Modelo de Interação [15] para mostrar a interação entre os objetos. Este modelo é derivado do Diagrama de Seqüência obtido na Especificação dos Componentes. Técnicas para representar a Arquitetura das plataformas Física e Lógica também são usadas neste nível [14]. A Figura 4 mostra as principais classes do componente *agent*.

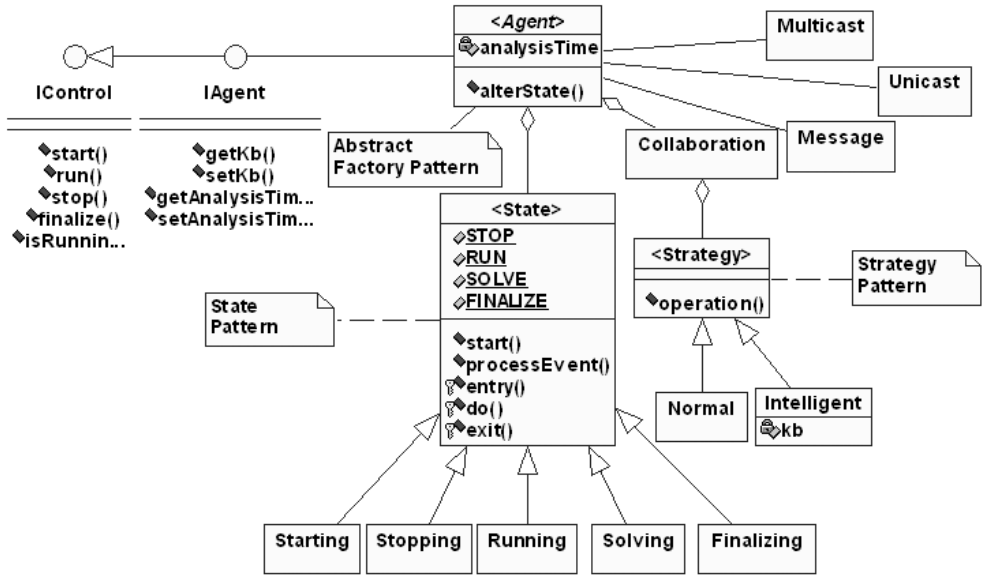


Figura 4. Modelo de Classes

9.2 DSAP – *Distributed Software Agent Platform*

A plataforma DSAP (*Distributed Software Agent Platform*) [21,22] é um ambiente para desenvolvimento de aplicações que utilizam a tecnologia de agentes de *software* em ambiente distribuído, implementada em Java.

Uma das evoluções da plataforma é a utilização de CORBA [17,18], que é um padrão já estabelecido pela OMG (*Object Management Group*) para suportar a distribuição de objetos. CORBA apresenta interfaces bem definidas e independentes de aplicações, através da IDL (*Interface Definition Language*) [17,18] que se encaixa perfeitamente no contexto de Desenvolvimento Baseado em Componentes. Outros aspectos que motivaram o uso de CORBA foram: independência de linguagem de programação, devido a possibilidade do mapeamento da IDL para diversas linguagens; a portabilidade entre ambientes computacionais; e os serviços de Segurança, Nomeação e Notificação, oferecidos pela especificação.

Nessa plataforma multi-agente, é utilizada a engenharia de *software* orientada a objetos, já que as tecnologias de agentes de *software* e orientação a objetos possuem abstrações complementares [13,29]. Assim, pode-se estender técnicas existentes na orientação a objetos, como padrões [30], *frameworks* e componentes [14], e linguagens de modelagem [15,16] para o desenvolvimento de sistemas multi-agentes.

Esta plataforma consiste de um conjunto escalável de servidores utilizados por aplicações de diversos domínios, como por exemplo: *e-commerce*, agentes de busca e educação a distância, criadas a partir da reutilização do *framework* disponível na plataforma. A Figura 5 apresenta a visão geral da plataforma.

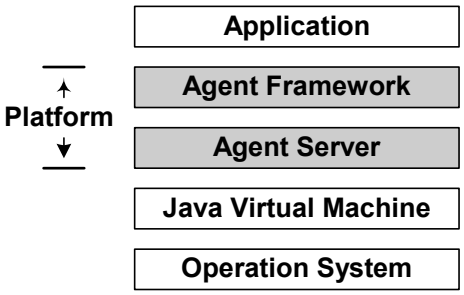


Figura 5. Plataforma DSAP – *Distributed Software Agent Platform*

A camada *Agent Framework* é descrita por um conjunto de classes abstratas e concretas relacionadas, que podem ser estendidas ou instanciadas, necessárias para criação dos agentes, e que irão realizar suas atividades em um ambiente de sistemas distribuídos. A Figura 6 mostra as principais classes do *framework*.

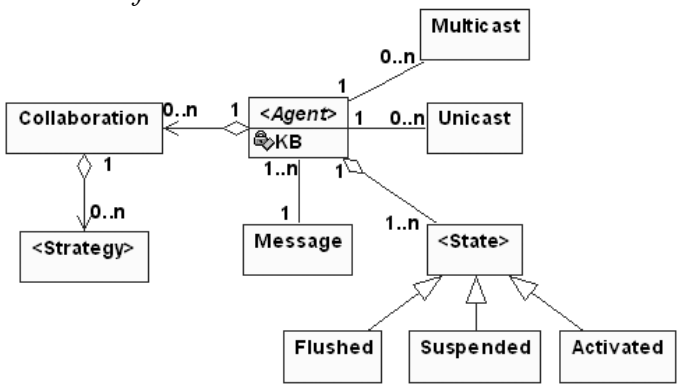


Figura 6. *Agent Framework*

A camada *Agent Server* fornece a infra-estrutura e os serviços necessários às aplicações distribuídas e multiplataformas em sistemas multi-agentes. É responsável por tratar as requisições das aplicações e de outros *Agent Servers*, fornecendo o ambiente de execução para os agentes móveis ou estacionários, conforme mostra a Figura 7.

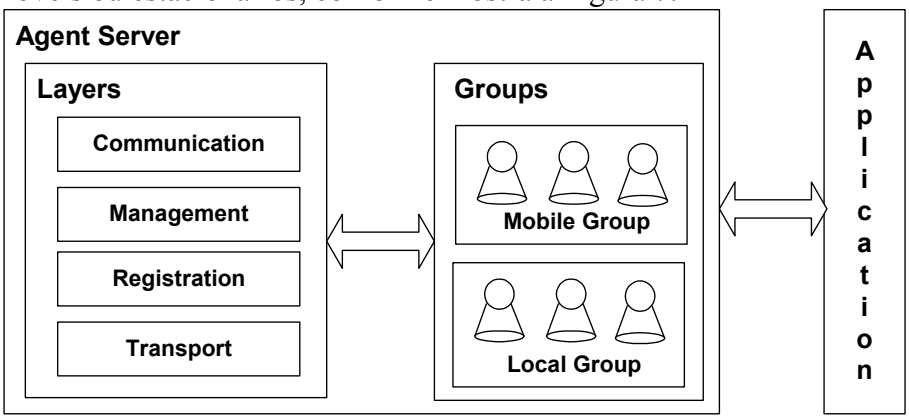


Figura 7. *Agent Server*

O *Agent Server* contém informações dos servidores disponíveis no momento e permite que as aplicações ou os agentes móveis encontrem os servidores distribuídos na rede. O *Agent Server* contém as seguintes camadas:

- *Communication*: responsável pela comunicação remota entre servidores, fornecendo serviços básicos para as aplicações, tais como: localização de servidores, localização de agentes e transporte de agentes;
- *Management*: responsável pela criação e controle dos agentes, fornecendo às aplicações as seguintes funcionalidades: criação, remoção, suspensão e ativação dos agentes. A suspensão e ativação são utilizadas pelos agentes móveis para migrar de um servidor para outro;
- *Register*: responsável por registrar o agente no servidor, tornando-o visível para as aplicações; e
- *Transport*: responsável pelo armazenamento do estado de execução do agente, preparando-o para ser enviado a um outro servidor. Realiza também a recuperação do agente.

10 Usos Conhecidos

A ferramenta MoDPAI [6,31] é um sistema de gerenciamento de redes que utiliza o protocolo SNMP [2,3]. Ela oferece recursos gráficos para auxiliar o administrador na tarefa de monitoramento, permitindo a criação de agentes inteligentes, programados um uma base de conhecimento, que analisam os dados coletados dos dispositivos.

Os agentes foram desenvolvidos utilizando o padrão GeR, construindo uma comunidade de ferramentas que aperfeiçoam o gerenciamento de redes com a troca de agentes entre si.

O comportamento dos agentes é descritos em uma linguagem lógica, e, ao transportar-se de uma ferramenta para outra, o agente leva consigo o código já instanciado em objetos.

A Figura 8 mostra a tela de gerenciamento da MoDPAI, com alguns dispositivos, servidores e *workstations*, sendo monitorados.

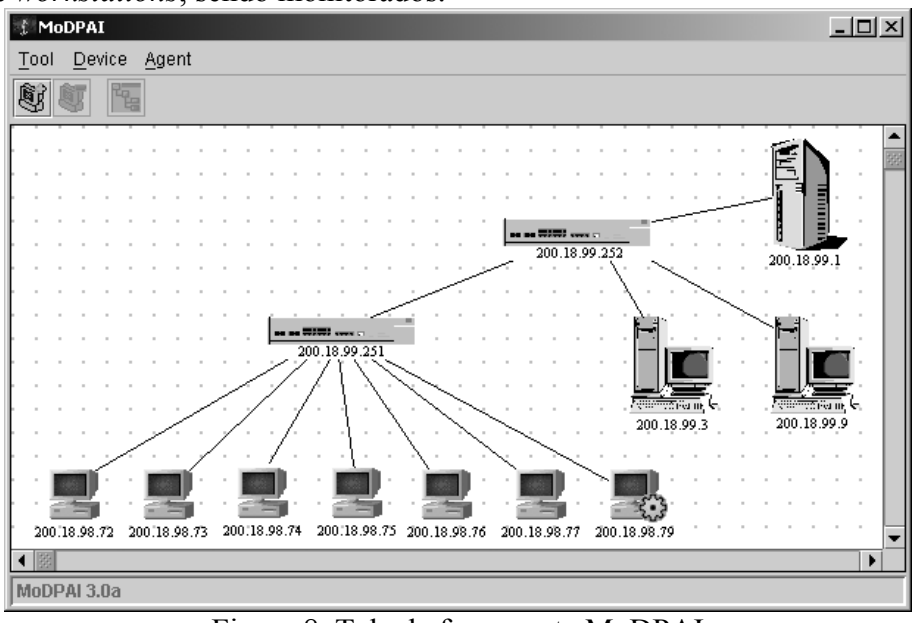


Figura 8. Tela da ferramenta MoDPAI

Para detalhar seu funcionamento em uma rede de computadores, é mostrado um estudo de caso feito no Departamento de Computação da Universidade Federal de São Carlos, onde foi instalada uma comunidade de quatro ferramentas. Foram adicionados todos os servidores (*e-mail*, *webmail*, arquivo, *www*, roteador, DNS) e aproximadamente 200 estações de trabalhos desse departamento. Para cada um desses dispositivos, foi definido um agente de *software* capaz de analisar as configurações coletadas. Em especial, foram analisadas as seguintes perspectivas:

- o número de pacotes TCP [32] e UDP [32] recebidos e enviados da rede interna pelo roteador, viabilizando uma segunda rota de transmissão de dados através de uma nova interface de rede, quando esse número estiver na iminência de sobrecarregar a primeira linha de dados;
- a quantidade de conexões *Web* estabelecidas, relatando seus tempos e verificando o período de maior número de conexões para caracterizá-lo como período crítico;
- o menor e maior período entre duas conexões de *e-mail*, estimando se o serviço de *e-mail*, considerado crítico, não falhou;
- os recursos de memória e disco do servidor de domínio, verificando se este não parou de funcionar; e
- o início e término de uso de cada estação, verificando o volume de informações trocadas pela rede e os recursos de memória e disco disponíveis.

Para o roteador, foram analisadas as entradas da tabela *tcpConnTable*, conforme a Mib mostrada na Figura 9, permitindo alterar as conexões ativas.

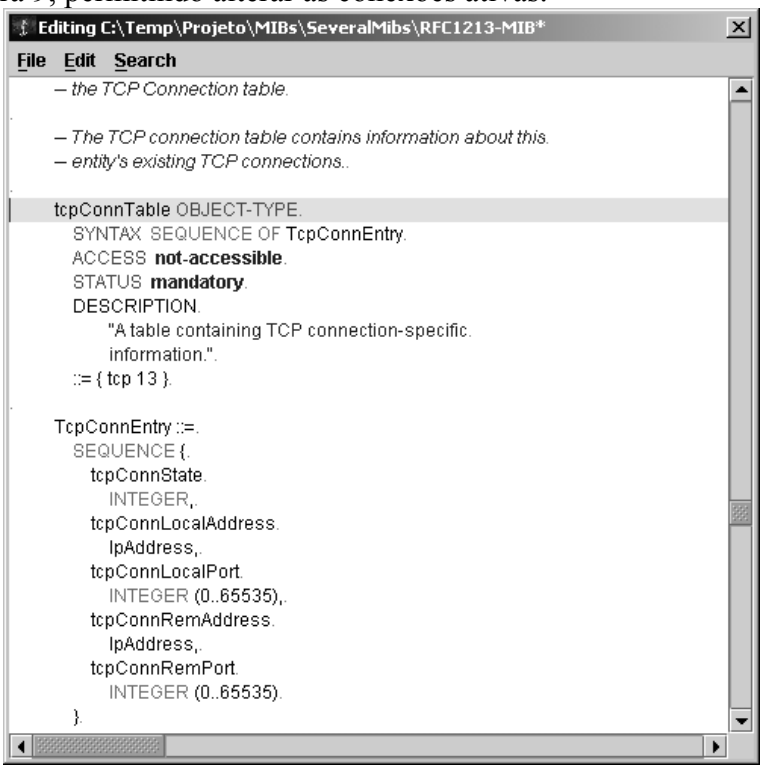


Figura 9. Tabela *tcpConnTable* no arquivo de descrição Mib

A Figura 10 mostra as telas da especificação do um agente de *software* responsável pela verificação dessa tabela.

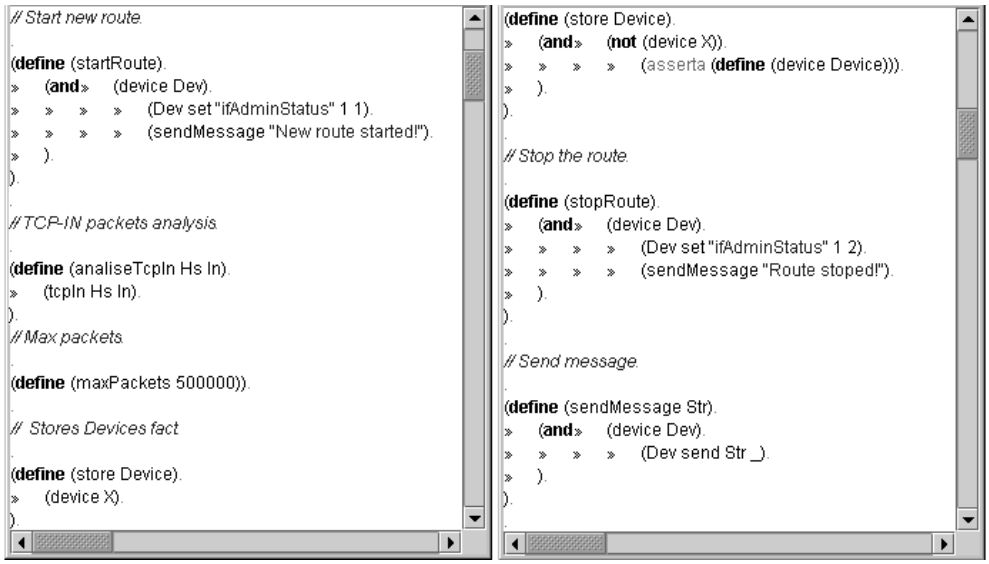


Figura 10. Código KB

Essa figura mostra parte do código KB que define o comportamento do agente, responsável pela análise do número de pacotes que trafegam pelo roteador, habilitando uma nova rota (1), caso haja necessidade. Nesse código, os fatos que indicam o período e a quantidade de pacotes transmitidos são criados ou alterados dinamicamente, mantendo atualizada a base de conhecimento. Com o auxílio de regras, os dados de tráfego são analisados, fazendo busca nos fatos já armazenados (2), e verificando se existe a necessidade de habilitar uma nova rota, conforme o limite máximo de pacotes (3) estipulado para a primeira rota. Além disso, é possível prever o período de maior uso da rede, antecipando a ação de habilitar essa nova rota, assim como o período decrescente de uso da rede, permitindo o desligamento dessa segunda rota (4). Para as decisões que interfiram na rota, uma mensagem é gerada e enviada para uma conta de *e-mail* (5), conforme definido nas propriedades do agente.

Além da MoDPAI, que efetivamente usou o GeR, outros trabalhos [7,8,9,10] envolvendo o monitoramento de redes e agentes de *software* podem implementar esse padrão. Esses trabalhos, além de utilizarem o protocolo SNMP para o gerenciamento, abordam diferentes métodos para a construção do agente. Com o uso do padrão GeR, os esforços na construção desses sistemas e dos agentes são minimizados, havendo uma concentração em solucionar os problemas de redes utilizando formas diferenciadas para a definição do comportamento e representação da inteligência do agente.

11 Padrões Relacionados

- *Abstract Factory* [30,16]: o GeR utiliza esse padrão para fornecer uma interface de criação de família de objetos relacionados ou dependentes sem a especificar sua classe concreta;
- *State* [30,16]: o GeR utiliza esse padrão para permitir que um objeto alterne seu comportamento de acordo com seu estado interno;
- *Strategy* [30,16]: o GeR utiliza esse padrão para encapsular algoritmos relacionados em subclasses de uma superclasse em comum. Isso permite que a seleção de um algoritmo varie não só através de objetos, mas também durante o tempo.

12 Agradecimentos

Os autores gostariam de agradecer à *Shepherd* Dra. Rossana Maria de Castro Andrade pelas sugestões recebidas durante o processo de *shepherding*.

13 Referências

1. Lucena, C.; ... [et al]. *Software Engineering for Large-Scale Multi-Agent Systems – SELMAS'2002*. Proceedings of The 24th International Conference on Software Engineering. p653-654. Orlando, Florida, USA. 2002.
2. Stallings, W. *Network Management*. IEEE Computer Society Press. 1993. 354p.
3. Stallings, W. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. 3 ed. Addison-Wesley, 1999. 619p.
4. Kavasseri, R.; Stewart, B. *Distributed Management Expression MIB*. Request for Comments 2982, October, 2000.
5. Kavasseri, R.; Stewart, B. *Event MIB*. Request for Comments 2981, October 2000.
6. Bianchini, C. P.; ... [et al]. *Devices Monitoring Tool using Pervasive Computing and Software Agents*. Proceedings of The 2002 International Conference on Security and Management. Las Vegas, Nevada, USA. 2002.
7. Spolidoro, F.; Rodriguez, N. *Distributed Environment for Web-based Network Management*. Proceedings of the 26th IEEE Conference on Local Computer Networks, 2001, p. 41-48.
8. Goldszmidt, G.; Yemini, Y. *Delegated Agents for Network Management*. IEEE Communications Magazine, March 1998.
9. Bieszcazad, A.; Pagurek, B. *Mobile Agents for Network Management*. IEEE Communication Surveys, Fourth Quarter 1998, Vol.1 No.1.
10. Fernandes, H. D. H.; Duarte Jr., E. P.; Musicante, M. A. *ANEMONA: Uma Linguagem de Configuração para Aplicações Práticas de Gerência Distribuída*. Anais do 20o. Simpósio Brasileiro de Redes de Computadores. Búzios, Brasil. 2002.
11. International Standard ISO/TEC 7498-4. *Information processing systems – Open System Interconnection – Basic Reference Model – Part 4: Management framework*. 1989. 13p.
12. Petrie, C. *Agent-Based Software Engineering*. Lecture Notes in AI, Springer-Verlag. 2000.
13. Lange, D. B.; Oshima, M. *Programming and Deploying Java™ Mobile Agents with Aglets™*. Addison-Wesley, 1998. 225p.
14. D'Souza, D. F.; Wills, A. C. *Objects, Components, and Framework with UML: The Catalysis Approach*. Addison-Wesley, 1998. 785p.
15. Booch, G.; Rumbaugh, J.; Jacobson, I. *UML, guia de usuário*. Editora Campus, 2000. 472p.
16. Larman, C. *Utilizando UML e Padrões*. Bookman, 2001. 494p.
17. Orfali, R., Harkey, D. *Client/Server Programming with Java and CORBA*. John Wiley & Sons, Second Edition, 1998.
18. CORBA. *The Common Object Request Broker: Architecture and Specification*. URL: http://www.omg.org. 04/2001.
19. J2SE 1.3. *Java 2 Platform SE v 1.3*. URL: http://java.sun.com/j2se/1.3/docs/api/index.html 01/2001.
20. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998. 432p.
21. Bianchini, C. P.; Fontes, D. S.; Prado, A. F. *Distributed Software Agents Platform Framework*. First International Workshop on Software Engineering for Large-Scale Multi-Agent Systems in conjunction with 24th International Conference on Software Engineering. Orlando, Florida, USA. 2002.
22. Bianchini, C. P.; ... [et al]. *Distributed Software Agents Platform*. Proceedings of ACIS International Conference on Computer Science, Software Engineering, Information Technology, e-Business, and Applications. p 174-179. Foz de Iguaçu, Paraná, Brazil. 2002.
23. Clocksin, W.F.; Mellish, C.S. *Programming in prolog*. 2ed. Springer-Verlag, 1984. 297p.
24. Sterling, L., Shapiro, E. *The Art of Prolog*. 2ed. The MIT Press, 1994. 509p.

25. Lumina Corporate Solution. *Lumina Corporate Solution – Development*. URL: http://www.luminacorp.com. Acessado em Janeiro de 2001.
26. Lumina Corporate Solution; Moura, L. M. *Implementação da KB*. Relatório Técnico Interno. 2000. 88p.
27. Sant'Anna, M. H. B. *Circuitos Transformacionais*. Rio de Janeiro: PUC-RJ, 1999. (Tese: Doutorado em Informática)
28. Microsoft Corporation. *Distributed Component Object Model (DCOM) - Downloads, Specifications, Samples, Papers, and Resources for Microsoft DCOM*. URL: http://www.microsoft.com/com/tech/dcom.asp. May, 2002.
29. OMG Document. *Agent Technology*. Green Paper. 2000. URL: http://www.objs.com/agent/agents_Green_Paper_v100.doc. Acessado em Dezembro de 2001. 67p.
30. Gamma, E.; ... [et al]. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994. 395p.
31. Bianchini, C. P. *Ferramenta para Monitoramento utilizando Computação Pervasiva e Agentes de Software Inteligentes*. São Carlos: UFSCar, 2002. (Dissertação: Mestrado em Ciência da Computação).
32. Stevens, W. R. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, 1999. 575p.

Special Session on Software Pattern Applications

SPA is dedicated to explore applications that involve software patterns. It provides a forum for researchers and practitioners in the area to meet and exchange research ideas and results.

We want to spread the use of patterns in Latin America, stimulating not only new patterns to write, but also disseminating the culture of patterns among software developers. This can be obtained if we show that patterns are a powerful reuse technique that has evolved in the last decade and is being used more and more in concrete projects.

The SPA Session Dynamics

SPA sessions were about 30 minutes each, with 25 minutes for authors' presentation and 5 minutes for questions.

MVCASE Tool – Working with Design Patterns

Daniel Lucrédio [1]
Alexandre Alvaro [1]
Eduardo Santana de Almeida [2]
Antonio Francisco do Prado [1]

[1] Computing Departament – Federal University of São Carlos
Rod. Washington Luiz, km 235 – São Carlos/SP - Brazil
P.O box 676 – Zip.Code 13565-905 - Phone/Fax: + 55-16-260-8232
{aalvaro, lucredio, prado, trevelin}@dc.ufscar.br

[2] Informatics Center - Federal University of Pernambuco - Recife Center for Advanced Studies and Systems
Av. Professor Luiz Freire - Recife/PE – Brasil - University City - Zip. Code: 50740-540
Phone: + 55-81-3271-8430
esa2@cin.ufpe.br

Abstract

Working with design patterns can be considerably improved when using tools that help in pattern creation and application. However, there are several issues involved. For instance, a tool must offer mechanisms to help the Software Engineer to find the right patterns to a particular solution, or to identify and recognize patterns after they were applied. This paper presents the main requirements for these tools, and presents an extension of MVCASE tool to help Software Engineers to create and reuse design patterns in a high abstraction level during the software development process. The tool uses a distributed repository to store the patterns, and uses a code generator to automate most of the implementation process.

1. Introduction

Design patterns are among the most important topics inside the Software Engineering research area. According to Florijn et al [10], design patterns describe general solutions for recurring design problems. A pattern[1] is described in terms that explains its usage including, for example, when and how to use it. This description usually follows the pattern format, as can be seen in [11].

It is commonly accepted that the use of design patterns in the development of object-oriented software offers several benefits. These include, among others: flexibility and understandability [4], experience encapsulation and reuse, better documentation, and the creation of a common pattern repository. However, as new patterns are created, it may

[1] Although there are different kinds of patterns, this paper deals only with design patterns. Therefore, the term pattern will be used to reference this kind of pattern.

become too difficult, or even impossible, to find one that fits a particular solution, due to the great number of patterns available [1].

In order to avoid this problem and to achieve the benefits, the patterns must be correctly used. For example, the right pattern must be found in order to solve a given problem. Another issue is that the code corresponding to a design pattern must be constructed every time it is applied [6]. It is also possible that two or more patterns are applied together to solve a single problem, which may cause some confusion.

These and other issues may be resolved manually, although it demands some effort. To help in design patterns use, several tools and environments have been proposed. These tools aim to facilitate the creation and application of design patterns. Automatic code generation and pattern structures visualization are examples of some features present in such tools.

In most of cases, these tools deal only with pattern-related issues. However, pattern usage is not isolated from the software development process, and cannot be treated as such. For this reason, in order to be effective, a tool that works with patterns must be integrated with the development process and, if possible, with the tools used for the development.

This paper presents an extension of the MVCASE tool [2,3,5,17,18], which attempts to solve this problem by integrating development and pattern-related activities in a single tool, helping to create and apply design patterns during the development process. The paper is organized as follows: Section 2 presents the main requirements for design pattern creation and application. In section 3, MVCASE tool and its support for design patterns are presented. Section 4 revises the tool in relation to the requirements. Related works are discussed in section 5. Section 6 presents some final considerations and future works.

2. Requirements for Design Pattern Creation and Application

Tool support for working with design patterns involves several issues. Many researches and discussions about this subject can be found in the literature [6,8,10,22]. These works identify mainly two major activities: pattern creation and pattern application. Next, the main requirements for tool support on design patterns are identified, discussing the issues related to these two activities.

2.1. Support for high abstraction level specifications

Design patterns, as the name suggests, represent design-level solutions. These solutions are reflected on the executable code, but do not depend on it. Therefore, a tool intended to support design patterns must be independent of any implementation language. More than that, it is natural to conclude that this tool should work in the same abstraction level as the patterns, in other words, the design level.

2.2. Support for pattern creation

When creating a new pattern, the Software Engineer must be able to create, modify and publish the new pattern, according to the pattern format [11], including all the information required, such as the pattern name, the problem to which it is related, the proposed solution and the consequences. To better illustrate the solution, models, such as classes or components structures and sample code may be included. Their relationship with other existing patterns may also be included.

2.3. Patterns storage

After creating the pattern, the Software Engineer must have ways to store it in a place from where it can be retrieved for later reuse. The storage mechanism must organize the patterns, storing it together with all the information involved. It is also responsible for:

- *avoiding redundancy*: the same pattern should not be stored twice;
- *maintaining the consistency*: the information concerning the pattern should not be missing or corrupted; and
- *managing patterns references*: in order to improve the user knowledge of the existing patterns and their relations; whenever two stored patterns relate to each other, it should be possible to navigate from one to another.

2.4. Support for patterns searching

Patterns application occurs during the software development process, when a design pattern can be used to help some problem solution. To identify the best suitable pattern for each problem is not an easy task. The software engineer must be aware of the existing patterns in order to choose one that solves the problem. However, if there are too many patterns, it is unlikely that somebody gets to know all of them. Therefore, there must be a searching mechanism which helps the Software Engineer to find a particular pattern, without needing to know all of them.

2.5. Pattern application

After the Software Engineer decided which pattern to use, his chosen solution must be applied into the software that is being developed. The simplest way to do this is to instantiate elements from the pattern, and then bind them to elements in the software. Florijn et al [10] state that three approaches must be considered when instantiating a design pattern:

- Top-down: "*given a pattern description, generate the necessary design elements in the program and bind them to the pattern. This is typically expressed by: `Give me an instance of the Observer pattern', after which the developer is given an initial set of classes (with methods, relations, etc.) that follows the canonical structure of the Observer pattern*".
- Bottom-up: "*given a number of elements in the program, bind them to a new pattern instance. This addresses situations such as: `These classes (and their methods) together reflect the Proxy pattern; let's turn it into a Proxy pattern.' The difference to the top-down approach is that no new design elements are created in the program, instead existing elements are used*".
- Mixed: "*this approach differs from bottom-up in that the program elements that only partly meet a pattern structure may be combined with newly generated elements that are generated. An example of this is: `This structure closely resembles the Composite pattern; turn it into a Composite instance.' In this case, the structure will be completed with new design elements that were missing*".

However, design patterns application is not just about instantiating some elements and binding them to the pattern. There is a whole solution included, called a *pattern realization plan*, that explains how the pattern is mapped into the language in a particular circumstance

[7]. This plan must be followed in order to correctly apply the pattern. A tool to support pattern application must also have features that help the correct plan execution.

2.6. Pattern tracing

After a pattern is applied, all that is left in the software is a portion of executable code that has no recognizable trace of a design pattern. There must be a way to identify and recognize the patterns that were applied, binding elements of the software to their respective roles in the pattern. In a debate on tool and language support for design patterns, Chambers et al [7] already mentioned the need for this tracing when applying patterns.

2.7. Integration with the development process

As mentioned before, design patterns must not be treated separately from the development process. The same idea is valid for pattern tools, which must interoperate with other development tools. Ideally, a pattern tool should be an integrated part of a software engineering environment, composed by several tools, including modeling and implementation tools, among others. Further discussion about software engineering environments and tools integration can be found in [13, 19, 21].

3. MVCASE tool

In order to better understand how MVCASE supports design patterns, the basic tool features to develop object-oriented software are presented first. Next, its support for design patterns is presented.

The MVCASE (Multiple View CASE) is a modeling tool that provides graphical and textual techniques based on UML [24], to develop object-oriented software. It is fully implemented in the Java language, thus it is platform-independent.

By using UML-based techniques, MVCASE allows the software engineer to specify object-oriented software in different abstraction levels and four different views, as follows:

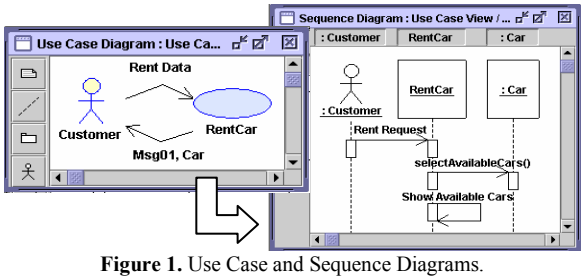


Figure 1. Use Case and Sequence Diagrams.

- *Use Case View*: offers a behavioral view, from the external actors' viewpoint. This view has two main diagrams: the use case diagram, that comprehends a set of use cases, actors and their relationships; and the sequence diagram that details a use case, with a set of objects and a temporal messages sequence [20]. Figure 1 shows use case and sequence diagrams examples, created in MVCASE.

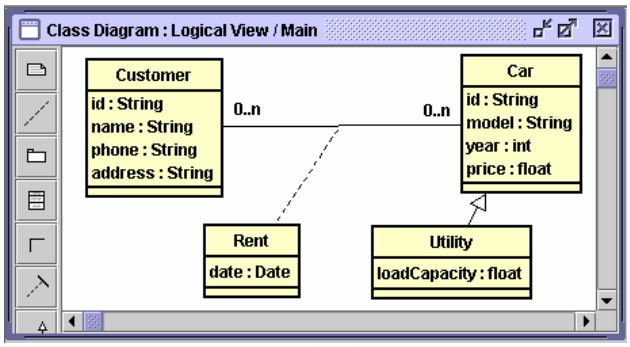


Figure 2. Class diagram.

- *Logical View*: offers a structure of packages, classes and relationships between them. The main diagram in this view is the class diagram, that provides a static view, containing classes, interfaces,

collaborations and relationships. Among these relationships are inheritances, associations, aggregations and dependencies [20]. Figure 2 shows a class diagram example in MVCASE.

- *Component View*: offers a static, architectural view, structured in modules. The component diagram is the main diagram of this view. It shows components and their relationships [20]. Figure 3 shows a component diagram example in MVCASE.

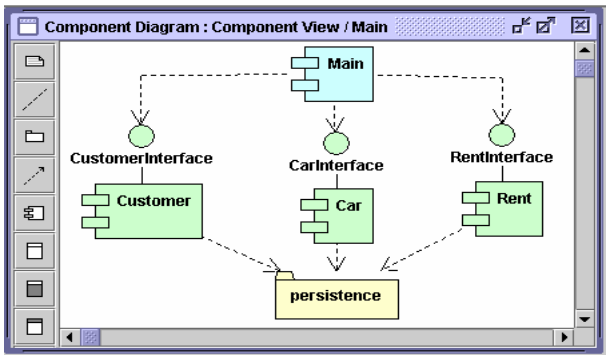


Figure 3. Component Diagram.

- *Deployment View*: offers a static view, showing the deployment architecture. There are nodes and connections showed in a deployment diagram [20]. Figure 4 shows a deployment diagram example constructed in MVCASE.

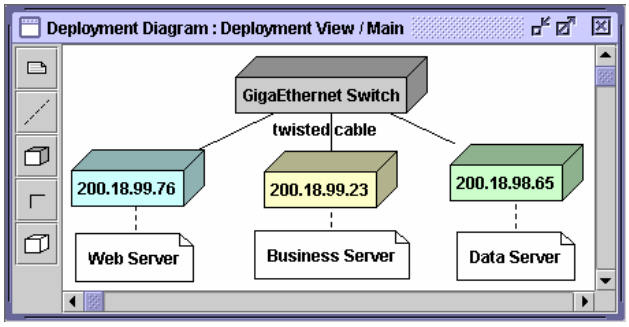


Figure 4. Deployment diagram.

These views are displayed in the tool's browser, which organizes the packages, elements and diagrams in a tree structure. The browser allows the software engineer to view, create, remove and edit the software elements.

To persist the UML specifications, MVCASE uses the OMG's XMI standard [25] and a repository. XMI is a XML-based format to represent UML metadata. Based on the four views described above, MVCASE generates XMI descriptions and stores them in the repository. The original XMI format was extended in order to represent graphical information, such as elements position within the diagrams.

The repository can store XMI documents. Since the scope of XMI is not fixed, different kinds of software artifacts can be stored, including UML specifications and executable code, for example. The repository has mechanisms for version control, which means that it can control different versions of a given artifact. It has also security and transaction management mechanisms, to assure the consistency and reliability of the stored artifacts.

The services for storing and searching software artifacts are available through a middleware [16]. By doing so, these services can be accessed over a network. The searching service combines searching and browsing, in order to obtain better results when finding artifacts that are stored in the Repository. Figure 5 shows the repository architecture.

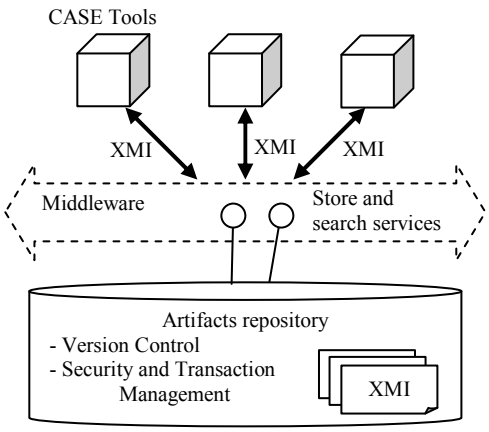


Figure 5. Software artifacts repository architecture.

Using the repository, the information produced during the software development process can be stored and managed, becoming available for later reuse. The repository is currently under development, using multi-agents systems technologies and searching engines, being part of a MSc. Dissertation.

In order to assist the implementation activities, MVCASE allows the Software Engineer to embed code into the design. Operations can have their behavior described directly in the executable language. This code is then stored together with the high-level descriptions, being used later for code generation. In this way, the Software Engineer can fully implement the software in MVCASE.

MVCASE also provides wizards, to automate tasks such as code generation, packaging and components publication. Until this moment, these wizards support the following technologies: Java 2 [14], EJB [9], CORBA [23] and Java Servlets [15]. MVCASE is currently being used in several Brazilian institutions, serving as a basis for other researches and for teaching.

These are the basic MVCASE features to help the object-oriented software development. Next, its support for design patterns is presented.

3.1. MVCASE support for design patterns

As mentioned earlier, there are two major activities when working with design patterns: pattern creation and patterns application. Next, a description of how these two activities are performed in MVCASE is presented. Some of these features are not completely implemented yet, such as the repository searching engine. However, the main features are implemented and functional.

Pattern creation.

In order to create a design pattern, the Software Engineer must describe it in a way that the problem, the solution, and the consequences related to that pattern can be clearly understood. Gamma et al [11] point out that graphical notations aren't sufficient to describe patterns, and propose a format that is widely used by patterns designers.

With basis on this idea, MVCASE have features to create patterns and describe it either in textual descriptions according to the pattern format [11], and in graphical UML notations. To edit the textual descriptions, MVCASE has an editor which allows the Software Engineer to fill out the several items of the description, as shown in Figure 6, for the *Command* pattern [11].

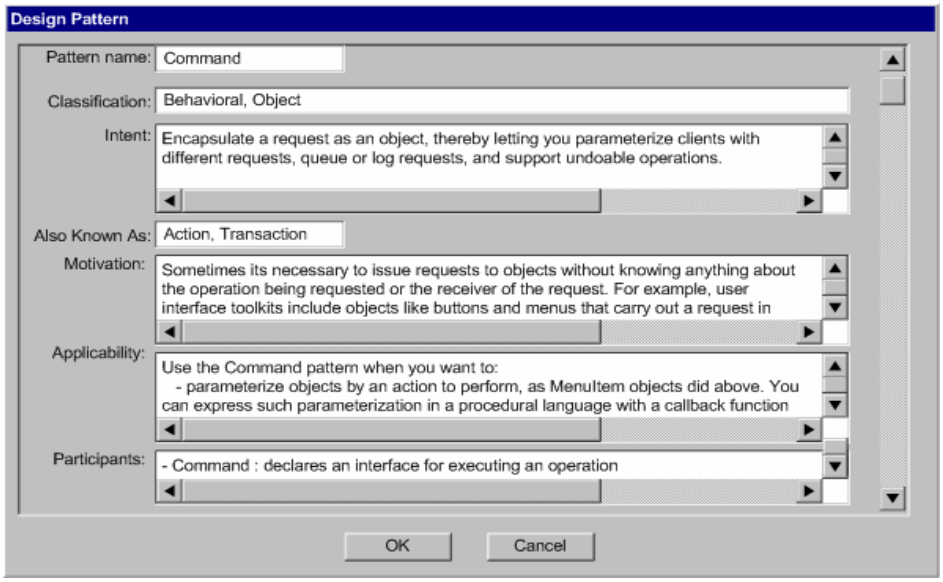


Figure 6. Patterns textual descriptions editor.

To create graphical notations, the Software Engineer uses the UML features offered by MVCASE. He can include, in the pattern description, classes and component structures, interaction models, and even executable code. Figures 7 and 8 shows, respectively, the classes structure of the *Command* pattern [11] and the interactions between the pattern objects described with graphical UML notations in MVCASE. Note that the *Execute()* operation, from the *ConcreteCommand*, has some embedded code indicating the invocation of the *Action()* operation of the *Receiver*.

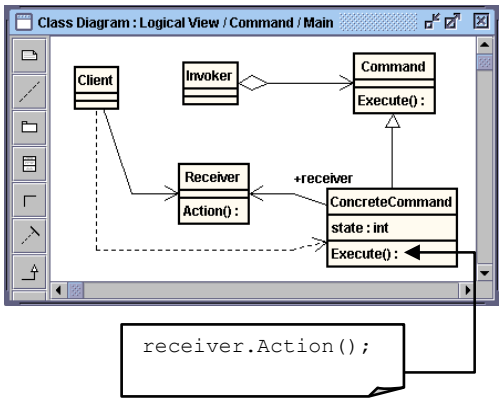


Figure 7. Classes structure and embedded code, of the "Command" pattern.

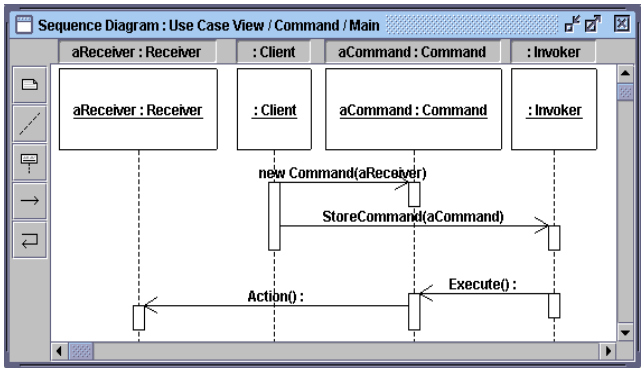


Figure 8. Interactions between the objects of the "Command" pattern, modeled through a sequence diagram.

After the creation of the textual and graphical descriptions, the pattern can be then stored in the repository, from where it can be reused. First, a unique identification is generated for each pattern, aiming to avoid redundancy and to manage pattern relationships. The pattern is then described in XMI, as shows Figure 9, for the *ConcreteCommand* class, of the *Command* pattern. In order to represent pattern-specific information, the XMI format was extended.

```
<?xml version="1.0" encoding="UTF-8"?>
…
<Class …>
  <Element.name>Command.ConcreteCommand</Element.name>
  <Class.isPublic xmi.value="true"/>
  <Class.superclass>
    <Class xmi.uuid="Command.Command" …/>
  </Class.superclass>
  <Element.contains>
    <Method …>
      <Element.name>Execute</Element.name>
      <Method.visibility xmi.value="public"/>
      <Method.dimension>0</Method.dimension>
    </Method>
    <Attribute …>
      <Element.name>state</Element.name>
      <Attribute.visibility xmi.value="public"/>
      <Attribute.type>
        <Class xmi.uuid="int" href="int.xml"/>
      </Attribute.type>
    </Attribute>
    …
  </Element.contains>
</Class>
```

Figure 9. XMI descriptions of class *ConcreteCommand*, from *Command* pattern.

Once stored in the repository, the pattern becomes available for reuse trough a network. This facilitates its reuse, since different Software Engineers, from different computers, can use the repository at the same time.

Patterns application.

In order to support the design patterns application, MVCASE allows the Software Engineer to search the repository for the existing patterns that may help him to solve the problem. The searching is performed in the following way:

1. Initially, the Software Engineer enters some keywords as the searching criteria, aiming to find a pattern based on its name and classification;

2. Next, the MVCASE repository performs a preliminary searching, comparing the keywords with the names and classification of the stored patterns;
3. The result of this preliminary searching is then presented to the Software Engineer;
4. The Software Engineer browses the presented patterns, examining each pattern fields, trying to find one that satisfies his needs; and
5. Finally, after a pattern is chosen, it is then loaded into MVCASE.

These five steps are illustrated in Figure 10.

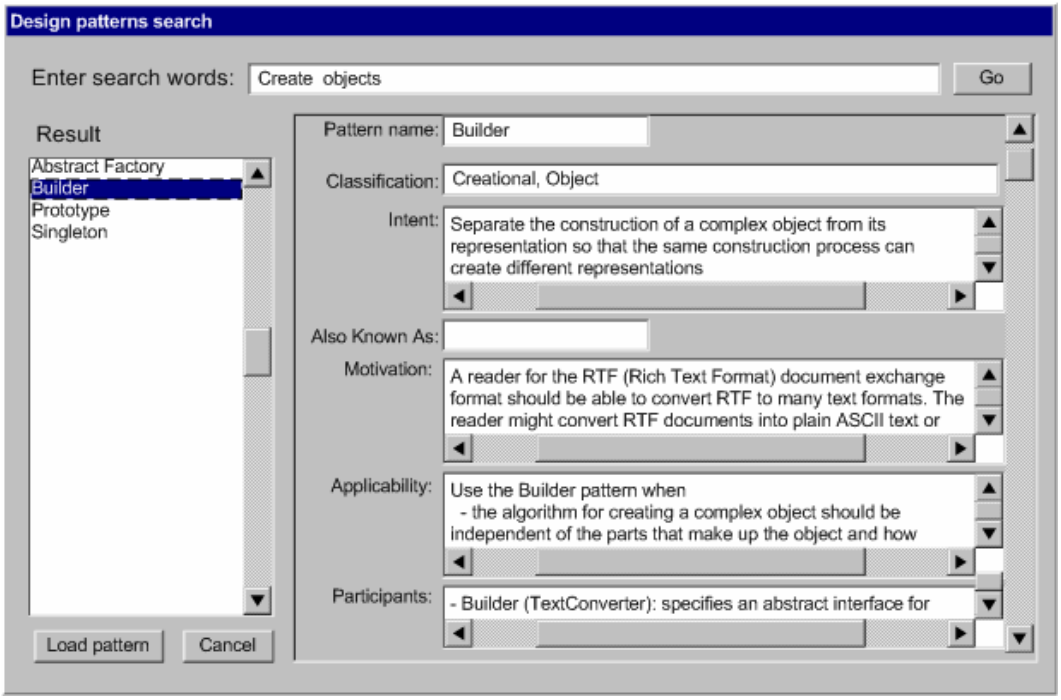


Figure 10. Design patterns search in MVCASE.

The Figure shows the Software Engineer entering the searching keywords "Create objects" (1). Next, he clicks on the *Go* button, which starts the searching engine. The engine is responsible for finding patterns that has names or classification which are similar to the keywords. In this case, *Abstract Factory*, *Builder*, *Prototype* and *Singleton* were found and displayed in a list, since they are classified as "Creational" and "Object" (3). Next, the Software Engineer browses the list, examining the information of each pattern (4). When he finally chooses one, the *Load pattern* button is pressed (5), and the pattern is loaded into MVCASE.

After finding the pattern, the Software Engineer must apply its proposed solution into the software. As seen before, the solution proposed by a design pattern is composed by textual and graphical descriptions. The graphical descriptions are usually high-level models, such as UML models, that shows the solution structure. In most cases, these models serve as a template for applying the pattern, and the Software Engineer replaces the pattern elements with the elements of the software that is being designed.

Since MVCASE works with UML, the graphical pattern descriptions are directly loaded, becoming available in the tool's browser. Once in the browser, the UML elements can be inserted into the software that is being designed, in a "drag and drop" operation. Figure 11 illustrates this process, where the *Adapter* pattern is being applied.

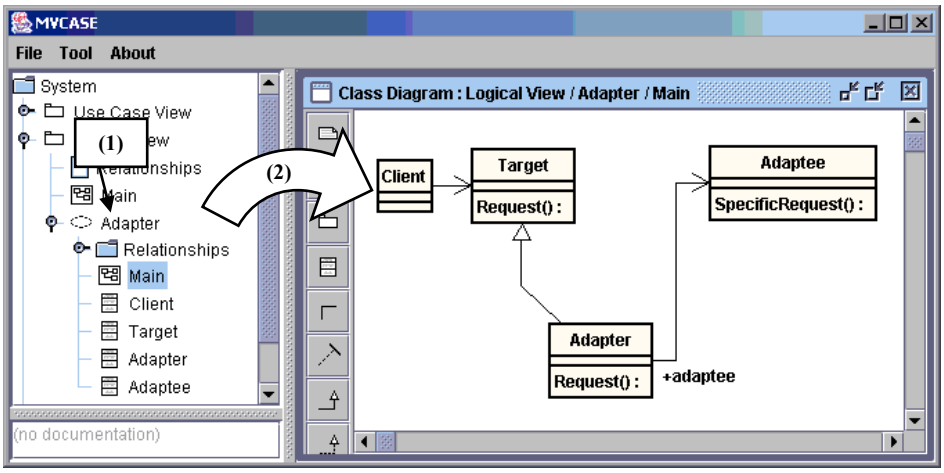


Figure 11. Pattern elements being loaded into MVCASE

In the Figure, the elements of the *Adapter* pattern are loaded into the browser (1). Next, these elements are inserted into the software's design, in a "drag and drop" operation (2). If there are relationships between the elements, they are automatically inserted. In this particular case, only the class structure of the *Adapter* pattern was inserted into the design, but other UML models could be used as well, such as sequence or component diagrams.

Once inserted in the design, the elements may be edited by the Software Engineer, through the tool features, to complete the software's design, as shown in Figure 12.

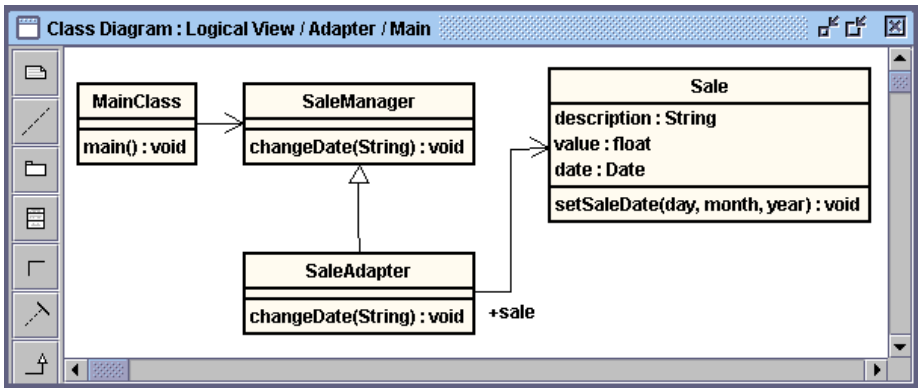


Figure 12. "Adapter" pattern being applied to a system

Figure 12 shows the *Adapter* pattern elements (see Figure 11) edited to become part of the software's design, in this case, a sales registering system. After the changes, the *Client* class became the *MainClass* class, the *Target* class became the *SaleManager* class, the *Adapter* class became the *SaleAdapter*, and the *Adaptee* class became the *Sale* class.

One important thing that must be stressed is that the code embedded in the operations is still present when the pattern is applied, since MVCASE has features to work with the code in this abstraction level. This code can also be reused, together with the design elements.

In order to indicate that a pattern was applied in some particular design, UML Collaborations are used. According to Rumbaugh et al [20], a Collaboration is a set of elements that work together to offer some behavior that is greater than the sum of its parts. This definition can be applied to design patterns, which are sets of elements that work

together to solve some particular problem. Therefore, these abstractions can be used to represent patterns in the design. Figure 13 shows how this is performed in MVCASE.

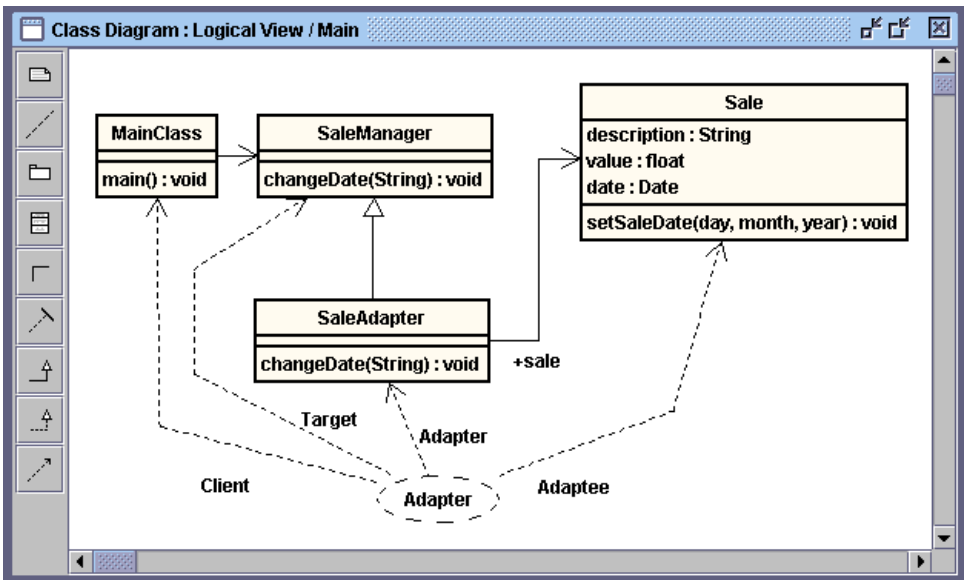


Figure 13. A collaboration to indicate that a pattern was applied

In the Figure, the dashed ellipse represents the pattern. The dashed lines indicates which software elements correspond to the pattern roles. In this case, the *MainClass* class is assigned to the *Client* role. The *SaleManager* plays the *Target* role, the *SaleAdapter* is the *Adapter* and the *Sale* is the *Adaptee*.

By doing this, the Software Engineer can identify in the design which decisions were based on patterns, and which pattern was applied. This helps to document the software, increasing its maintainability.

4. Summary

The previous section presented the main MVCASE features to support design patterns. It supports pattern creation, storage and application, during the software development process. Next a brief discussion about how MVCASE implements each requirement is presented.

4.1. Support for high abstraction level specifications

By providing UML-based techniques to perform analysis, design and implementation activities, MVCASE supports high abstraction level specifications.

4.2. Support for pattern creation

MVCASE supports the pattern creation, allowing the Software Engineer to include textual and graphical information concerning the pattern.

4.3. Patterns storage

The repository supports different kinds of software artifacts storage, in XMI format, including design patterns. To help to avoid the redundancy, an identifier is used to uniquely identify a pattern. However, the management of the relation between the stored patterns is not implemented yet.

4.4. Support for patterns searching

The searching engine uses a preliminary searching with basis on the names and classification of the patterns. Next, with basis on the results of this preliminary searching, it supports the patterns browsing, where the Software Engineer can choose one pattern by examining its information. Although it is very simple, this searching mechanism allows to identify, in a large library, the patterns that may help to solve a particular problem.

4.5. Pattern application

MVCASE is more effective in one of the three approaches for instancing patterns proposed by Florijn et al [10]: the top-down approach. In MVCASE, the elements of the pattern that is being applied are always generated, no matter if they already existed. However, the other two approaches can be used as well, only needing some manual effort to replace the existing software elements with the generated ones.

There is no mechanism implemented to help the correct instantiation of a pattern. Instead, MVCASE relies on the existing structure contained in the pattern's description. For instance, consider the case where two pattern elements must be connected through a generalization relationship. With MVCASE, this generalization is already present when applying the pattern, since its structure is copied into the design of the software that is being developed. Although this does not assure that the pattern will be correctly applied, it facilitates this task.

4.6. Pattern tracing

The pattern tracing is facilitated though the use of UML Collaborations to represent patterns application. With this feature, it is possible to identify and recognize patterns in the final design. The binding between the applied pattern and the code can also be achieved, since MVCASE works in a high abstraction level, treating the code as being an integral part of the design. Therefore, if the pattern can be traced in the design, it can also be traced in the code, since it is embedded in the design. However, although facilitated through MVCASE, the pattern tracing must be performed by the Software Engineer.

4.7. Integration with the development process

In MVCASE, development activities and pattern-related activities are performed at the same time. We believe that this is one of the most valuable contributions of MVCASE as a pattern tool. When developing a software using some tool, the Software Engineer does not want to change to another tool to apply patterns. Also, it is not likely that these tools can be easily integrated, and some manual effort may be needed to make the tools interoperable.

The use of a distributed repository, allowing different users to access the stored patterns simultaneously, also contributes for the integration with the development process, since in most of the cases there is more than one Software Engineer working in the same project.

Although most of the requirements are implemented, some implementations may require refinements and improvements in order to provide full support for design patterns. Some issues, such as the pattern tracing in a high abstraction level, are still being researched by the community, and there is no definitive solution yet. However, we believe that MVCASE offers a practical help for those who want to use patterns in their projects.

5. Related works

Since design patterns were first proposed, several researches attempts to provide tool support for the creation and application of patterns:

Gamma et al [11] present the design patterns concept, and offer clear, practical ways to describe and use them. In our work, we attempt to implement in MVCASE the principles proposed by the authors, such as the format for describing patterns and the way the patterns are selected and used.

A debate on tool and language support for design patterns is presented in [7]. In this paper, the authors discuss the extent to what languages and tools should support design patterns. In particular, the tool support for patterns is discussed. The authors state that the tools still do not fully support design patterns, mainly because the tools are too low-level. They say that a main change is needed: the raising of the conceptual level of tooling. We agree with this point of view, and believe that MVCASE is giving a step forward through this way, by treating the artifacts (particularly the patterns) in a high abstraction level, and embedding the code in the design. This gives the Software Engineer a view of the software that is closer to the "real world" than the traditional programming language view, facilitating, among other things, the use of design patterns.

One of the first attempts to offer tool support for design patterns was proposed by Budinsky et al [6]. The authors present a tool to automatically generate code for design patterns. The code is generated with basis on some specification, and saves the effort of creating the same code every time a pattern is applied. However, as one of the authors of this tool states in a later work [7], this code generation approach has many problems, including:

- although it is configurable, the generated code is not very flexible;
- generated code is often difficult to integrate with existing code; and
- when regenerating some code, important changes may be discarded.

Despite these drawbacks, this first attempt offered many contributions to today's tools, including our work. In fact, code generation is one of the main features implemented in MVCASE.

Florijn et al [10] proposes another tool to work with design patterns. The authors present some important requirements which represent what is needed in order to offer tool support for design patterns. The proposed tool helps to instantiate new patterns, bind existing elements with patterns, and check if the pattern was correctly applied. Another interesting feature of this tool is that it can be used either in forward engineering and backward engineering. The former is the usual pattern application mode. The latter is performed by identifying pattern occurrences in existing programs and modifying a program to better reflect a pattern structure. Some of these important features are not present in MVCASE, such as pattern application checking and backward engineering techniques. The reason for this is that

one of our main objectives was to evaluate the benefits of integrating development activities with patterns activities, and not providing a complete support for design patterns. However, future works shall deal with these matters.

A discussion about the use of UML to represent design patterns application is presented in [22]. In this paper, the authors discuss the use of UML Collaborations and Parameterized Collaborations to represent design patterns application. Next, they identify many limitations of these abstractions, and present what is needed in order to overcome these limitations. MVCASE uses UML Collaborations to represent patterns application. Given its limitations, this abstraction constitutes a less than perfect solution for representing design patterns in UML, and a more complete treatment is desirable. However, we decided to use UML Collaborations because it would be sufficient to satisfy our pattern tracing requirement.

In [12] a tool to apply design patterns is presented. This tool, called Fred (Framework Editor), uses the concept of *specialization pattern*, which is a specification of a recurring program structure. According to the authors, a specialization pattern is given in terms of *roles* to be played by (or bound to) structural elements of a program, such as classes or methods. Fred generates descriptions of the tasks to be performed in order to bound program elements to the pattern roles, or it can generate a default form of the element. It also allows the Software Engineer to associate program elements with pattern roles. MVCASE can only generate default specifications, not supporting tasks description or automatic binding between elements and roles. However, Fred only works with executable code, which makes it restrict to the implementation tasks. In MVCASE, instead, patterns are treated in higher abstraction levels, making it suitable for a wider range of development activities.

6. Final considerations and future works

This paper presented an extension of MVCASE tool, to support design patterns usage during the software development process. Some requirements for tool support on patterns were identified, and descriptions of the main MVCASE extensions that implement these requirements were presented.

Several issues remain to be resolved. As seen in previous sections, MVCASE provides efficient support for only one of the three approaches for applying patterns, and a more refined support for the other two should be implemented in future works. Some backward engineering ideas could be applied as well, in order to help in checking patterns application and correcting programs to reflect the patterns structures.

An important issue is the pattern representation using UML Collaborations. As already mentioned, this approach has its limitations, and a better solution is needed, in order to give the Software Engineer an unambiguous way to graphically define and document design patterns.

Currently, only patterns described in a standard format [11] are supported. We decided to adopt this format because it is well accepted and known by most of Software Engineers. However, other formats may be stored in the repository as well, needing only to add specific information in XMI format and to create specific interfaces for searching. Future works may go further in this direction, extending MVCASE's support for patterns.

Other improvements currently under development are related to the mechanisms to store and recover software artifacts (including design patterns) in MVCASE. Multi-agent technologies and more refined searching engines to improve the quality of the repository are being researched.

In this work, the main concern is to evaluate the impact of integrating development activities and pattern support activities into a single tool. Although it has many points that need improvement, the use of MVCASE to apply patterns inside our research group already indicates that this integration offers several benefits, such as the increase of productivity and wider user of the patterns.

7. References

[1] Agerbo, E., Cornils, A. **How to preserve the benefits of design patterns, In Proceedings of the conference on Object-oriented programming, systems, languages, and applications – OOPSLA'98.** Vancouver, British Columbia, Canadá, 1998.

[2] Almeida, E., S., Lucrédio, D., Bianchini, C., P., Prado, A., F., Trevelin, L., C. **Ferramenta MVCase - Uma Ferramenta Integradora de Tecnologias para o Desenvolvimento de Componentes Distribuídos**, XVI Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas, Gramado/RS, 2002.

[3] Almeida, E, S.; Bianchini, C, P.; Prado, A, F.; Trevelin, L, C. **MVCase: An Integrating Technologies Tool for Distributed Component-Based Software Development**. In: The 6Th Asia - Pacific Network Operations and Management Symposium, Proceedings of IEEE, Poster Session, 2002, Jeju Island – Korea.

[4] Albin-Amiot, H.; Guéhéneuc, Y.G. **Design Pattern Application: Pure-Generative Approach vs. Conservative-Generative Approach quality.** Proceedings of OOPSLA Workshop on Generative Programming, 2001.

[5] Barrére, T. S. **CASE com Múltiplas Visões de Requisitos de Software e Implementação Automática em Java – MVCASE**, MSc. Dissertation, Universidade Federal de São Carlos, 1999.

[6] Budinsky, F.J., Finnie, M.A., Vlissides, J. M., Yu, P. S. **Automatic code generation from design patterns**. IBM Systems Journal, 35(2), 1996.

[7] Chambers, C., Harrison, W., Vlissides, J.M. **A debate on language and tool support for design patterns**. In Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages: 277 – 289, ACM Press, 2000.

[8] Conte, A., Freire Jr, J.C., Giraudin, J.P., Hassine, I., Rieu, D. **A tool and a formalism to design and apply patterns**. In Proceedings, SugarLoaf PloP 2002, Rio de Janeiro – RJ, 2002.

[9] **Enterprise Java Beans (EJB).** Available at site Sun Microsystems, URL: http://java.sun.com/j2ee/ - Consulted in February, 2003.

[10] Florijn, G., Meijers, M., Winsen, P. van. **Tool Support for Object-Oriented Patterns**. In Proceedings, ECOOP '97, pages 472-495, Springer-Verlag, 1997.

[11] Gamma, E., Helm, R., Johnson, R., Vlissides, J.M., **Design Patterns: Elements of Reusable Object-Oriented Software**. Addison-Wesley, Reading, MA, 1995.

[12] Hammouda I., Koskimies K.: **Generating Pattern-Based Application Development Environment for Enterprise JavaBeans**. In: Proceedings of the 26th Annual International Computer Software and Applications Conference, COMPSAC 2002.

[13] Harrison, W., Ossher, H., and Tarr, P. **Software Engineering Tools and Environments: A Roadmap**. In The Future of Software Engineering. ACM, New York, 2000, 261-277.

[14] **Java 2 Platform, Standard Edition.** Available at site Sun Microsystems, URL: http://java.sun.com/products/j2se/ - Consulted in February, 2003.

[15] **Java Servlet Technology**. Available at site Sun Microsystems, URL: http://java.sun.com/products/servlet/ - Consulted in February, 2003.

[16] Orfali, R., Harkey, D. **Client/Server Programming with Java and CORBA**. John Wiley & Sons, Second Edition, 1998

[17] Prado, A, F., Lucrédio, D. **Ferramenta MVCASE -Estágio Atual: Especificação, Projeto e Construção de Componentes**, XV Simpósio Brasileiro de Engenharia de Software, Rio de Janeiro/RJ, 2001.

[18] Prado, A, F., Lucrédio, D. **MVCASE: Ferramenta CASE Orientada a Objetos**, XIV Simpósio Brasileiro de Engenharia de Software, João Pessoa/PB, 2000.

[19] Pressman, R. S. **Software Engineering: A Practitioner's Approach**, McGraw-Hill 2001.

[20] Rumbaugh, J., Jacobson, I., Booch, G. **The Unified Modeling Language Reference Manual**, Addison-Wesley, 1999.

[21] Sommerville, I. **Software Engineering (6th Edition).** Pearson Education, August 2000.

[22] Sunye, G., Guennec, A.L., Jezequel, J.M. **Design patterns application in UML**, In Proceedings of the 14 tn European conference on Object Oriented programming, Springer LNCS 1850, pages 44-62, 2000.

[23] **The Common Object Request Broker Architecture: Core Specification, Version 3.0.2.** Available at site Object Management Group, URL: http://www.omg.org/technology/documents/formal/corba_iiop.htm - Consulted in February, 2003.

[24] **Unified Modeling Language 1.4 specification**. Available at site Object Management Group. URL: http://www.omg.org/technology/documents/formal/uml.htm. Consulted in February, 2003.

[25] **XML Metadata Interchange (XMI)** – Version 1.2. Available at Site OMG. URL: http://www.omg.org/technology/documents/formal/xmi.htm - Consulted in February, 2003.

Uma Proposta de um Repositório de Padrões de Software Integrado ao RUP[1]

Fabiana Marinho[*], Misael Santos, Rute Nogueira Pinto e Rossana Andrade
[*] Instituto Atlântico, Rua Chico Lemos, 94, 660822-780, Fortaleza - Ceará – Brasil,
Universidade Federal do Ceará, Departamento de Computação, Campus do Pici
Bloco 910, 60455-760, Fortaleza - Ceará - Brasil
fabiana@atlantico.com.br,{misael,rute,rossana}@lia.ufc.br

Resumo

Com o crescimento tanto do número de padrões de software para análise, projeto e implementação quanto da reutilização dos mesmos no desenvolvimento de sistemas, surge a necessidade de facilitar a busca e aplicação desses padrões. Esse artigo visa apresentar uma proposta de integração de padrões de software armazenados em um repositório com um modelo de processo de desenvolvimento. O Rational Unified Process (RUP), tem sido utilizado comercialmente com sucesso e é o modelo escolhido para esta proposta.

Abstract

With the increase number not only of software patterns for analysis, design, project and implementation but also of pattern reuse in the system development process, there is a need for finding an appropriate mechanism for searching and applying these patterns. This work aims to present a proposal to integrate software patterns available in a repository with a development process model. Rational Unified Process (RUP) has been successfully applied commercially and it is the chosen process for this proposal.

1. Introdução

Uma grande quantidade de padrões de software de análise, de projeto e de implementação (denominados idiomas) estão disponíveis em Web sites e na literatura [4][7][12][13][20], prontos para serem reutilizados por engenheiros de software nas diferentes fases de desenvolvimento de software. Embora uma tentativa de catalogar os padrões existentes e já publicados em conferências *Pattern Languages of Programming* (PLoP) tenha sido apresentada em [23], os padrões são divididos por categoria e fica a critério do engenheiro de software fazer uma busca exaustiva para decidir o padrão correto para resolver determinado problema.

Já existem ferramentas que se propõem a facilitar a aplicação de padrões de projeto no desenvolvimento de software, auxiliando na modelagem de classes e incluindo a geração automática de código [2][3][6][8][14][15][19][27]. Apesar disso, é difícil encontrar um repositório que contenha padrões aplicáveis a diferentes fases do processo de desenvolvimento e que auxilie o engenheiro de software na busca e reutilização desses padrões de forma integrada ao processo.

Em [25], um processo de desenvolvimento baseado na reutilização de componentes nas fases de requisitos e projeto da arquitetura é proposto. Neste processo, a reutilização de um componente é limitada pelas decisões de projeto detalhadas na implementação desse

[1] O Instituto Atlântico e o CNPq suportam financeiramente este trabalho

componente. Essas decisões podem causar conflitos com as necessidades do usuário, tornando a reutilização do componente impossível ou introduzindo ineficiências significantes na aplicação que está sendo desenvolvida. A reutilização de padrões não possui este tipo de restrição. Os padrões de software são mais abstratos, podendo ser implementados de forma a se adaptar às necessidades específicas da aplicação que está sendo desenvolvida.

O objetivo deste trabalho é apresentar uma proposta de um repositório de padrões de software e a sua integração com um modelo de processo de desenvolvimento de software. O Rational Unified Process (RUP) é o processo de desenvolvimento de software escolhido. A integração proposta pode ser generalizada para qualquer modelo de processo de desenvolvimento que tenha as características de incremental e iterativo. O RUP foi escolhido por possuir as notações mais apropriadas para a reutilização de padrões nas diferentes fases de desenvolvimento, além de já estar sendo largamente adotado por inúmeras empresas nacionais.

A proposta deste artigo não pretende modificar o RUP original e nem substituir o suporte a padrões de projeto que o mesmo possui, mas oferecer aos engenheiros de software um conjunto mais abrangente de padrões de requisitos, análise e projeto que possa ser consultado e adaptado durante as fases de desenvolvimento. Para isso, é apresentado o protótipo de um repositório de padrões que será usado para catalogar os padrões que servirão de insumo para diferentes fases do RUP. A integração do padrão selecionado com o sistema que está sendo desenvolvido não vai acontecer automaticamente. O engenheiro de software será responsável por interpretar e adaptar o padrão de acordo com as suas necessidades.

Este trabalho está organizado da seguinte forma, na seção 2 apresentamos uma breve descrição dos trabalhos relacionados à nossa proposta. Na seção 3 analisamos a reutilização de padrões de software em sistemas comerciais a fim de enfatizar a motivação deste trabalho. O protótipo do repositório de padrões é descrito na seção 4. Na seção 5 descrevemos o ciclo de vida adotado pelo RUP e a nossa proposta para a integração do repositório de padrões com o mesmo. Finalmente, a Seção 6 contém nossas conclusões e direcionamentos para trabalhos futuros.

2.Trabalhos Relacionados

Um catálogo de padrões de software [23] e uma proposta de reutilização de componentes [25] foram brevemente discutidos na Seção 1. Ferramentas para o auxílio à aplicação de padrões de software que trabalham com geração automática de código e de modelagem de classes também estão disponíveis na literatura [3][6][8][19][27] e são introduzidas a seguir.

Budinsky et al apresentam em [3] uma das primeiras ferramentas para a geração automática de código para padrões. A ferramenta adota o formato apresentado em [13] para o template de padrões e gera códigos referentes aos padrões a partir de alguns parâmetros informados pelo usuário. O processo de *wizard* permite que a implementação gerada do padrão esteja adequada às necessidades do problema.

O UMLStudio [27] é uma ferramenta comercial de modelagem de classes em UML que possui um catálogo de padrões que podem ser selecionados em uma lista e inseridos na modelagem. A ferramenta inclui a modelagem dos seguintes padrões [13]: Command, Iterator, Memento, Abstract Factory, Factory Method, Adapter e Proxy. A inserção é feita de forma estática, ou seja, qualquer interação com classes já existentes, ou modificações nas classes ou nos métodos devem ser feitas manualmente após a inserção.

Uma abordagem mais dinâmica para a automação de padrões é apresenta pelo

ModelMaker [19] e pelo Together [2].

O ModelMaker, é uma ferramenta comercial com suporte à modelagem de dados em UML, geração de classes e geração de pacotes de componentes para o Borland Delphi. O ModelMaker implementa os seguintes padrões de [13]: Adapter, Mediator, Singleton, Decorator, Visitor e Observer. Além disso, ela implementa os padrões *Lock* e *Reference Count*. A ferramenta tem um nível de automação avançado, por exemplo, classes envolvidas com padrões respondem de maneira automática e inteligente às mudanças feitas pelo projetista em outras partes do modelo UML.

O CodePro Studio [6] é uma ferramenta comercial que também é integrado a ambientes Java de desenvolvimento como o Eclipse [11] e o Websphere Studio [14]. O módulo Java Pattern Wizard da ferramenta permite gerar classes implementando vários padrões de projeto e outros padrões de implementação gerais para Java. Os padrões catalogados na ferramenta são divididos em 6 categorias, a seguir: padrões de Criação, Comportamentais, Estruturais, GUI (Interface Gráfica), J2EE e Teste. Novos padrões podem ser inseridos ao repositório criando novas instâncias xml dos elementos que definem a estrutura padrões. A estrutura dos padrões definida pelo sistema é bastante limitada (id, name, icon, description, category, source e strategy), o que faz com que o usuário não tenha acesso ao padrão no seu formato original.

Outra exemplo de ferramentas que trabalham com padrões de software são as ferramentas de *Refactoring* [10][15][16], que utilizam técnicas de reestruturação de código visando aumentar a legibilidade e manutenibilidade do sistema, sem contudo alterar o seu comportamento. O *Refactoring* é realizado através da incorporação de um novo padrão ao código.

Em [8] é apresentado o protótipo de um ambiente de desenvolvimento para a definição e reutilização de padrões. No trabalho é criado um formalismo, denominado P-Sigma, para a representação de padrões e de sistemas de padrões. Além de representar os elementos de um *template* de padrões (identificador, classificação, problema, contexto, forças, solução, diagramas e conseqüências), os relacionamentos entre os padrões (usa, refina, requer e alternativo) são representados.

Vale a pena mencionar nesta seção o trabalho desenvolvido em [17], onde os autores introduzem uma ferramenta de auxílio ao desenvolvimento de sistemas baseado em componentes. O objetivo do trabalho é propiciar técnicas seguras para desenvolvimento de sistemas em grupos de trabalho, com políticas de manutenção e ferramentas de auxílio à reutilização de software. O repositório de componentes usado, além das informações técnicas dos componentes (endereço do arquivo, tamanho, criador, data de inclusão e atualização, etc.), armazena características dos componentes, tais como Contexto, Problema e Solução que compõem parte de um *template* dos padrões de software. O trabalho possui um mecanismo de busca semelhante ao proposto pelo nosso artigo. Existem dois tipos de busca, uma simples que utiliza a pesquisa entre as palavras-chave inseridas no registro de cada componente, e uma busca avançada aonde a pesquisa é realizada nas características do componente.

Em [5], as deficiências apresentadas pelas ferramentas de geração de código foram discutidas. Por exemplo, a forma de integração do código gerado ao código existente pode ser utilizado para ajudar a diferenciar as ferramentas analisadas.

Uma das diferenças entre o trabalho proposto neste artigo e os demais trabalhos referidos acima é que no nosso trabalho existe uma preocupação em dar suporte à inserção de vários tipos de padrões, não restrito a um número limitado de padrões de projeto como em [19][27]. Além disso, o engenheiro de software visualizará o padrão no seu *template* original, ficando a seu critério adaptar o padrão às suas necessidades.

Outra diferença importante é quanto à classificação dos padrões. Usamos a classificação baseada nas categorias citadas em [23], que consiste em classificar os padrões tanto quanto às fases de desenvolvimento nas quais eles podem ser aplicados (i.e., análise, projeto, implementação), quanto à natureza do padrão (e.g., comportamental, estrutural e arquitetônico), quanto ao domínio de aplicação específico (e.g., Web, interface, segurança e sistemas distribuídos), garantindo assim uma classificação mais abrangente e tornando também mais eficiente o mecanismo de busca no repositório e a aplicação dos padrões durante as diferentes fases do processo de desenvolvimento.

3. Considerações sobre a Reutilização de Padrões

Nesta seção, é analisado o impacto da reutilização de padrões dentro do processo de desenvolvimento de software no Instituto Atlântico (IA), que é uma instituição de pesquisa e desenvolvimento localizada em Fortaleza, Ceará. O IA vem desenvolvendo software em diversas áreas tecnológicas, posicionando-se como fonte inovadora de conhecimento e de geração de resultados tecnológicos. Aplicações de software para sistemas de suporte a negócios e operações, para engenharia, planejamento, inteligência de negócios, para serviços voltados à Internet e diversos outros setores de telecomunicações, de energia e do governo fazem parte do escopo de atuação do IA. O RUP é utilizado pelas equipes do IA para o processo de desenvolvimento de software.

Para identificar que padrões de software estavam sendo aplicados foram realizadas entrevistas com dois integrantes de cada um dos seis projetos (cada sistema a ser desenvolvido no IA é associado a um projeto) nas fases finais do processo de desenvolvimento. Levando-se em consideração as entrevistas, foi identificada a aplicação de padrões em todos os projetos investigados, embora em fases distintas do processo de desenvolvimento. Verificou-se, principalmente, a aplicação dos padrões de projeto da *Gang of Four* (GoF) [13], os padrões orientados a arquitetura de software (POSA) [4] e os padrões do catálogo *Core J2EE* [9].

De acordo com os resultados das entrevistas, as equipes foram estimuladas a utilizar padrões ou por membros do mesmo projeto ou de outros projetos do IA ou ainda pelos próprios clientes. A existência de um *framework* do próprio Instituto Atlântico contendo padrões de projeto já implementados funcionou como um estímulo para a aplicação de padrões.

As principais vantagens citadas durante as entrevistas sobre a reutilização de padrões estão descritas abaixo:

- evolução de código;
- modularidade;
- desacoplamento entre áreas de responsabilidades de forma que as mudanças em uma não ocasionem mudanças nas outras;
- diminuição da complexidade do projeto e do código final;
- estabilidade do código;
- os padrões em uso são genéricos o suficiente para se encaixarem bem em projetos distintos;
- confiabilidade na reutilização de padrões cujas soluções são comprovadas;
- ganho de produtividade;
- facilidade de repassar conhecimento entre os engenheiros de software experientes; e
- facilidade de aprendizado de novas áreas de conhecimento para a equipe sem experiência na aplicação a ser desenvolvida.

Esse relato do impacto positivo indicou que a reutilização de padrões funciona como um instrumento eficaz para aumentar a produtividade, diminuir o custo de desenvolvimento e promover a integração e comunicação entre projetos. Daí a importância de estimular a busca e a reutilização de padrões durante as diferentes fases de desenvolvimento de software.

Por outro lado, quando padrões são aplicados nos projetos sem um entendimento prévio da solução proposta, das conseqüências de sua aplicação e do relacionamento com outros padrões, um impacto negativo que pode ser observado é a complexidade de código com o aumento do número de classes (no caso de reutilização de padrões de projeto).

É importante ainda conservar entre os membros da equipe de desenvolvimento um senso crítico de que padrões não são soluções para todos os problemas durante o desenvolvimento de software e de que nem sempre a reutilização de padrões acarreta uma diminuição da complexidade do código final e um aumento da qualidade do produto final. Entretanto, a reutilização de padrões é uma boa prática da engenharia de software e um dos caminhos para atingir a qualidade.

A experiência relatada pelos integrantes dos projetos analisados consolidou a recomendação da comunidade de padrões de que a opção pela utilização de padrões deve ser feita nas fases iniciais de desenvolvimento para ressaltar as vantagens e evitar as desvantagens citadas anteriormente. Com a integração de um procedimento de busca em um repositório de padrões e a aplicação dos padrões recuperados nas fases iniciais de desenvolvimento de sistemas utilizando o RUP, este trabalho pretende contribuir para agilizar a reutilização de um conjunto mais extenso de padrões disponíveis na Web e na literatura. Além disso, este trabalho pretende contribuir com as eventuais discussões entre as equipes de desenvolvimento quanto à fase onde cada padrão deve ser aplicado. O IA é parceiro no desenvolvimento deste trabalho.

4. O Repositório de Padrões

No repositório, os padrões que servem de insumo para as fases do processo de desenvolvimento do RUP são armazenados. Esta seção descreve as funcionalidades que serão implementadas no protótipo do repositório de padrões e os diagramas de caso de uso para os dois atores envolvidos no repositório: o usuário e o administrador. Além disso, os mecanismos de busca de padrões utilizados são apresentados.

4.1. Definição do Repositório

As funcionalidades que serão implementadas no protótipo do repositório de padrões são descritas a seguir:

- Cadastrar linguagem de padrões: o repositório deve permitir o cadastro de linguagens de padrões. Padrões pertencentes a uma mesma linguagem devem estar relacionados de acordo com [9] que descreve uma relação entre contexto e contexto resultante em padrões pertencentes a uma linguagem de padrões. Para este caso, a partir do formalismo de [8], o seguinte relacionamento foi criado:
    - Se os padrões P1 e P2 pertencem a uma mesma linguagem de padrões L1 e possuem o relacionamento de dependência nesta linguagem P2 *depende* de P1, então o contexto do padrão P2 deve ser expresso usando o contexto resultante de P1.
- Cadastrar categoria de padrões: o repositório deve permitir a classificação dos padrões com o cadastro de categorias dos padrões. As principais categorias consideradas estão

baseadas em [4][12][13]: padrões de análise, padrões de projeto, padrões estruturais, padrões comportamentais, entre outros. Categorias específicas para o domínio das aplicações como sugerida em [23] também são permitidas, por exemplo: padrões de segurança, de interface e Web.

- Cadastrar sistema: o repositório deve realizar o cadastro de sistemas que utilizaram padrões no seu desenvolvimento. Dessa forma, será permitido ao usuário verificar posteriormente em quais sistemas um determinado padrão foi aplicado, adicionalmente aos usos conhecidos apresentados no próprio padrão.
- Cadastrar padrão: o repositório deve permitir a inclusão de padrões em uma base de dados, adotando um *template* genérico que seja capaz de englobar os vários formatos de padrões existente na literatura, por exemplo [4][9][13][18]. O *template* adotado consiste, inicialmente, de: Nome, Analogia, Contexto, Problema, Intenção, Aplicabilidade, Solução, Forças, Resumo, Implementação, Estrutura, Contexto Resultante, Conseqüências, Padrões Relacionados, Exemplos, Sintomas, Forças Contrárias, Usos Conhecidos e Referências. A intenção é dar suporte aos mais variados tipos de padrões de software existentes, mantendo o seu formato original. Esse formato semi-estruturado dos dados incentivou o uso de XML para armazenamento. Durante a inclusão dos padrões relacionados, o administrador pode definir relacionamentos de acordo com [8], que apresenta os seguintes relacionamentos possíveis entre padrões:
    - Se um padrão P_1 *usa* um padrão P_2, então a solução do padrão P_1 deve ser expressa usando P_2.
    - Se um padrão P_1 *refina* um padrão P_2, então o problema do padrão P_1 deve ser uma especialização do padrão P_2.
    - Se um padrão P_1 *requer* um padrão P_2, então aplicação do padrão P_2 é exigida na aplicação de P_1.
    - Se um padrão P1 é *alternativo* a um padrão P2, então os padrões P1 e P2 fornecem soluções diferentes para o mesmo problema.
- Modificar padrão: o repositório deve fornecer um mecanismo de atualização dos dados de um padrão cadastrado.
- Excluir padrão: o repositório deve fornecer um mecanismo de exclusão de um padrão cadastrado. Antes de excluir o padrão, a ferramenta deve verificar se existem padrões que referenciam o padrão a ser excluído. Duas opções devem ser oferecidas: exclusão das referências ou manutenção das mesmas. Em caso de manutenção, o link ao conteúdo do padrão deve ser desfeito, mas o nome do padrão permanece no item padrões relacionados.
- Realizar busca: o objetivo principal do repositório é tornar os padrões disponíveis para busca e recuperação. Alguns itens devem ser adicionados ao banco de dados para facilitar os mecanismos de busca que serão implementados no repositório de dados (veja maiores detalhes na seção 4.2.). Após a pesquisa, todos os dados e arquivos relacionados ao(s) padrão(ões) encontrados devem estar disponíveis ao usuário. Outra importante informação resultante dessa busca é a visualização de sistemas desenvolvidos que já aplicaram aquele padrão.

O administrador do sistema e o usuário do sistema são os dois atores identificados para o protótipo do repositório de padrões (veja Figura 1). O principal objetivo do usuário do repositório é reutilizar um ou mais padrões. Portanto, a funcionalidade mais importante para o usuário é a captura dos padrões dentro do repositório de dados que possam ser usados no desenvolvimento de sua aplicação (busca de padrões). O administrador deve ser capaz de

manter os padrões (cadastro, alteração e exclusão). As principais atividades do administrador e do usuário do repositório de padrões podem ser sumarizadas nos diagramas de casos de uso mostrado na Figura 1.

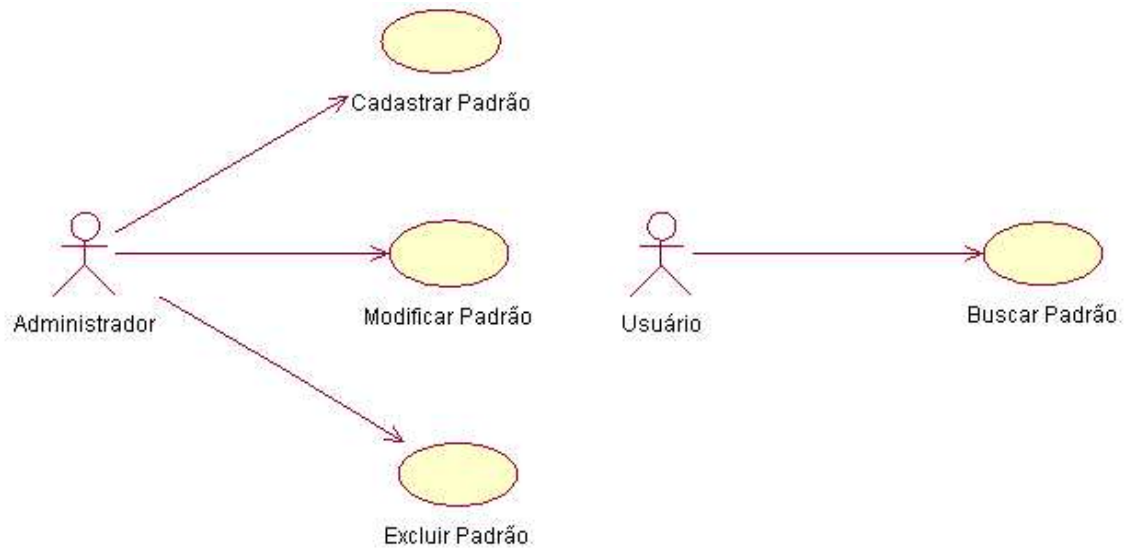


Figura 1 – Diagrama de Casos do Protótipo do Repositório de Padrões

4.2. Mecanismos de Busca

O mecanismo de busca do repositório consiste em uma busca por palavras chave entre campos selecionados do *template* dos padrões. O formalismo para a entrada da pesquisa é semelhante à maioria dos sistemas de busca da Web conhecidas como Search Engines [26]. Operadores lógicos, tais como "e" e "ou", também devem ser aceitos. O texto de consulta passa então por um processamento para que sejam definidos todos os parâmetros da consulta, palavras e operadores, que serão acompanhados dos campos de consulta selecionados.

O usuário pode realizar buscas sobre todos os padrões catalogados no repositório de padrões. Os padrões que correspondem à busca realizada são capturados e visualizados. Dois mecanismos de busca para o repositório de padrões são adotados: busca simples e busca avançada. Esses mecanismos são descritos a seguir.

- **Busca Simples**

A busca simples consiste inicialmente da pesquisa nas palavras chave do padrão a partir do texto de entrada. A Figura 2 mostra a janela do protótipo do repositório de padrões que será utilizada para a realização de busca simples. No frame esquerdo, estão localizados os parâmetros da consulta e um link para o modo de Busca Avançada. No frame direito, são listados os resultados encontrados.

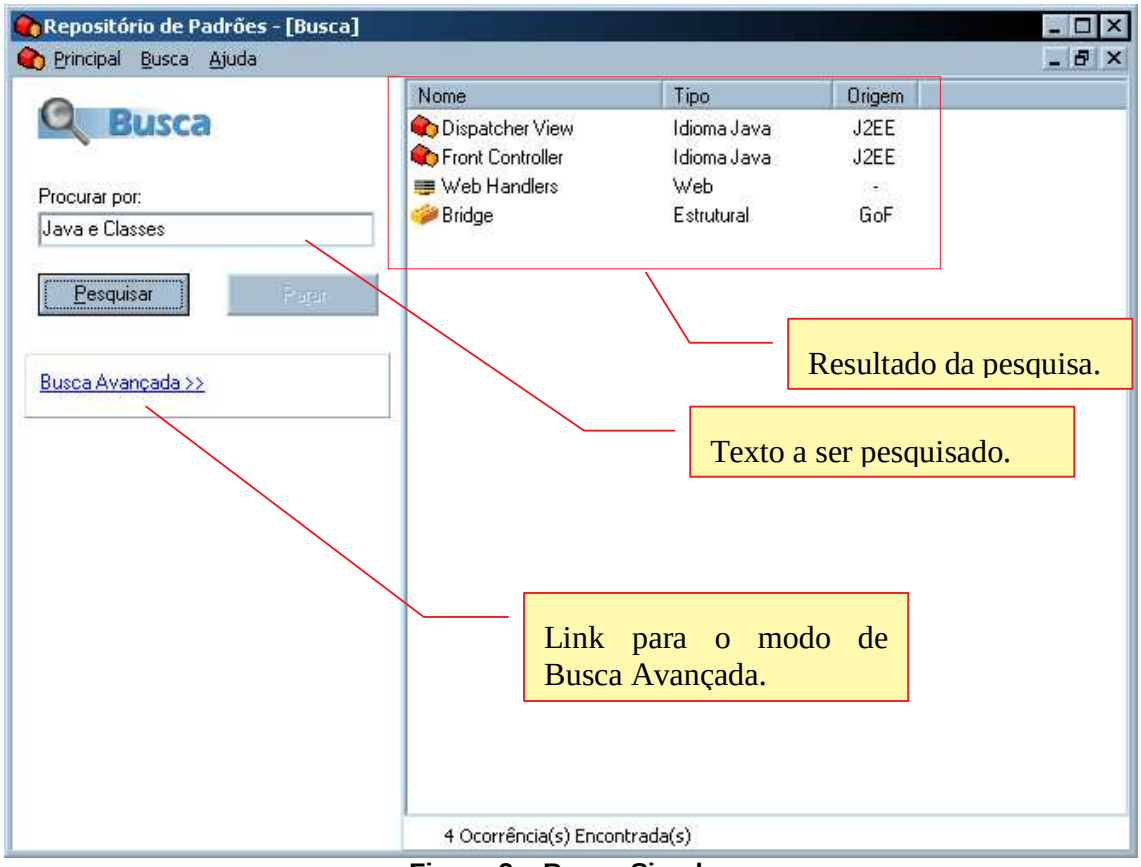


Figura 2 – Busca Simples

- **Busca Avançada**

A busca avançada permite que sejam selecionados múltiplos campos para a consulta. Pode-se ainda refinar a pesquisa selecionando a categoria do padrão procurado e/ou um sistema, a fim de buscar somente os padrões utilizados no seu desenvolvimento. A Figura 3 mostra a janela do protótipo do repositório de padrões que será utilizada para a realização de busca avançada. No frame esquerdo, ficam localizados os parâmetros da consulta que consistem no campo texto de consulta, nas opções de seleção múltipla de campos de pesquisa, na seleção da categoria do padrão, seleção de sistema e em um link para o modo de Busca Simples. Assim como na Busca Simples os resultados encontrados são listados no frame direito da janela.

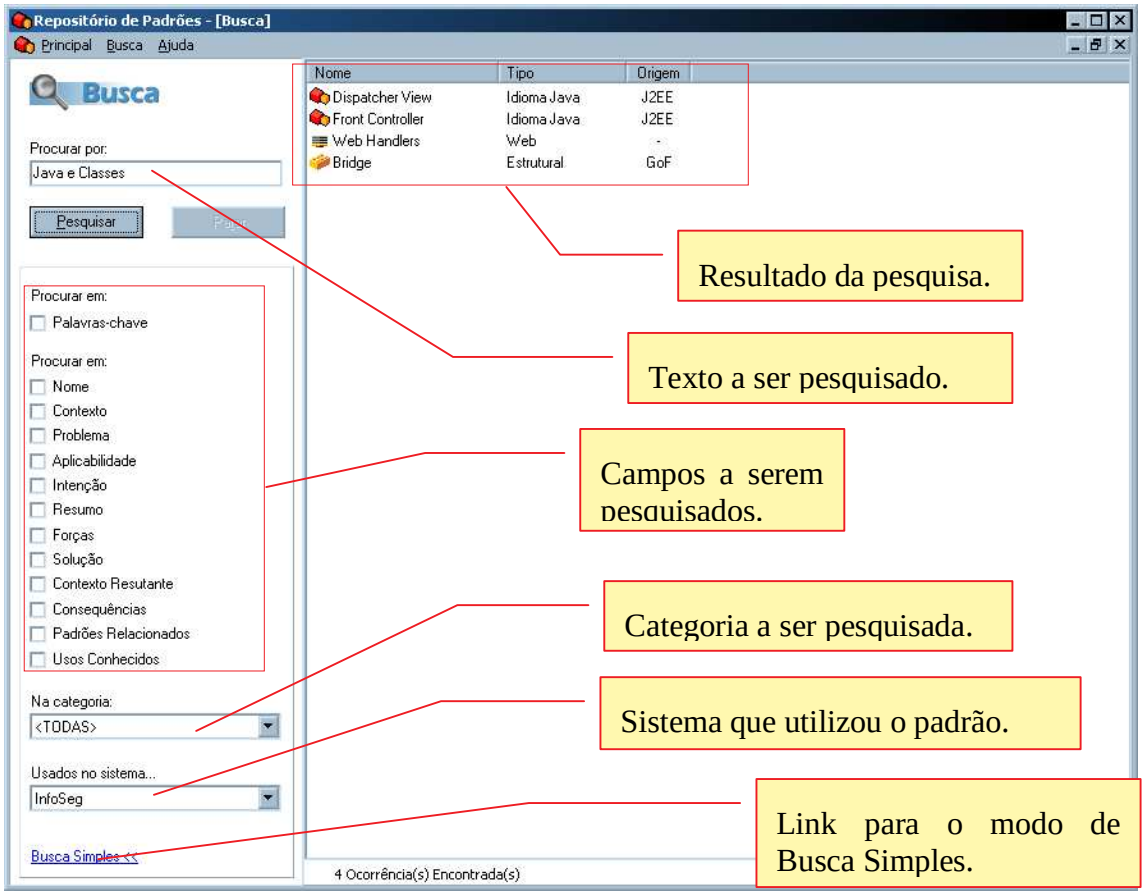


Figura 3 – Busca Avançada

5. Integração do Repositório de Padrões com o RUP

A aplicação de padrões pode ocorrer em diferentes fases do processo de desenvolvimento de software. Uma ferramenta que se proponha a armazenar os padrões existentes na literatura e que permita buscar aqueles padrões que sejam mais adequados para a solução de um determinado problema, pode facilitar bastante essa tarefa. Nesta seção, apresentamos como o repositório de padrões pode ser é integrado ao modelo de processo de desenvolvimento de software do RUP.

5.1. Rational Unified Process - RUP

O RUP é um processo iterativo e incremental que provê uma abordagem disciplinada para o desenvolvimento de software [21]. Na engenharia de software, um modelo de processo é um conjunto de passos parcialmente ordenados com a intenção de construir um produto de software de qualidade, capaz de atender às necessidades e exigências do usuário final de acordo com planejamento e orçamento previstos [25].

O RUP encoraja o controle de qualidade e o gerenciamento de riscos contínuos e objetivos. A avaliação da qualidade é inserida no processo em todas as atividades. O gerenciamento de riscos é inserido no processo de forma que os riscos que se opõem ao

sucesso do projeto sejam identificados e atacados no início do processo de desenvolvimento. Além disso, o desenvolvimento é centrado na arquitetura. Uma arquitetura robusta minimiza o re-trabalho e aumenta a reutilização de componentes e a capacidade de manutenção do sistema.

Conforme apresentado na Figura 4, o RUP possui duas dimensões. O eixo horizontal representa o aspecto dinâmico do processo e mostra aspectos do ciclo de vida à medida que este se desenvolve. O eixo vertical representa o aspecto estático do processo, como ele é descrito em termos de componentes, disciplinas, atividades, fluxos de trabalho, artefatos e papéis do processo [22].

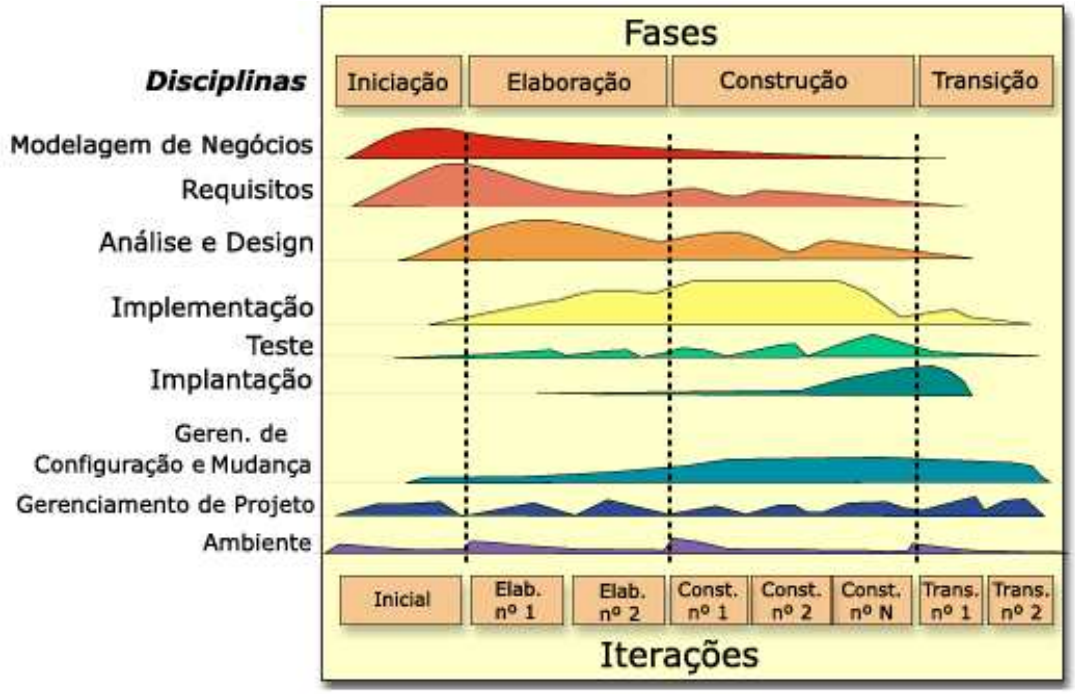


Figura 4 – Ciclo de vida de desenvolvimento do RUP[1]

O ciclo de vida de software do RUP é dividido em quatro fases seqüenciais: iniciação (termo adotado na tradução oficial da *Rational* para Português), elaboração, construção e transição [22].

O objetivo da fase de iniciação é delimitar o escopo do projeto, discriminando os principais cenários de operação. Além disso, custos, planos e riscos são estimados. A fase de elaboração tem por objetivo analisar o domínio do problema, de modo a propor uma arquitetura para os cenários da aplicação. O objetivo da fase de construção é esclarecer os requisitos restantes e concluir o desenvolvimento do sistema com base na arquitetura definida. Finalmente, a fase de transição tem por objetivo assegurar que o software esteja disponível para seus usuários finais. A fase de transição inclui testar o produto e realizar pequenos ajustes com base no feedback do usuário. No fim do ciclo de vida da fase de transição, os objetivos devem ter sido atendidos e o projeto deve estar em uma posição para fechamento. Em alguns casos, o fim do ciclo de vida atual pode coincidir com o início de outro ciclo de vida no mesmo produto, conduzindo à nova geração ou versão do produto. Para outros projetos, o fim da fase de transição pode coincidir com uma liberação total do produto a terceiros que poderão ser responsáveis pela operação, manutenção e melhorias no sistema liberado.

[1] Figura retirada do Rational Unified Process Tutorial [22]

Segundo [24], a iniciação e a elaboração abrangem as atividades de engenharia do ciclo de vida do desenvolvimento. A construção e a transição constituem sua produção. Nossa proposta para a reutilização de padrões se concentra apenas nas fases de iniciação e elaboração.

Como mostrado na Figura 4, em cada fase podem ocorrer várias iterações. Uma iteração representa um ciclo completo de desenvolvimento. O gráfico mostra como a ênfase varia através do tempo. Por exemplo, nas iterações iniciais, é dedicado mais tempo aos requisitos. Já nas iterações posteriores, é gasto mais tempo com implementação.

5.2. Integração

A Figura 5 ilustra um diagrama de atividades UML que representa a proposta de integração dos mecanismos de busca e aplicação de padrões armazenados no repositório com o RUP. O repositório de padrões aparece interagindo com as fases de iniciação e elaboração do RUP de acordo com os padrões efetivamente aplicados.

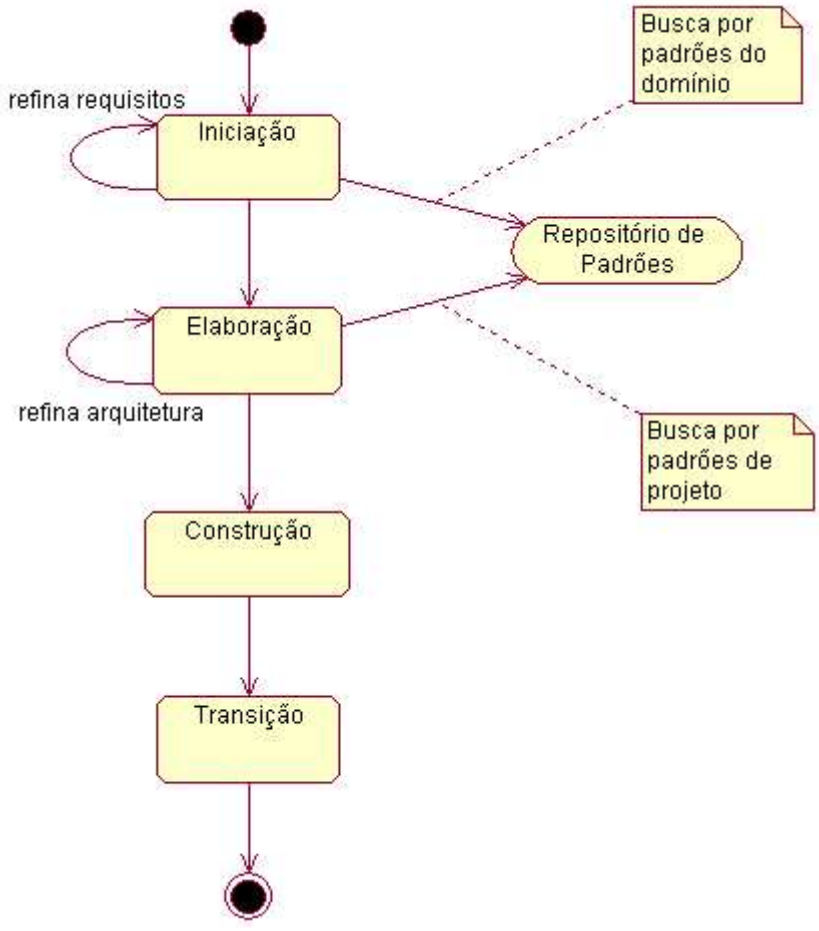


Figura 5 - Procedimento de busca e aplicação de padrões utilizando o RUP

Na fase de iniciação do RUP, o objetivo é estabelecer o escopo do projeto e as condições limite, incluindo uma visão operacional e critérios de aceitação [21]. Nessa fase,

palavras chave do domínio da aplicação são utilizadas para efetuar a primeira busca de padrões. Uma busca no repositório de padrões (veja Seção 4.1.) é realizada com o foco em padrões específicos para o domínio da aplicação. Os padrões, que corresponderem à busca realizada, são capturados e visualizados. De acordo com os padrões capturados, eles podem facilitar o entendimento e a implementação dos requisitos funcionais e não funcionais da aplicação que está sendo desenvolvida.

Na fase de elaboração do RUP, a arquitetura para suportar a solução da aplicação é proposta pela equipe de desenvolvimento. Nesse momento, uma busca no repositório de padrões com o foco em padrões genéricos para soluções de projeto é realizada. Os padrões que correspondem à busca são capturados. O projeto da aplicação pode ser refinado de acordo com os padrões selecionados. Na nossa proposta, as fases de construção e transição do RUP não sofrem nenhuma alteração.

6. Conclusão

Uma das motivações para a realização deste trabalho foi a possibilidade de facilitar a busca de padrões de software e a aplicação dos mesmos nas fases iniciais de desenvolvimento. A importância da reutilização de padrões foi inicialmente investigada através de um estudo de sistemas sendo desenvolvidos no Instituto Atlântico. Em seguida, as dificuldades encontradas na reutilização de padrões serviram de entrada para a especificação dos requisitos e posterior modelagem do repositório de padrões de software.

Este artigo então propõe um procedimento para busca em um repositório de padrões de software e a aplicação dos mesmos no desenvolvimento de sistemas utilizando o RUP. A abordagem apresentada é composta de duas fases de busca de padrões que interagem com as fases de iniciação e elaboração do RUP. Na primeira fase (i.e., fase de iniciação do RUP), a busca dos padrões é realizada com foco em padrões específicos para o domínio da aplicação que está sendo desenvolvida. Na segunda fase (i.e., fase de elaboração do RUP), padrões genéricos para soluções de projeto são selecionados. O mecanismo de busca dos padrões nestas fases utiliza um repositório que possui um catálogo dos prováveis padrões de insumo para as fases do RUP. É permitido ao usuário realizar uma busca sobre todos os padrões catalogados no repositório por meio de palavras chave.

É importante mencionar que o repositório de padrões ainda está em fase de implementação. Como discutido anteriormente, a especificação dos requisitos e a modelagem dos dados já foram feitos. Atualmente, estão sendo realizados estudos voltados para o levantamento de técnicas e algoritmos de busca disponíveis que poderão ser utilizados com eficiência no repositório. Como os templates dos padrões a serem armazenados variam de um padrão para outro, o armazenamento de dados do repositório está utilizando um servidor de banco de dados que suporta uma linguagem apropriada para trabalhar com tipos de dados não estruturados que é XML – eXtensible Markup Language.

7. Referências

[1] Borland JBuilder. Disponível em: http://www.borland.com/jbuilder. Acessado em: 03/07/2003.

[2] Borland Together. Disponível em: http://www.borland.com/together. Acessado em: 03/07/2003.

[3] Budinsky, F., Finnie, M., Vlissides, J., and Yu, P.S., "*Automatic Code Generation From Design Patterns*". IBM Research Journal, Vol. 35, No. 2 , 1996 . Disponível em http://www.research.ibm.com/journal/sj/352/budinsky.html.

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., *Pattern-Oriented Software Architecture*, John Wiley and Sons, New York, NY, 1996.

[5] Chambers, C., Harrison, W. e Vlissides, J.M., *A debate on language and tool support for design patterns*. In Proceedings In 27th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, Pages: 277 – 289, ACM Press, 2000.

[6] CodePro Studio. Disponível em: http://www.instantiations.com/codepro/ws/docs/default.htm. Acessado em: 10/06/2003.

[7] Coplien, J. O., Software Patterns, SIGS books and Multimedia, June 1996.

[8] Conte, A., Freire, J., Giraudin., J., Hassine, I., Rieu, D. A tool and a formalism to design and apply patterns. SugarLoafPLoP 2002.

[9] Core J2EE Pattern Catalog. Available at http://java.sun.com/blueprints/corej2eepatterns/. Acessado em: 03/07/2003.

[10] DPT-Tool. University of Technology Munich/Departement of CS (Informatik), Disponível em: http://dpt.kupin.de/. Acessado em: 03/07/2003.

[11] Eclipse Project. Disponível em: http://www.instantiations.com/codepro/ws/docs/default.htm. Acessado em: 02/07/2003.

[12] Fowler, M., *Analysis Patterns: Reusable Object Models*, Addison-Wesley, Reading, MA, 1997.

[13] Gamma E., Helm R., Johnson R., Vlissides J., "*Design Patterns: Element of Reusable Object-Oriented Software "*, 1995.

[14] IBM Websphere Studio. Disponível em: http://www-3.ibm.com/software/info1/websphere. Acessado em: 02/07/2003.

[15] IntelliJ IDEA. Disponível em: http://www.intellij.com/idea/. Acessado em: 30 de abril de 2003.

[16] JRefactory. Disponível em: http://jrefactory.sourceforge.net/csrefactory.html. Acessado em: 03/07/2003.

[17] Kroth, E., Pfanfeseller, M., *Uma ferramenta de apoio ao desenvolvimento de software baseado em componentes*. XV Simpósio Brasileiro de Engenharia de Software, Sessão de Ferramentas, Rio de Janeiro-RJ, outubro, 2001.

[18] Meszaros, G., Doble, J., "*A Pattern Language for Pattern Writing*," PatternLanguage of Program Design 3, edited by Robert C. Martin, Dirk Riehle, and Frank Buschmann, Addison-Wesley (Software Patterns Series), 1997.

[19] ModelMaker. Disponível em: http://www.modelmakertools.com. Acessado em: 03/07/2003.

[20] Pattern Languages of Program Design I, II, III & IV; Patterns from the PLoP Conference at Allerton Park in Illinois, US and EuroPLoP in Europe; Addison-Wesley, 1994-95-96-98.

[21] Pollice, Gary. Using the Rational Unified Process for Small Projects: Expanding Upon extreme Programming. Rational Software White Paper.

[22] Rational Unified Process Tutorial. Versão 2002 05 00.

[23] Rising, Linda, *The Pattern Almanac 2000*, Software Pattern Series, Addison-Wesley, 2000. ISBN 0-201-61567-3.

[24] Rumbaugh, J., Booch, G., Jacobson, I. *UML Guia do Usuário*. Editora Campus. 2000.

[25] Sommerville, Ian. *Software Engineering*, 6th Edition, Addison-Wesley Publishers Ltd., 2001. ISBN 0-201-39815-X.

[26] Sullivan, D., "*A Webmaster's Guide to Search Engines*", disponível em http://calafia.com/.

Acessado em: 01/07/2003.

[27] UMLStudio. Disponível em: http://www.pragsoft.com. Acessado em: 10/06/2003.

Utilização do *design pattern Architecture Configurator* em um ambiente de suporte para Configuração de Arquiteturas

Jonivan Coutinho Lisbôa, Orlando Gomes Loques Filho

Instituto de Computação – Universidade Federal Fluminense (UFF)
Rua Passo da Pátria 156 – Bloco E – 3º. Andar – 24210-240 – Niterói – RJ – Brasil
{jlisboa,loques}@ic.uff.br

Resumo

O objetivo deste artigo é mostrar que o padrão Architecture Configurator provê uma base para a concepção e implementação de ambientes de suporte à configuração de arquiteturas de software. O padrão foi proposto porque observam-se certos fatores recorrentes no processo de implantação de uma arquitetura, aplicado à configuração de sistemas baseados em componentes que interagem entre si através de requisição e fornecimento de serviços. Os conceitos propostos no padrão são aplicados na prática através de um ambiente de suporte, que permite a configuração e execução de uma arquitetura de modo transparente e com mínimo impacto no processo de implementação de seus componentes.

Abstract

This paper aims to show that design pattern Architecture Configurator provides a sound basis for conception and implementation of support environments for the configuration of software architectures. Architecture Configurator was proposed because certain recurrent facts are observed in the software architecture configuration process, when applied to systems based on components that interact with each other requiring and supplying services. The concepts proposed in the pattern have a practical application by using a support environment that allows configuration and execution of a software architecture, in a transparent fashion and with minimum impact on the basic implementation process of its components.

1. Introdução

Cada vez mais a concepção e o desenvolvimento de sistemas de software vêm se pautando em aspectos como a abstração, a separação entre interesses funcionais e não funcionais, a modularização e a reutilização de componentes em diferentes contextos de software, entre outros. Com isso, a abordagem arquitetural no projeto de software vem ganhando destaque. Aplicam-se conceitos como a descrição de arquiteturas de software por meio de linguagens de descrição arquitetural (ADL – *Architecture Description Language*), e a utilização de padrões para representar modelos recorrentes de software, e também para modelar o processo de implantação da configuração de um sistema [Schmidt *et al.*2000].

Um sistema de software pode ser definido a partir de seus componentes e das interações entre eles [Shaw, Garlan 1996]. Os componentes funcionais do sistema são chamados

comumente de *módulos*. Os módulos do sistema carregam em si a funcionalidade do sistema, e podem ser representados pelas classes de aplicação, procedimentos, funções e programas executáveis em si. Num sistema, os módulos podem desempenhar o papel de *servidores*, quando executam alguma ação concreta no sistema (serviço), e de *clientes*, quando coordenam a ação dos servidores, requisitando apropriadamente os serviços oferecidos. Em alguns casos, um módulo pode ao mesmo tempo desempenhar os dois papéis.

Os módulos de um sistema podem ser interligados diretamente entre si, ou através de *conectores*, que encapsulam os requisitos não-específicos do sistema, e permitem com isso uma maior flexibilidade na execução do mesmo. Os conectores mediam os contratos de interação entre os módulos, e possuem similaridade com os módulos, pois podem ser escritos em alguma linguagem de programação, sendo representados por objetos, classes, ou mesmo por alguma facilidade oferecida por um *middleware* ou *framework* operacional [Loques *et al.* 2000].

As ligações entre módulos são definidas em função de pontos de interação específicos – as *portas*. Estas são responsáveis pela ligação real entre componentes e/ou conectores. Existem dois tipos de porta: portas de saída, que são representadas pela requisição de um serviço, e portas de entrada, que são as assinaturas dos métodos disponíveis em um módulo servidor, e que podem ser encontradas na definição da interface de tal módulo.

Uma arquitetura Cliente-Servidor simples, com a interação entre o Cliente e o Servidor mediada por um conector, pode ser representada como na figura 1. Notem-se a porta de saída (*request*) e a porta de entrada (*provide*).

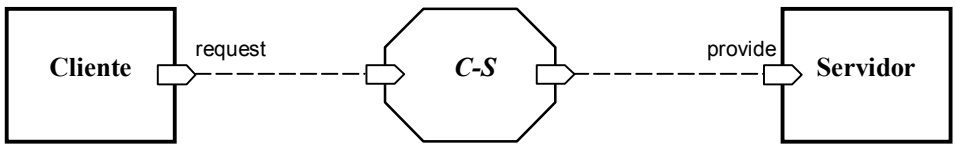


Figura 1. Uma arquitetura Cliente-Servidor simples

Pode-se dizer que a configuração de um sistema consiste em duas fases: a instanciação dos módulos funcionais (os componentes do sistema) e eventuais conectores que possam ser utilizados, e a realização das ligações entre os módulos e/ou conectores, através da conexão de portas de entrada e saída apropriadamente, de acordo com a descrição da topologia do sistema. Esse processo de configuração pode ser entendido como a implantação de uma arquitetura de software específica, segundo a qual o sistema foi modelado.

2. O padrão de projeto *Architecture Configurator*

O padrão *Architecture Configurator* foi proposto porque observam-se certos fatores recorrentes no processo de implantação de uma arquitetura de software. Tal padrão fornece um suporte para a implementação de configurações arquiteturais, fundamentando-se nos mecanismos de interceptação, encaminhamento e manipulação de requisições realizadas entre módulos de uma aplicação, e também na interligação entre módulos e conectores. Esses mecanismos são aplicados de forma transparente em relação aos elementos básicos da arquitetura (componentes, conectores e portas), e suportam propriedades como reutilização, abstração e separação de interesses. Deste modo, os módulos e conectores participantes de uma arquitetura podem ser implementados de forma autônoma e integrados de acordo com a configuração arquitetural.

Uma descrição detalhada do padrão pode ser encontrada em [Lisbôa, Carvalho, Loques 2002]. De modo geral, o padrão pode ser aplicado a arquiteturas genéricas que apresentem componentes que fornecem e requisitam serviços. Os requisitos não-funcionais são encapsulados através de conectores configurados na arquitetura. Como ilustração da utilização do padrão, o diagrama da figura 2 mostra a solução apresentada para a implementação de uma arquitetura Cliente-Servidor simples.

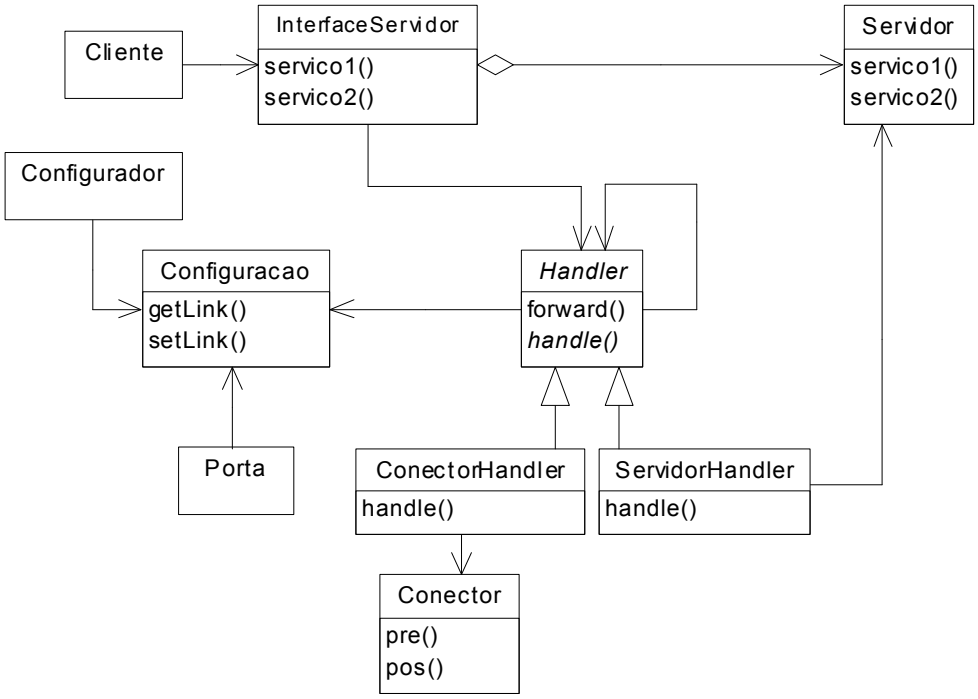


Figura 2. Diagrama de classes de uma possível solução para a arquitetura Cliente-Servidor utilizando *Architecture Configurator*

As classes `Cliente` e `Servidor` representam os módulos funcionais e a classe `ConectorHandler` representa o conector da arquitetura. A programação da configuração está representada pela classe `Configuracao`, que mantém a informação de descrição arquitetural.

`Configurador` é uma classe que define um mecanismo que interpreta as instruções de uma ADL, e a partir da descrição interpretada, define os tipos para componentes, conectores e portas presentes na aplicação.

A classe abstrata `Handler` é responsável pelo encadeamento entre conectores e componentes, conforme a descrição da arquitetura. No modelo, as classes `ConectorHandler` e `ServidorHandler` são encadeadas por `Handler`. `ServidorHandler` representa a classe `Servidor` no encadeamento e possui uma referência a esta última.

As requisições feitas por `Cliente` são interceptadas por `InterfaceServidor`, que busca em `Configuracao` a referência ao próximo conector `ConectorHandler` e invoca a operação *`handle()`* do mesmo.

Conforme a porta de entrada configurada no conector, *handle()* invoca a operação adequada. Cada operação descrita no conector invoca *forward()*, responsável por dar seqüência ao encadeamento controlado por Handler. Na seqüência, ServidorHandler tem sua operação *handle()* solicitada, a qual encaminha a requisição original para Servidor, finalizando o encadeamento.

A aplicação do padrão para arquiteturas genéricas (múltiplos clientes, servidores e conectores) é possível designando-se para cada conector configurado na arquitetura um manipulador específico, acontecendo o mesmo para os servidores. O encaminhamento das requisições terá como base as ligações descritas na arquitetura. Essa informação é consultada no momento da chamada *forward()*, de modo que as solicitações feitas por um cliente sigam seu caminho correto na seqüência de componentes configurados, até chegar ao servidor ligado a ele.

Para ilustrar a participação das classes no padrão, a seguir é apresentada a tabela 1, com as respectivas responsabilidades e colaborações.

Tabela 1. Resumo das classes participantes em *Architecture Configurator*, suas responsabilidades e colaborações

Classes	Responsabilidades	Colaboradores
Cliente	Utiliza a interface fornecida por Interface Servidor para requisitar um serviço particular;	InterfaceServidor
Servidor	Implementa um ou mais serviços particulares;	-
InterfaceServidor	Fornece a interface do servidor aos clientes; Recupera a referência do conector interligado ao cliente no nível da configuração, ou do próprio servidor, caso não haja nenhum conector envolvido; Repassa a solicitação do cliente ao conector recuperado, ou ao servidor, no caso de não haver conectores; Retorna ao cliente resposta oriunda do servidor;	Servidor Configurador ConectorHandler
Handler	Serve como classe abstrata base para o servidor e para os conectores; Realiza o encadeamento de conectores e componentes a partir da configuração estabelecida;	Configuracao
ConectorHandler	Implementa serviços relacionados à funcionalidade de um conector; Invoca uma de suas operações correspondente à porta de entrada requisitada; Requisita junto ao próximo conector configurado a operação correspondente a uma de suas portas de saída, através da operação *forward()* da classe Handler.	Configuracao

Tabela 1. Resumo das classes participantes em *Architecture Configurator*, suas responsabilidades e colaborações (cont.)

Classes	Responsabilidades	Colaboradores
ServidorHandler	Encaminha ao servidor a requisição vinda originariamente do cliente;	Servidor
Configurador	Define a arquitetura da aplicação a partir de uma ADL; Recebe instruções a partir de uma determinada ADL e invoca serviços da classe Configuração para executá-las;	Configuracao
Configuracao	Fornece a configuração estabelecida entre componentes e conectores; Disponibiliza serviços para configurar componentes e conectores e iniciar a aplicação; Mantém a descrição da arquitetura, bem como informações quanto à execução da mesma;	Porta
Porta	Fornece as portas configuradas de conectores e componentes com suas respectivas assinaturas;	-

3. Utilização de *Architecture Configurator* em um Ambiente de Suporte

3.1. Considerações iniciais

Com base em *Architecture Configurator*, um Ambiente de Suporte para configuração de arquiteturas deve possuir uma estrutura que consiga, a partir da interpretação de comandos presentes na descrição arquitetural, escrita em uma ADL, obter a informação de descrição do sistema, armazenando-a em estruturas de dados que permitam sua recuperação posterior, e também manter a informação de execução do sistema, que é derivada da descrição – as referências a instâncias de componentes e interligação entre elas. A partir dessa informação de execução, o ambiente deve instanciar os módulos e conectores do sistema, realizando as ligações entre eles e efetivando a execução do sistema.

A implementação de um Ambiente de Suporte pode ter como ponto de partida o diagrama de classes apresentado na figura 2, no qual é feita uma descrição de *Architecture Configurator*. Basicamente, essa implementação consiste em:

- Criação do módulo `Configurador` para agir sobre as classes `Configuracao` e `Porta`, alimentando seus objetos com os dados referentes à arquitetura do sistema;

- Elaboração de estruturas de dados para armazenar a situação arquitetural do sistema – os módulos, portas, instâncias e ligações presentes na topologia do mesmo. Esse é o papel da classe `Configuracao` em *Architecture Configurator* ;
- Implementação das funcionalidades de interceptação e encaminhamento de requisições entre objetos `Cliente` e `Servidor`, realizadas pelas classes `Interface Servidor` e `Handler`.

A figura 3 destaca as classes de *Architecture Configurator* envolvidas na implementação do Ambiente de Suporte.

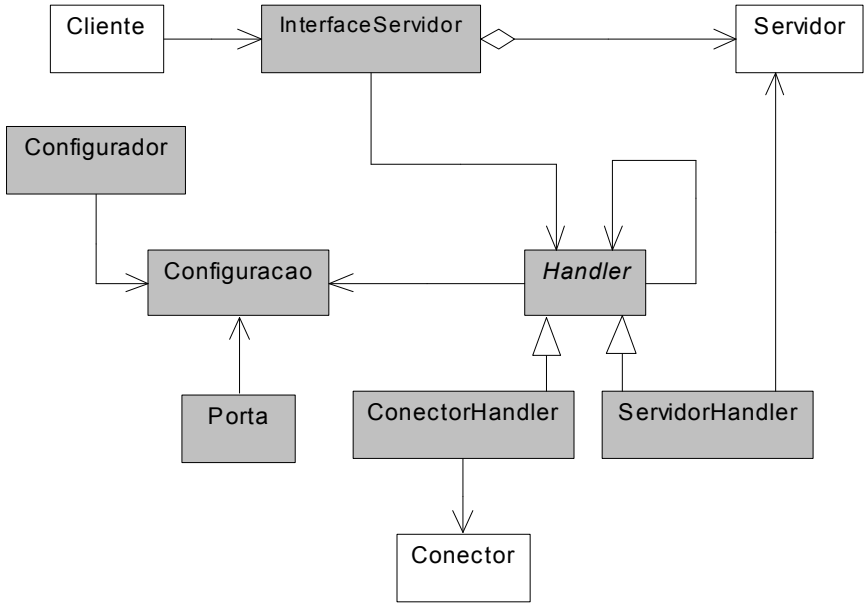


Figura 3. Diagrama simplificado de *Architecture Configurator* mostrando as classes envolvidas na implementação de um Ambiente de Suporte

3.2. Uma proposta de implementação

Após a definição dos pontos de integração de *Architecture Configurator* com um ambiente de suporte, é apresentada aqui uma proposta para implementação prática. Deve ser salientado que a abordagem utilizada refere-se a uma maneira de implementar um ambiente de suporte, não devendo ser entendida como uma forma absoluta e inflexível para a obtenção dos resultados desejados.

A implementação de um Ambiente de Suporte visa a consolidação dos conceitos propostos em *Architecture Configurator*. O ambiente proposto para tal foi concebido como uma composição de três módulos de execução:

- um Módulo de Linha de Comando, que recebe solicitações do usuário e contém o interpretador da ADL. A partir da interpretação de uma descrição é gerada a informação de configuração, que será repassada ao Coordenador;

- um Módulo Coordenador, que recebe mensagens do módulo de Linha de Comando para atualização da configuração (no caso de interpretação de uma descrição) e controle de execução da aplicação. O Coordenador contém as estruturas de dados da classe `Configuracao`, que serão distribuídas para cada módulo Executor presente no ambiente;
- um Módulo Executor, responsável pela instanciação e execução dos módulos funcionais da arquitetura configurada. Pode ser replicado no ambiente para a execução distribuída de componentes, em vários *hosts* de rede.

A figura 4 ilustra a estrutura para a implementação do Ambiente de Suporte proposto neste artigo.

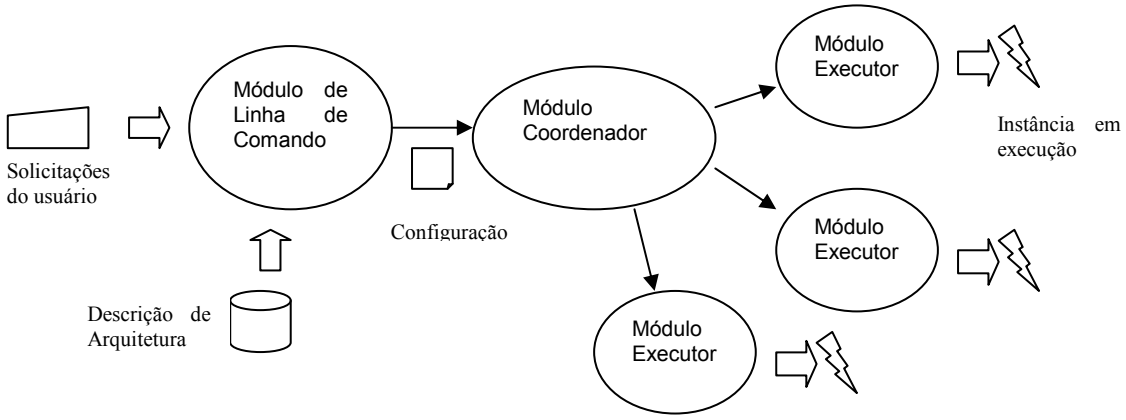


Figura 4. Estrutura proposta para o Ambiente de Suporte à configuração de arquiteturas

3.3. Descrição dos módulos de execução

O módulo de Linha de Comando possui duas funções básicas:

- ***Interpretar a descrição arquitetural escrita em uma ADL:*** é a função da classe `Configurador` de *Architecture Configurator*. Através da interpretação da descrição, é criada a informação sobre a topologia da aplicação, a ser armazenada na classe `Configuracao`. Na proposta apresentada, a função da classe `Configuracao` é feita por uma estrutura de dados que envolve informações sobre os módulos, portas, instâncias e ligações. Essa estrutura pode ser implementada, por exemplo, na forma de tabelas de símbolos, tabelas dinâmicas ou qualquer outro tipo de repositório de objetos, de acordo com a linguagem de programação utilizada.
- ***Receber a intervenção manual do usuário sobre a execução do sistema:*** assim, é possível a interação com a execução da aplicação, permitindo, entre outras coisas, a reconfiguração em tempo de execução.

O módulo Coordenador possui a função de receber solicitações do módulo de Linha de Comando, que podem ser de duas naturezas: o carregamento de uma nova descrição, ou a intervenção do usuário para mudar o estado da aplicação. No primeiro caso, o Coordenador recebe a estrutura de dados com a informação de configuração e a distribui para os módulos

Executores. No segundo caso, o Coordenador interfere na execução das instâncias de componentes do sistema.

O módulo Executor cuida efetivamente da instanciação e execução dos componentes da aplicação. Sua atividade pode ser dividida em três etapas:

- ***Armazenamento da configuração:*** o Executor mantém uma cópia da configuração da aplicação. Essa cópia lhe é enviada pelo Coordenador assim que o mesmo recebe a informação de descrição vinda do módulo de Linha de Comando.
- ***Instanciação de componentes:*** no momento da inicialização da aplicação, o Coordenador envia ao Executor um sinal para a criação das instâncias de componentes que estarão hospedadas em seu *host*, e que serão gerenciadas por ele. O Executor então procede com a criação das instâncias dos componentes, do *proxy* do servidor (`InterfaceServidor`) e dos manipuladores de componentes (`ConectorHandler` e `ServidorHandler`).
- ***Encaminhamento de solicitações:*** as solicitações são encaminhadas segundo o encadeamento de componentes previsto na descrição arquitetural. Cabe ao Executor, consultando a informação de descrição que mantém, descobrir o próximo componente presente do encadeamento, para que a solicitação feita possa ser tratada e encaminhada adiante. É a função da classe `Handler` de *Architecture Configurator*.

4. Conclusão

A implementação da proposta apresentada neste artigo permite que seja atingido o objetivo de aplicar na prática os conceitos propostos em *Architecture Configurator*. Com a implementação dos pontos principais propostos no padrão – um interpretador para a descrição arquitetural, um esquema para interceptação e encaminhamento de requisições e um esquema para gerenciamento da configuração através de estruturas para armazenamento e consulta – torna-se possível a obtenção dos principais requisitos desejados na aplicação de arquiteturas: reutilização de software, abstração, separação de interesses, flexibilidade, extensibilidade, entre outros [Lisbôa 2003].

Em comparação com outras abordagens existentes, a utilização de *Architecture Configurator* em um ambiente de suporte tem como diferenciais:

- A disponibilização da informação de descrição arquitetural, armazenada em estruturas de dados, permite que esta seja utilizada para análises estruturais de comportamentais do sistema (verificação de propriedades, validação, testes, etc.), e também guiar atividades de reconfiguração dinâmica, de modo a adaptar o sistema a mudanças no ambiente operacional, através de, por exemplo, contratos de qualidade de serviço diferenciada [Cerqueira *et al.* 2003]. Essa possibilidade não é contemplada em abordagens como as utilizadas no padrão *Component Configurator* [Schmidt *et al.* 2000], e no servidor JBoss [Fleury, Reverbel 2003], por exemplo. Nestas, são privilegiados somente os aspectos de ligação entre componentes, sem haver uma preocupação com a arquitetura.
- O esquema de interceptação e encaminhamento de requisições é encapsulado no ambiente, de modo que os componentes do sistema não precisam ter seu código adaptado para sua utilização, que é feita de maneira transparente. Isso aumenta a

possibilidade de reutilização de componentes. No servidor JBoss, por exemplo, os clientes devem ter suas chamadas modificadas para a interação com um componente específico do ambiente (o *MBeans*), que faz a ligação entre os clientes e o *middleware* de suporte à configuração.

O emprego de arquiteturas de software de forma transparente em aplicações baseadas em interação de componentes cliente-servidor, com a utilização do Ambiente de Suporte, faz com que *Architecture Configurator* funcione como um conector de configuração de software, pois ocorre a intermediação da interação entre cliente e servidor, com isolamento e separação de interesses. Na verdade, todos os requisitos não-específicos da aplicação implementados pelo encadeamento de conectores configurado no padrão fazem com que ele funcione como uma composição de conectores, conforme ilustrado na figura 5.

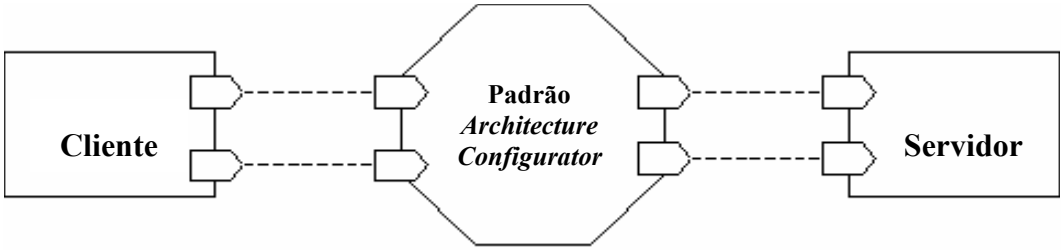


Figura 5. *Architecture Configurator* funcionando como um conector para configuração de arquiteturas

5. Referências

Cerqueira, R. *et al.* (2003) "Deploying Non-Functional Aspects by Contract". The Second Workshop on Reflective Adaptative Middleware, International Middleware Conference – MW2003, junho.

Fleury, M., Reverbel, F. (2003) "The JBoss Extensible Server". The Second Workshop on Reflective Adaptative Middleware - MW2003, junho.

Lisbôa, J., Carvalho, S. e Loques, O. (2002) "Um *Design Pattern* para Configuração de Arquiteturas de Software". SugarLoafPLoP 2002 Proceedings, impresso por ICMC-USP, dezembro, p. 37-54.

Lisbôa, J. (2003) "Utilização do *Design Pattern Architecture Configurator* em um Ambiente de Suporte à Configuração de Arquiteturas". Dissertação de Mestrado. IC/UFF, fevereiro.

Loques, O. *et al.* (2000) "On the Integration of Configuration and Meta-Level Programming Approaches". Lecture Notes in Computer Science, vol. 1826. Cazzolla, W. Stroud, R. Tizato, F. (eds). , p. 189-208.

Schmidt, D. *et al.* (2000) "Pattern Oriented Software Architecture Vol. 2: Patterns for Concurrent and Networked Objects". Wiley & Sons.

Shaw, M. e Garlan, D. (1996) "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall.

Structural Modeling of Design Patterns: REP Diagrams

José Luis Isla Montes[1], Francisco Luis Gutiérrez Vela[2]

[1]Departamento de Lenguajes y Sistemas Informáticos – University of Cádiz
E.S. Ingeniería – c/ Chile, 1 – C.P. 11002 – Cádiz – España

[2]Departamento de Lenguajes y Sistemas Informáticos – University of Granada
E.T.S.I. Informática – Periodista Daniel Saucedo Aranda s/n – C.P 18071 – Granada – España

joseluis.isla@uca.es, fgutierr@ugr.es

Abstract

Software modelling languages should possess a notation, semantics and treatment adapted to design patterns, but its correct specification is a challenge for many investigators. UML treats them as parameterised collaborations, however, this approach is not exempt of problems. The goal of this work is the elaboration of a simple and intuitive model for the structural specification of design patterns and its integration in UML. The patterns specified with this model expect to be true reusable templates that can be applied in different contexts. For that, we use a visual, complete, simple and easy to learn notation. We believe that this model could be used successfully for the construction of a future tool that facilitates the definition, application, visualization and validation of patterns.

1 Introduction

Design patterns have acquired great popularity among researchers and software designers. The reason of their great success is that they form a common vocabulary of good solutions applicable to typical design problems that can appear in different contexts. Design patterns facilitate the reuse of the expert knowledge as “design components", improving the documentation, understanding and communication of the final design.

We think that the software modeling languages should possess an appropriate notation, semantics and treatment that facilitates the use of the design patterns as modeling elements of first order. On the other hand, the tools of software design based on these languages should facilitate their definition, application and validation in specific contexts.

In the following section we will summarize several important works that have treated the specification of design patterns and we will justify the interest of this new approach. In the third section we will show some of the inconveniences that UML presents to represent the essence of a pattern appropriately. Next, in the fourth section, we will try to summarize the new philosophy of the specification model, defining the main concepts and its relationships. As an example, in the section fifth, we show the application of the new approach for the specification of the Abstract Factory pattern [5]. In the last section we will present the conclusions, as well as the questions that are still to explore.

2 Works about design patterns specification

The description that Gamma et al. [5] make of their patterns is good, however, it is mainly narrative and it is based on concrete examples expressed with OMT. These are obstacles for the treatment with a tool.

It is necessary to find a notation that has the capacity to specify, in a precise way, the essence/invariant of a pattern. That is to say, the group of structural and behavioural restrictions that characterize it and whose instances it will have to satisfy.

Several interesting works exist in this line, among these it is necessary to highlight:

- The formal language "LePUS" (**L**anguag**e** of **P**atterns **U**niform **S**pecification) [1][2][3][4] which abstracts from the implementation elements (objects, attributes, methods, etc.). It uses logical expressions and a semantically equivalent visual notation starting from a simple universe where the classes and the functions are atomic entities. LePUS allows to define relationships among groups of these entities (inheritance, reference, etc.), which should exist in a program or model to be able to conform with the specification.
- According to Lauder and Kent [8], the specification of a pattern should be divided in three levels (roles, types and classes) ordered from the greater to lesser level of abstraction. The first level captures the essence of the pattern obviating specific details of the application domain, the second refines the previous one usually adding specific restrictions of the domain and the last represents a concrete implementation. A visual notation denominated “constraint diagrams” [7] is used for the precise modeling of the static and dynamic structure of the patterns. These are used in combination with another modeling graphic notation, in this case UML.
- The “Fragment Model” of Meijers [10] in which a pattern is divided in fragments connected by roles. A fragment would be each one of the outstanding components of a pattern and a role, a property of a fragment, would connect with the fragments that are actors of that role.
- The “Role Diagrams” of Riehle [12] where the patterns are represented as collaborations among roles (a specific view of the responsibilities of an object) as abstraction from the class diagrams.
- Lastly, the interesting works of Le Guennec et al. [9] and Sunyé et al. [13] centered on UML. They modify the metamodel to use stereotyped occurrences of patterns with OCL restrictions in the meta-level and they use the base of the stereotype as context (the metaclass PatternOccurrence). Their ideas are transferred to the UMLAUT tool.

In this work we present an alternative model of specification for the patterns, based on what we denominate REP Diagrams. Contrary to the other approaches, with this model it is possible to carry out a precise and visual specification for the structure of a pattern by means of a simple extension to UML, without necessity of appealing to a formal notation such as OCL. The result is a familiar, simple, complete and very easy to learn notation for the designer and a model that treats the design patterns as true templates. For these reasons, we believe that the construction of a future tool based on this model would be of a great interest.

3 The case of UML

As it is already known, UML [11] treats design patterns as parameterised collaborations, however, this treatment presents some limitations to express the semantics associated to a pattern correctly, mainly because initially these have different purposes.

A collaboration simply seeks to model the interaction among a group of instances to achieve a specific task. However, a pattern has a more general purpose, it should specify what the structure and behaviour of a design solution is. In fact, the instantiation of a parameterised collaboration originates another collaboration, but the instance of a design pattern is a design portion that should satisfy the requirements of the pattern and solve a concrete problem, it may even be purely structural and not include interaction.

A pattern should be used like a template to guide the designer in the flexible selection, creation and validation of the elements (classes, objects, relationships, attributes, methods, etc.,) that participate in the solution of a typical design problem. However, the "*ClassifierRole*" or "*AssociationRole*" of a parameterised collaboration depend on base classifiers or associations that should exist previously and where the "*ClassiferRole*" or "*AssociationRole*" are respectively a view of these. Also, they can only bind classifiers or associations to one parameter with the only restriction that the participant is of the same type (or a descendant of the type) as the corresponding parameter. On the other hand, it is impossible to specify patterns with a variable number of participants, the most frequent type, because the number of arguments should be similar to the number of parameters.

These and other inconveniences are an obstacle to be able to specify the essence of a pattern appropriately. For this reason we present the following approach.

4 Proposal for the modeling and treatment

In this work we propose a different model for the structural specification of design patterns and their integration with UML [6]. For this, we take advantage of the expressive capacity of a very well-known notation (UML) and we add a minimum group of modeling elements that allow us to define the invariant of a pattern establishing the appropriate mechanisms to check it.

This proposal arises from the idea that a pattern should help the designer in the flexible assignment, creation and validation of the different elements of modeling that participate in the solution.

Figure 1 shows a basic outline with the fundamental concepts that have been defined.

A *role-element* is the specification of a design element that will play a fundamental role in the solution proposed by a pattern. A role-element defines the type (class, object, attribute, operation, parameter, relationship, etc.) and the restrictions (structure and/or behaviour and/or relationships) that will satisfy this design element.

The role-elements are key pieces to model the restrictions that a pattern requires. These will be its essential components. A role-element uses the same symbol that the design element that is represented.

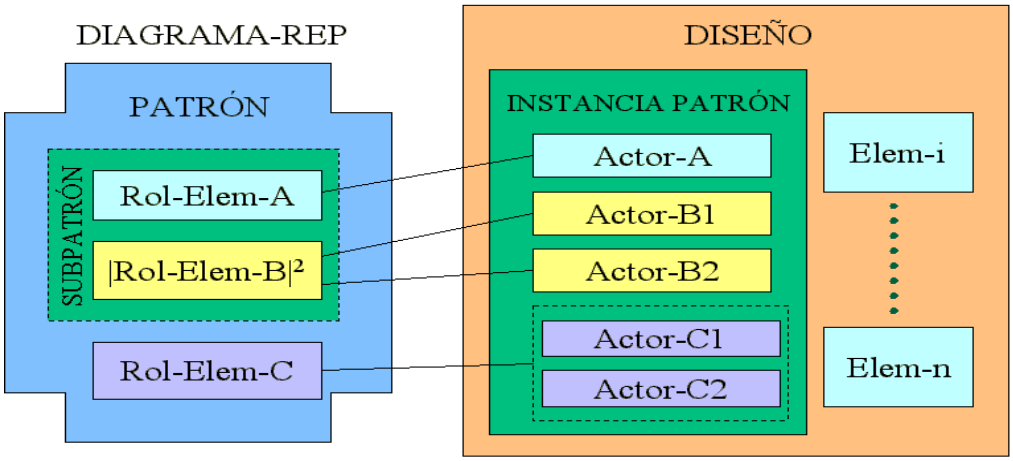


Figure 1. Basic concepts

A *REP diagram* (**R**ole-**E**lements of a **P**attern) is made up of the group of role-elements that define a certain pattern. The structure, behaviour and/or relationships that each role-element presents in this diagram represent the characteristics that the design elements of their instances should possess. Also, to complete the invariant of a pattern, other kinds of restrictions that should be completed by groups of participants can be specified (number of elements, organization of these, etc.). These restrictions can also be expressed in the REP diagram using a special notation.

A mechanism that connects design elements in a certain context with role-elements of a REP diagram could check the execution of the restrictions demanded for each element and could check if a group of design elements is an instance of the pattern.

When a design element completes all the restrictions that demands a certain role-element it can be an *actor* of this. In some ways one could think that it is assuming or playing the role of that role-element.

As has been said previously, the symbol associated to each role-element is the same as that of the actor or actors that will play it, however it is necessary to differentiate its semantics. The symbols of the actors represent elements that are part of the implementation and, consequently, their instances belong to the system in execution, however, the role-element represents design elements and it is their actors that can be instanciated.

Every role-element has an identity, that is to say, it should possess a unique identifier inside the REP diagram.

Two role-elements are different when the restrictions that they represent are different and must have different identifiers.

Two identifiers will be different if they show different identifiers or different format (italic, underlined, etc.).

Each type of design element will be characterized by some specific properties, and each type of role-element (role-class, role-attribute, role-relationship, role-operation, role-parameter, etc.) will define the restrictions on its possible actor. For example, a role-class will

define the role-relationships that it has with the other role-classes, it will indicate if its actor will be abstract, the role-attributes, the role-operations, etc.

A role-element can be:

- Obligatory. The role-element should have some actor.
- Optional. The role-element can have some actor.

A role-element can possess a *cardinal* that will indicate the number of actors that should play it.

The cardinal can be:

- Fixed. A concrete value will appear.
- Variable. Some variable will appear. The value of the cardinal is not known a priori. It can be any value or it can depend on the value of another cardinal variable if they share some of their variables. For example, if the cardinal of two role-elements is j the values will always coincide, if the cardinal is j in one place and 2*j in another the cardinal of the latter will be double the previous one.

If the cardinal is not specified it is fixed and its value is one.

A *bond* is the assignment of an actor to a role-element.

Often the instance of a pattern can present design elements that do not coincide exactly with some of the role-elements specified in the REP diagram of the pattern, disguising its ideal form. Sometimes this is because the restrictions of a role-element are satisfied by a group of elements. For example a role-relationship could be carried out by several relationships which transitively satisfy the restrictions imposed by this, the responsibilities of a class could be distributed among several classes, the behaviour required by a role-operation can be distributed among several methods, etc. Therefore, we denominate *simple bond* to that which is established between a role-element and a single actor that is forced to satisfy all the demanded restrictions and *shared bond* when it happens between a role-element and several actors that satisfy jointly all the required restrictions.

The following bond rules are applied:

- An actor can be linked to different role-elements.
- An actor cannot be linked twice to the same role-element.
- Several actors can be linked with the same role-element if their cardinal is greater than the unit or the bond is shared.
- A role-element can appear several times inside a REP diagram. In this case, the bond of an actor with one of its occurrences extends to the rest.
- The actors that participate in bonds with role-elements belonging to another role-element should be part of the actor bound to that other role-element (container).

A *bond* is *valid* when it completes the bond rules and the actor (simple bond) or actors (shared bond) complete the necessary requirements to be able to play the designated role-element. Failing this it will be *invalid*.

A *bond* is *complete* when at least all its obligatory role-elements are part of some valid bond and its cardinals are satisfied. Otherwise it will be a *partial bond*.

An *instance-pattern* is a diagram formed by the modeling elements that are actors of the role-elements of a certain REP diagram after a complete bond.

A *subpattern* is a defined subset of role-elements that belong to a REP diagram.

An *instance-subpattern* is a diagram formed by the modeling elements that are actors of a certain subpattern after a complete bond.

A subpattern can have a fixed or variable cardinal.

When a subpattern with a certain cardinal contains other cardinals belonging to subpatterns or internal role-elements we say that the cardinals are nested.

We have defined a simple visual notation (based on UML) that easily expresses all the notions mentioned previously.

As an example, we present the Abstract Factory pattern specified according to the notation used in Gamma et al. [5] and the associated REP diagram according to this new approach.

5 Application example. The Abstract Factory pattern.

This pattern (figure 2) provides an interface to create families of related objects without specifying their concrete classes.

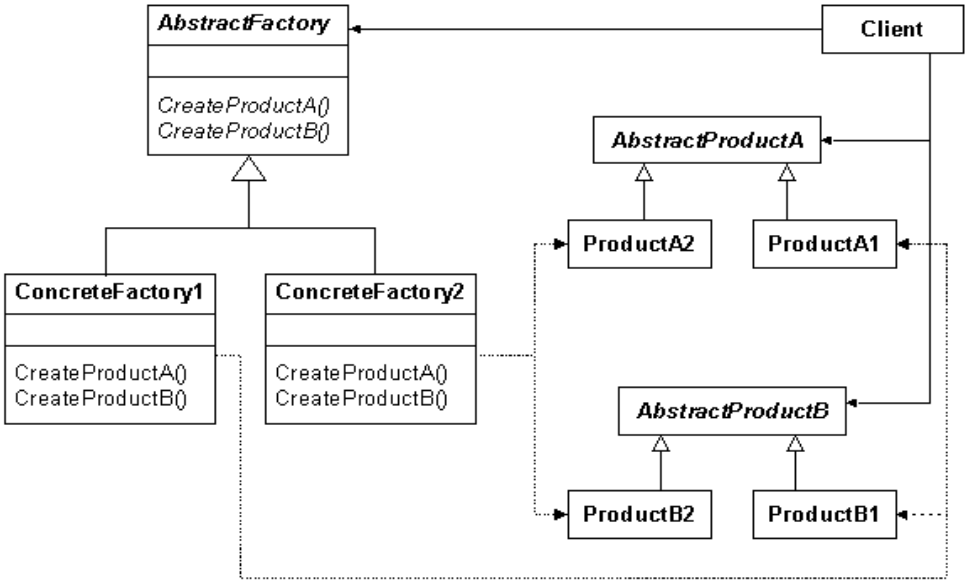


Figure 2. Abstract Factory pattern [5].

This specification is not general because it is based on a concrete example. In this case, one can observe that a variable number of classes or methods are specified by means of a fixed number of these, in this case two. This notation doesn't allow us to express the generality of a pattern without ambiguities.

In figure 3, we can observe that the REP diagram associated to this pattern is more general and simpler. The role-elements indicate the type and the structural restrictions which the actors must fulfill in order to be linked.

A restriction has been applied on the number of actors that will be able to play some of the role-elements using a variable cardinal. In this case, it is indicated that the role-element "*CreateProduct()*" abstract, "CreateProduct()" concrete and the subpattern can have a variable number of actors, but it should be the same one for all because the variable "i" is shared. This situation is also presented in "ConcreteFactory" and "Product" where the variable "j" is shared.

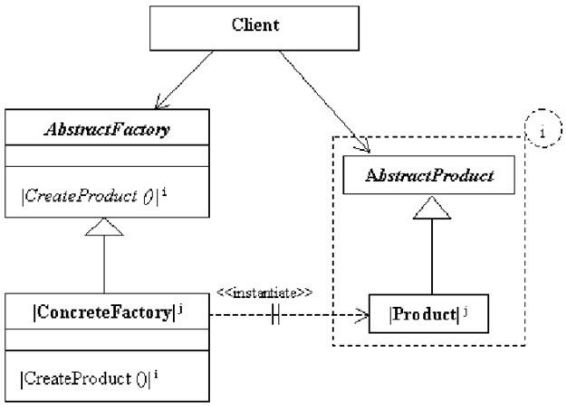


Figure 3. REP Diagram corresponding to the Abstract Factory pattern.

It is also possible to observe a restriction on the role-relationship that exists between "ConcreteFactory" and "Product". This restriction is represented by two parallel lines that cut the line of the role-relationship perpendicularly and expresses that this relationship is biyective, that is to say, the relationships between the groups of actors of the role-element origin and destination should form an biyective application.

Figure 4 presents an instance of the pattern. We can check the restrictions that the pattern imposes. By means of a color that identifies the pattern the role-element is visualized. If the element participates in different patterns, these can be specified sequentially. This characteristic could be visualized or hidden as suits by means of a tool.

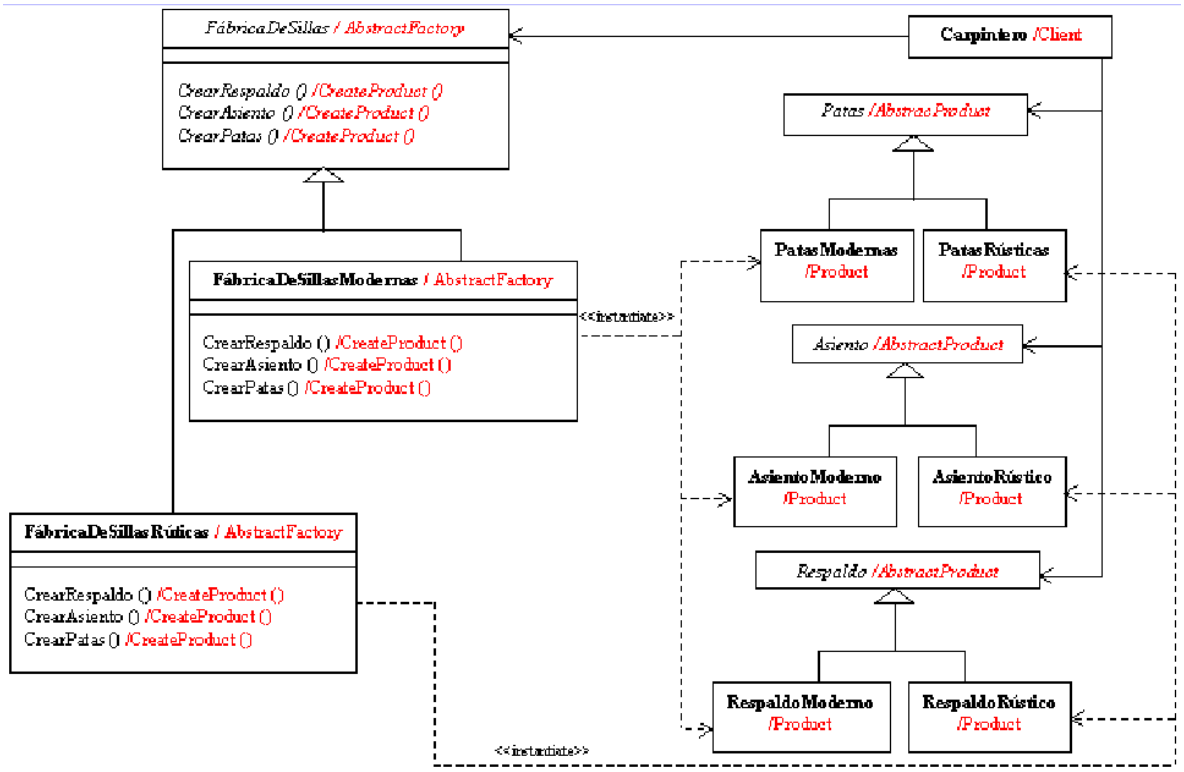


Figure 4. An instance of the REP diagram corresponding to the Abstract Factory pattern.

5 Conclusions and future work

To carry out the precise specification of a design pattern is complicated because it is not easy to give answers to the following key questions:

- What aspects should be specified?, that is to say, what should be part of the essence of a pattern?. Must not be forgotten that a pattern describes a problem, a solution and the consequences of its use. Being aware that the intrinsic complexity of a pattern is great, it seems to be that the existing works, including the present work, they are centered mainly on the more or less precise specification of that solution. At the moment, in this work we have centered on the structural aspects of this solution, however, is left pending for future works the specification of the behaviour properties.
- How to carry out this specification?. The answer necessarily depends on the answer given to the previous question. The pattern here presented carries out a precise and visual specification of the structure of a pattern through a simple extension to UML, without necessity of appealing to a formal notation as OCL. The result is a familiar notation to the designer, simple, complete and easy to learn, and an intuitive model that treats the design patterns as if they were true templates.

At the moment we have been able only to specify the structural restrictions in a visual and precise way, by means of REP diagrams. Regarding the specification of the behaviour properties, we think that it is necessary to investigate how to adapt the interaction diagrams of UML to the abstraction level of a pattern, how an extension of OCL with temporal logic could help, etc.

We think that the characteristics of the specification model that we have presented can be the base for the development of a future tool of great interest for the designers.

6 References

[1] Eden, A. H. (2000). “Precise Specification of Design Patterns and Tool Support in their Application”, PhD diss., Department of Computer Science, Tel Aviv University.

[2] Eden, A. H. (2002). “LePUS: A Visual Formalism for Object-Oriented Architectures”. The 6th World Conference on Integrated Design and Process Technology. Pasadena, California, June 26-30, 2002.

[3] Eden, A. H., Gil, J. and Yehudai, A. (1997). “Precise Specification and Automatic Application of Design Patterns”. In The 12th IEEE International Automated Software Engineering Conference – ASE 1997.http://www.math.tau.ac.il/~eden/bibliography.html#ase

[4] Eden, A. H., Hirshfeld, Y. and Yehudai, A. (1998) “LePUS – A Declarative Pattern Specification Language”. Technical report 326/98, department of computer science, Tel Aviv University. 1998. http://www.math.tau.ac.il/~eden/bibliography.html#lepus

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995). Design Patterns: Elements Reusable of Object-Oriented Software. Addison Wesley Professional Computing Series, Reading, MA.

[6] Isla, J.L. (2002). “A new approach for modelling design patterns with UML”. The 12th PHDOOS workshop (ECOOP2002), Málaga, 10-14 june 2002, Spain. http://www.softlab.ece.ntua.gr/ facilities/public/AD/phdoos02/jose.ps

[7] Kent, S. (1997). “Constraint Diagrams: Visualising Invariants in Object-Oriented Models”. In Proceedings of OOPSLA’97. ACM Press.

[8] Lauder, A. and Kent, S. (1998). “Precise Visual Specification of Design Patterns”. In Proceedings of ECOOP’98. Springer-Verlag.

[9] Le Guennec, A., Sunyé, G. and Jézéquel, J. (2000). “Precise Modeling of Design Patterns”. In Proc. UML’ 2000. Springer Verlag, LNCS 1939.

[10] Meijers, M. (1996). Tool Support for Object-Oriented Design Patterns. Master’s Thesis, INF-SCR-96-28. Utrecht University.

[11] Object Management Group. (2002). OMG Unified Modeling Language Specification V 1.4., http://www.omg.org

[12] Riehle, D. (1996). “Describing and Composing Patterns Using Role Diagrams.” WOON ’96 (1st International Conference on Object-Orientation in Russia), Conference Proceedings. St. Petersburg Electrotechnical University, 1996. Reprinted in K.-U. Mätzel and Hans-Peter Frei (eds.), Proceedings of The Ubilab Conference ’96, Zürich. Germany, Universitätsverlag Konstanz, pp. 137-152.

[13] Sunyé, G., Le Guennec, A. and Jézéquel, J. (2000). “Design Patterns Application in UML”. In Proc. European Conference in Object Oriented Programming – ECOOP’ 2000. Springer Verlag, LNCS 1850.

Special Session on Writing Patterns

WP wants to provide newcomers an opportunity to write their first pattern. There is no better way to learn what patterns are all about than writing one yourself. After having done the tutorial about "Software Patterns", in the first day of the Conference, the WP sessions aim at guiding newcomers to get started with their patterns.

The WP Session Dynamics

WP sessions were about 1 hour each. All participants were invited to read the papers in advance, so that they could contribute, during the session, with constructive criticism about the pattern. The patterns discussed in these sessions are beginning to emerge, so authors have heard about the format, the adequacy of notations, what to include in the several pattern elements, etc.

The session dynamics was similar to the Writers' Workshop, but the author could interact with questions and clarifications during the session, rather than only in the last 10 minutes.

Papers of this session are candidates to be submitted to future PLoP's, where they will be shepherd and workshoped.

Observer Pattern using Aspect-Oriented Programming

Eduardo Kessler Piveta* and Luiz Carlos Zancanella
Departamento de Informática e Estatística
Universidade Federal de Santa Catarina
88040-900 Florianópolis, SC
{kessler,zancanella}@inf.ufsc.br

Abstract

This paper discusses the representation and implementation of the Observer design pattern using aspect-oriented techniques.

1 Introduction

Several object-oriented design techniques have been used to specify and implement design patterns efficiently. However there are several patterns that affect system modularity, where base objects are highly affected by the structures that the pattern requires. In other cases, we want to apply a pattern to classes that already belong to a hierarchy. This could be hard depending on the pattern.

Several patterns crosscut the basic structure of classes adding behavior and modifying roles in the classes relationship. As an example you could see the Observer pattern. When you have to implement the pattern, you should provide implementation to the Subject and Observer roles. The implementation of this roles adds a set of fields (`Subject.observers`, `Observer.subject`) and methods (`Attach`, `Detach`, `Notify`) to the concrete classes or to its superclasses, modifying the original structure of the subject and observer elements.

Another example is the Visitor pattern. Its main goal is to provide a set of operations to a class hierarchy without changing the structure of the underlying classes. In order to accomplish that task, the pattern adds to the `Element` class a method (`Accept`) to allow the `Element` instances to be visited.

Althougth the use of these patterns brings several benefits, they could "hard-code" the underlying system, making difficult to express changes. To implement one of the patterns described above in an envolving system you may have to modify several classes, affecting their relationships and their clients.

Aspect Oriented Programming [12] can help on separating some of the system's design patterns, specifying and implementing them as single units of abstraction.

*Professor at Centro Universitário Luterano de Palmas - TO, Brazil

The main goal is to show how the Observer [8] pattern could be implemented using aspect-oriented programming and how it could be specified using an aspect-oriented design model.

The intended audience of this paper is composed by object-oriented developers that do not have experience on aspect-oriented programming and by aspect-oriented developers aiming for implementations of design patterns based on crosscutting mechanisms.

In the following section, the core concepts of aspect-oriented programming (AOP) are discussed. Readers that are familiarized with AOP could go directly to the section that describes the pattern.

2 Aspect Oriented Programming

Software engineering and programming languages exist in a mutual relationship support. The most used design processes break a system down into a set of small units. To implement these units, programming languages provide mechanisms to define abstractions and composition mechanisms in order to implement the desired behavior [2].

A programming language coordinates well with a software design when the provided abstraction and composition mechanisms enable the developer to clearly express the design units. The most used abstraction mechanisms of languages (such as procedures, functions, objects, classes) are derived from the system functional decomposition and could be grouped into a generalized procedure model [12].

However, there are many properties that do not fit well into generalized procedures, such as: exception handling, real-time constraints, distribution and concurrency control. They are usually spread over into several system modules, affecting performance and/or semantics systematically.

When these properties are implemented using an object-oriented or a procedural language, their code is often tangled with the basic system functionality. It is hard to separate one concern from another, see or analyze them as single units of abstraction.

This code tangling is responsible by part of the complexity found in computer systems today. It increases the dependencies among the functional modules, deviating them from their original purposes, making them less reusable and error-prone.

This separation of concerns is a fundamental issue in software engineering and it is used in analysis, design and implementation of computer systems. However, the most used programming techniques do not always present themselves in a satisfactory way regarding to this separation.

Aspect-oriented programming allows separation of these crosscutting concerns, in a natural and clean way, using abstraction and composition mechanisms to produce executable code.

The aspect-oriented programming main goal is to help the developer in the task of clearly separate crosscutting concerns, using mechanisms to abstract and compose them to produce the desired system. The aspect-oriented programming extends other programming techniques (object oriented, structured, functional etc) that do not offer suitable abstractions to deal with crosscutting [12].

An implementation based on the aspect oriented programming paradigm is usually composed of:

- a component language to program components (i.e. classes);

- one or more aspect languages to program aspects;
- an aspect weaver to compose the programs written in these languages;
- programs written in the components language;
- one ore more programs written in the aspect language.

2.1 Components

Components (in AOP) are abstractions provided by a language to implement systems basic functionality. Procedures, function, classes and objects are components in aspect-oriented programming. They are originated from functional decomposition. The language used to express componnents could be, for instance, an object-oriented, an imperative or a functional one [16].

2.2 Aspects

Properties affecting several classes could not be well expressed using current notations and languages. They are usually expressed through code fragments that spread over the system classes [5].

Some concerns that are frequently aspects: concurrent objects synchronization [6], distribution [13], exception handling [15], coordination of multiple objects [10], persistence, serialization, replication, security, visualization, logging, tracing, load balance, fault tolerance amongst others.

2.3 Component language

The component language should provide developers with mechanisms to write programs implementing the basic requirements and also do not predict what is implemented in the aspects, (this property is called obliviousness [7]). Aspect-oriented programming is not limited to object orientation, although, the most used component languages are object oriented ones, such as: *Java*, *SmallTalk* or *C#*.

2.4 Aspect language

The aspect language defines mechanisms to implement crosscutting in a clear way, providing constructions describe the aspects semantics and behavior [12].

Some guidelines should be observed in the specification of an AO language: syntax should be related to the component language syntax (making easier the tools acceptance), the language should be designed to specify the aspect in a concise and compact way and the grammar should have elements to allow composition of classes and aspects [4].

2.5 Aspect Weaver

The aspect weaver main responsibility is to process aspect and component languages, in order to produce the desired operation. To do that, it is essential the *join-point* concept. A join-point is a well defined point in the execution or structure of a program. For instance,

in object-oriented programs join-points could be method calling, constructor calling, field read/write operations etc.

The representation of those points could be generated in runtime using a reflective environment. In this case, the aspect language is implemented through meta-objects, activated at method invocations, using join-points and aspects information to weave the arguments [14].

An aspect-oriented system design requires knowledge about what should be in classes and in aspects, as well as characteristics shared in both. Although aspect-oriented and object-oriented languages have different abstraction and composition mechanisms, they should use some common terms, allowing the weaver to compose the different programs.

The weaver parses aspect programs and collects information about the (join) points referenced by the program. Afterwards, it locates coordination points between the languages, weaving the code to implement what is specified in them [3].

An example of a weaver implementation is a pre-processor that traverse the classes parsing tree, looking for joint-points and inserting sentences declared in the aspects. This weaving process could be static (compile time) or dynamic (load and runtime).

3 Observer Pattern

3.1 Intent

It allows the definition of a "one to many" relationship between a model (Subject) and its dependents (Observers) in a way that promotes low coupling. Using aspect-oriented programming you could also attach and dettach the design pattern in compile-time or runtime (depending upon the choosen language)

3.2 Motivation

The problem that the Observer pattern solves is how to maintain consistency among several objects that depends on a model data in a way that promotes reuse, keeping a low coupling among classes. In this pattern, every time the Subject state changes, all the Observers are notified.

The main problem with the object-oriented Observer pattern is that you should modify the structure of classes that participate in the pattern. So, it is hard to apply the pattern into an existing design as well as remove from it.

3.3 Context

The Observer pattern is used in the following situations, according to [8]:

- When a change in the state of an object demands modification in unknown or variable objects
- When an object needs to notify others without knowing whose are the objects that are going to receive this notification.

3.4 Structure

The design of the Observer pattern is changed in order to represent it as classes and aspects. It could be seen in the Figure 1, represented as a class diagram. There are no `Observer` role neither a `Subject` one. Both structure and behavior of these two roles are expressed in the `ObserverPattern` aspect.

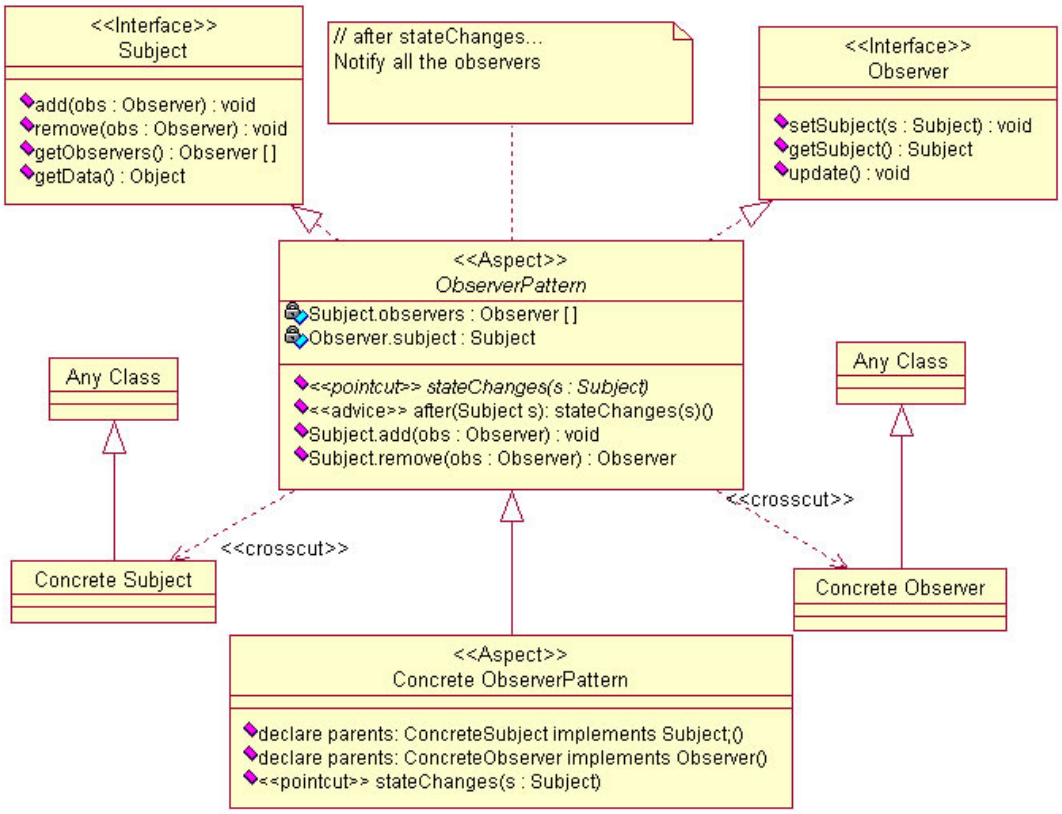


Figure 1: Observer's class diagram

3.5 Participants

Subject Describes the interface that all the concrete subjects must be in accordance to (enforced by the `ObserverPattern` and `ConcreteObserverPattern`). When implemented, the subject will contain a reference to its observers, and allow the dynamic addition and deletion of observers.

Observer Describes the interface that all the concrete observers must be in accordance to (enforced by the `ObserverPattern` and `ConcreteObserverPattern`). They are notified everytime the state of the subject changes.

ConcreteSubject Store state information to be used by ConcreteObservers. It does not, however, send notifications to its Observers. This responsibility is part of the ObserverPattern role. [8]

ConcreteObservers Serve as basis to field and method's introduction performed by the ObserverPattern.

ObserverPattern The ObserverPattern is an abstract aspect that encapsulte the behaviour of the Observer pattern. The `ObserverPattern` contains the fields and methods to be included in the classes that are affected by the `ConcreteObserverPattern`.

Concrete ObserverPattern This participant specifies in what situations the `ConcreteObservers` are going to be notified as well as what is going to be executed when the `ConcreteObservers` are notified.

3.6 Dynamics

When the classes corresponding to concrete subjects and concrete observers are weaved together with the concrete ObserverPattern, the following methods and fields are attached to the concrete subject and concrete observer:

fields `Observer[] observers` (Concrete Subject), `Subject subject` (Concrete Observers)

methods `void add(Observer obs)`, `void remove(Observer obs)` (Concrete Subject), `void setSubject(Subject s)`, `Subject getSubject()` (Concrete Observers)

These attachments are specified in the abstract aspect (the ObserverPattern). The concrete subjects implements the subject interface. The concrete observers implement the observer interface. The other modifications are done to the dynamic nature of the observer and subject classes, telling that every time that the state of the subject changes the update method of the observers is invocated.

3.7 Consequences

- The use of the Observer pattern allows to reuse subjects and observers in an independent way, since you can add new observers without change the subject or the others existing observers.
- Using an aspect-oriented implementation, the ObserverPattern could be reused without further modifications, as well as the classes affected by the pattern (that could be reused without considerations about the pattern).
- The use of aspect-oriented programming helped to separate the code related to the design pattern from the code of the base program itself.
- The use of an abstract aspect leads to a better reuse of the design pattern since it can be applied to several cases without changes. The developer should develop only the concrete aspect (as in this paper) to apply the pattern to the base code.

- Other consequences of the Observer pattern, as stated in [8] are: abstract coupling among subjects and observers and support to a broadcast communication. One disadvantage on using this pattern is that the subject does not know how much costs an updating in all its observers.

- Another advantage is that the classes do not need to extend the subject and observer classes in an explicit way. The user attaches the fields and methods needed to implement the pattern.

3.8 Implementation

Some considerations could be made while implementing the Observer pattern:

- In order to send notification to its observers, the subject usually declares an explicit vector containing all its Observers. You could implement it as other data structure (such as: hash tables, linked lists etc) to solve performance or memory problems.

- `ObserverPattern` and `ObserverPatternImpl` could be mixed into one class. The reuse of the pattern in this case is limited.

- Other implementations issues can be found in [8]

3.9 Example with AspecJ

In this section we are going to show an example on using `AspectJ` [11] to implement the Observer design pattern using a temperature domain. Suppose a set of thermometers that gather information from a temperature source. Each time that the source temperature changes the thermometers display should be updated.

An `Observer` interface is defined in order to describe the Observer role. All Observers must be in accordance to this interface. The idea here is to allow each Observer to have a corresponding Subject. This example does not treat multiple Subjects to one Observer.

```
1  interface Observer {
2       void setSubject(Subject s);
3       Subject getSubject();
4       void update();
5  }
```

Figure 2: Observer Interface

In a similar way, we have a `Subject` interface (Figure 3), that all subjects must be in accordance to.

In Figure 4 we have a class called `Celsius` which is a source of temperature data. In this class is stored information about current temperature in Celsius. This information can be modified or retrieved using the `setDegrees` and `getDegrees` methods, respectively. This class is going to perform the `Subject` role. Note that there are not references to the `Subject` class or interface. Each time that the `setDegrees` method is invocated, the subjects are notified about changes in the temperature source state (this is going to be implemented as aspects).

```
1  import java.util.Vector;
2  interface Subject {
3       void add(Observer obs);
4       void remove(Observer obs);
5       Vector getObservers();
6       Object getData();
7  }
```

Figure 3: Subject Interface

```
1  public class Celsius{
2       private double degrees;
3       public double getDegrees(){
4            return degrees;
5       }
6       public void setDegrees(double aDegrees){
7            degrees = aDegrees;
8       }
9       Celsius(double aDegrees){
10           setDegrees(aDegrees);
11      }
12 }
```

Figure 4: Celsius class - The Subject

Another important class is the class that represents a thermometer (Figura 5). This class has a field called `tempSource` that points to a `Celsius` instance. This class is the base classe for the thermometers in the examples and is going to perform the observer role.

```
1  public class Thermometer{
2       private Celsius tempSource;
3       public void setTempSource(Celsius atempSource){
4            tempSource = atempSource;
5       }
6       public Celsius getTempSource(){
7            return tempSource;
8       }
9       public void drawTemperature(){}
10 }
```

Figure 5: Thermometer class - The Observers superclass

Extending the thermometer class we have a Celsius (Figure 6) and a Fahrenheit thermometer (Figure 7). These classes override the `drawTemperature()` method providing different scales of temperature.

In this example the `Celsius` class has the subject role and the thermometers classes are the observers. The reader could have already noted that `Celsius` class neither thermometers classes have explicit connection with the observer and subject classes.

The use of AspectJ allows the developer to separate the code related to the Observer pattern from the classes that use it.

In Figure 8 we have an abstract aspect called ObserverPattern that implements the basic functionality of the design pattern. This abstract aspect was retrieved from the AspectJ 1.1 examples and small modifications were made in the original structure.

```
1  public class CelsiusThermometer extends Thermometer{
2       public void drawTemperature(){
3         System.out.println("Temperature in Celsius:"+
4              getTempSource().getDegrees());
5       }
6  }
```

Figure 6: The CelsiusThermometer class

```
1  public class FahrenheitThermometer extends Thermometer{
2    public void drawTemperature(){
3         System.out.println("Temperature in Fahrenheit:"+
4         (1.8 * getTempSource().getDegrees())+32);
5    }
6  }
```

Figure 7: The FahrenheitThermometer class

This abstract aspect creates an abstract pointcut called `stateChanges` that will be extended in the concrete aspect to tell in what situations the observers are going to be notified. It implements an after advice that notifies all the observers when these points (situations) are reached.

It also create some fields (`Subject.observers` and `Observer.subject`) and methods (`Subject.add(..)`, `Subject.remove(..)`, `Subject.getObservers()`, `Observer.setSubject(..)`, `Observer.getSubject()`) in the classes that implements the `Subject` and `Observer` interfaces as implemented in Figure 8;

```
1   import java.util.Vector;
2   abstract aspect ObserverPattern {
3        abstract pointcut stateChanges(Subject s);
4        after(Subject s): stateChanges(s) {
5             for (int i = 0; i < s.getObservers().size(); i++)
6               ((Observer)s.getObservers().elementAt(i)).update();
7        }
8        private Vector Subject.observers = new Vector();
9        public void Subject.add(Observer obs) {
10            observers.addElement(obs);
11            obs.setSubject(this);
12       }
13       public void Subject.remove(Observer obs) {
14            observers.removeElement(obs);
15            obs.setSubject(null);
16       }
17       public Vector Subject.getObservers() { return observers; }
18       private Subject Observer.subject = null;
19       public void Observer.setSubject(Subject s) { subject = s; }
20       public Subject Observer.getSubject() { return subject; }
21  }
```

Figure 8: Observer/Subject Protocol from AspectJ 1.1 examples

Here we extend the abstract aspect in order to tell to the compiler which classes are going to be treated as observers, as subjects and in which cases observers will be notified (the methods that are going to be executed when the notification happens are specified too). The sentences in `lines 3 and 5` bellow tells the compiler that the Cel-

sius class implements the `Subject` interface and the `Thermometer` class implements the `Observer` interface. In `line 4` the method `getData()` is implemented in accordance with the `Subject` interface and in `line 6` the `update()` method is defined in order to accomplish the `Observer` interface.

The abstract pointcut defined in the `ObserverPattern` aspect is extended here defining that the observers are going to be notified every time the `setDegrees()` method is called.

```
1  import java.util.Vector;
2  aspect ObserverPatternImpl extends ObserverPattern {
3       declare parents: Celsius implements Subject;
4       public Object Celsius.getData() { return this; }
5       declare parents: Thermometer implements Observer;
6       public void Thermometer.update() {
7            drawTemperature();
8       }
9       pointcut stateChanges(Subject s): target(s) &&
10           call(void Celsius.setDegrees(..));
11 }
```

Figure 9: Concrete Observer protocol

3.10 Example using only Java

Consider an implementation of the Observer Pattern using Java, where we have an *Termometer* class playing the *Observer* role (Figure 10) and a *Clecius* class playing the *Subject* Role (Figure 11) .

```
1  public abstract class Termometer{
2       private Subject subject = null;
3       private Celcius tempSource;
4       // getter and setter methods
5       public abstract void drawTemperature();
6       public void update() {
7            drawTemperature();
8        }
9  }
```

Figure 10: Termometer Class

Note that there is an explicit reference to the subject.

In this implementation of the pattern, all the methods to add, remove and get observers are directly defined in the class playing the subject role. The concrete observers are not affected by this solution neither by the aspect-oriented one.

The diference between implementations is that in the OO code, the sentences related to the Observer pattern are tangled with the program basic functionalities. Although some of this problems could be solved using wrapper methods, reflection or changing the implementation language (smalltalk for instance), the use of aspect-oriented programming could still being another alternative to implement this pattern.

```
1  import java.util.Vector;
2  public class Celcius implements Subject{
3       private double degrees;
4       private Vector observers = new Vector();
5       public Object getData() { return this; }
6       public double getDegrees(){
7            return degrees;
8       }
9       public void setDegrees(double aDegrees){
10           degrees = aDegrees;
11            for (int i=0;i<getObservers().size();i++){
12                 ((Observer)getObservers().
13                 elementAt(i)).update();
14            }
15       }
16       public void add(Observer obs) {
17            observers.addElement(obs);
18            obs.setSubject(this);
19       }
20       public void remove(Observer obs) {
21            observers.removeElement(obs);
22            obs.setSubject(null);
23       }
24       public Vector getObservers()
25       { return observers; }
26       Celcius(double aDegrees){
27            setDegrees(aDegrees);
28       }
29  }
```

Figure 11: Celcius Class

3.11 Known Uses

Some common uses of this pattern (in the object-oriented way) could be found in [8]. All the GoF patterns was already implemented in AspectJ and discussed in [9] with different levels of success.

4 Future Work

Future work will focus on finding common patterns in aspect-oriented programming that do not appear in object-oriented ones and on defining refactorings to aspect-oriented code.

5 Acknowledgements

We would like to thanks Deise Saccol and Thereza Padilha for their comments in early versions of this paper, to Federico Balaguer to the sheperding activities and to Robert Hammer, Joe Yoder, Paulo Borba and others in the group for their suggestions during the writing patterns sessions.

References

[1] *Workshop on Aspect Oriented Programming (ECOOP 1998)*, June 1998.

[2] Christian Becker and Kurt Geihs. Quality of service - aspects of distributed programs. In *Int'l Workshop on Aspect Oriented Programming (ICSE 1998)*, April 1998.

[3] K. Böllert. Aspect-oriented programming case study: System management application. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].

[4] Kai Böllert. On weaving aspects. In *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[5] Krzystof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, Boston, 2000.

[6] John Dempsey and Vinny Cahill. Aspects of system support for distributed computing. In *Workshop on Aspect Oriented Programming (ECOOP 1997)*, June 1997.

[7] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns (OOPSLA 2000)*, October 2000.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. Design patterns: Elements of reusable object-oriented software. 1995.

[9] Jan Hannemann and Gregor Kiczales. Design pattern implementation in java and aspectj. In Mamdouh Ibrahim, editor, *Proc. 17th Annual Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA-2002)*. ACM Press, November 2002.

[10] William Harrison and Harold Ossher. Subject-oriented programming—a critique of pure objects. In *Proc. 1993 Conf. Object-Oriented Programming Systems, Languages, and Applications*, pages 411–428, September 1993.

[11] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proc. ECOOP 2001, LNCS 2072*, pages 327–353, Berlin, June 2001. Springer-Verlag.

[12] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *11th Europeen Conf. Object-Oriented Programming*, volume 1241 of *LNCS*, pages 220–242. Springer Verlag, 1997.

[13] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, 1997.

[14] Anurag Mendhekar, Gregor Kiczales, and John Lamping. RG: A case-study for aspect-oriented programming. Technical Report SPL-97-009, Palo Alto Research Center, 1997.

[15] H. Ossher and P. Tarr. Operation-level composition: A case in (join) point. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].

[16] B. Tekinerdoğan and M. Akşit. Deriving design aspects from canonical models. In *Workshop on Aspect Oriented Programming (ECOOP 1998)* [1].