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.

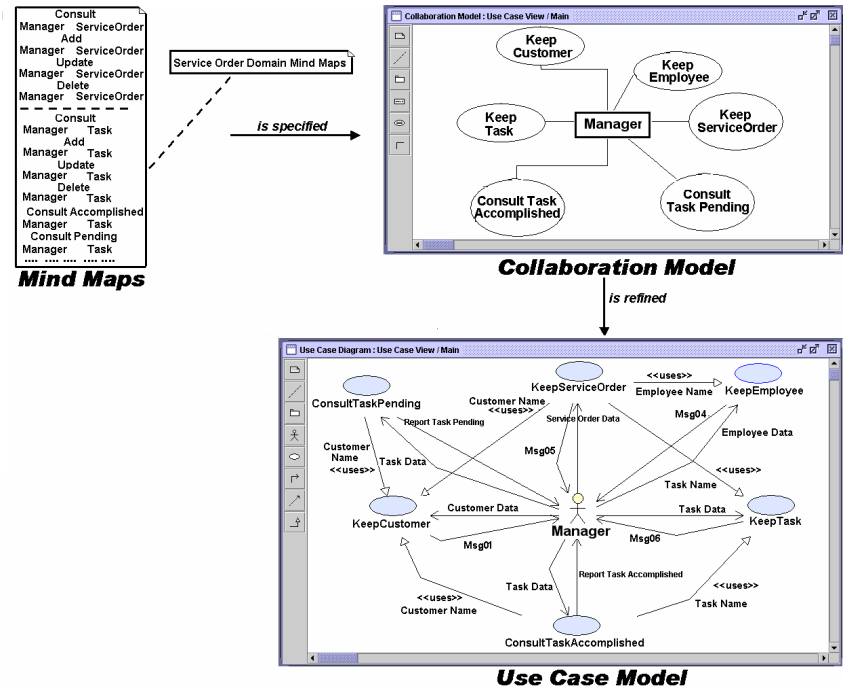

Figure 1. Models generated from Define Problem pattern.

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.

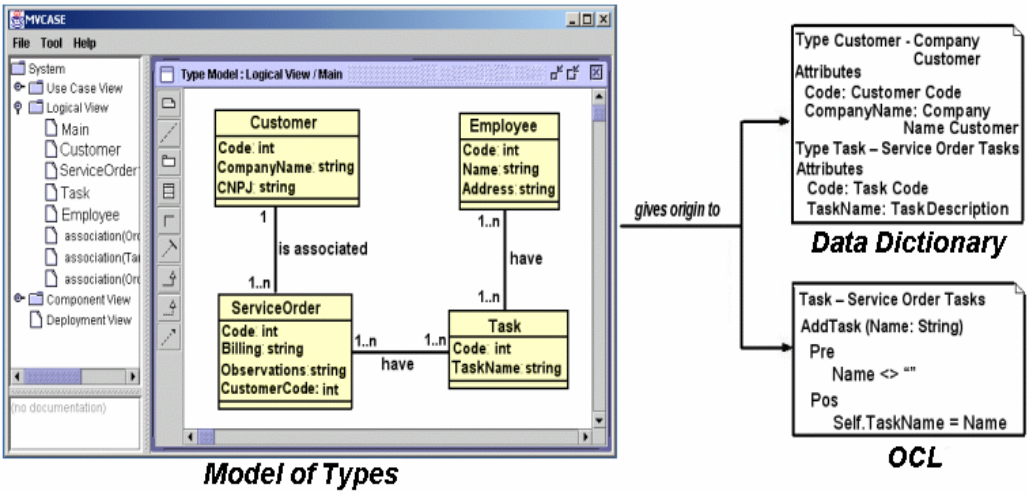

Figure 2. Model of Types from Specify Components pattern.

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.

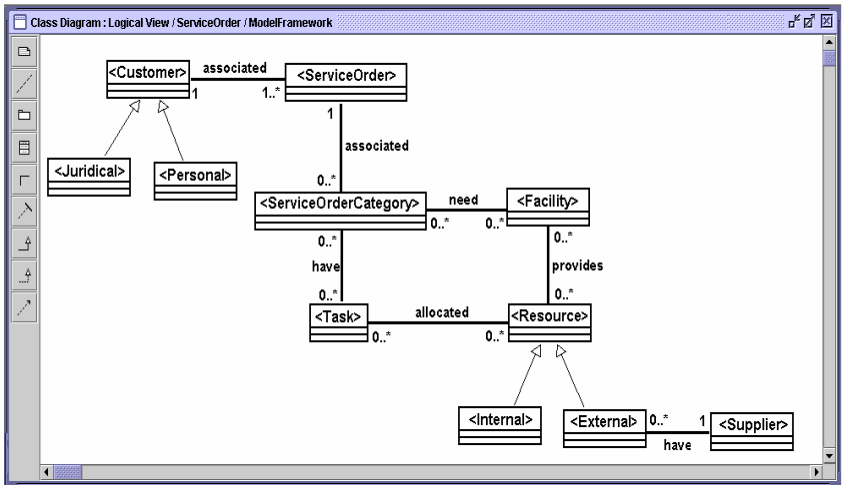

Figure 3. Service Order Model Framework.

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.

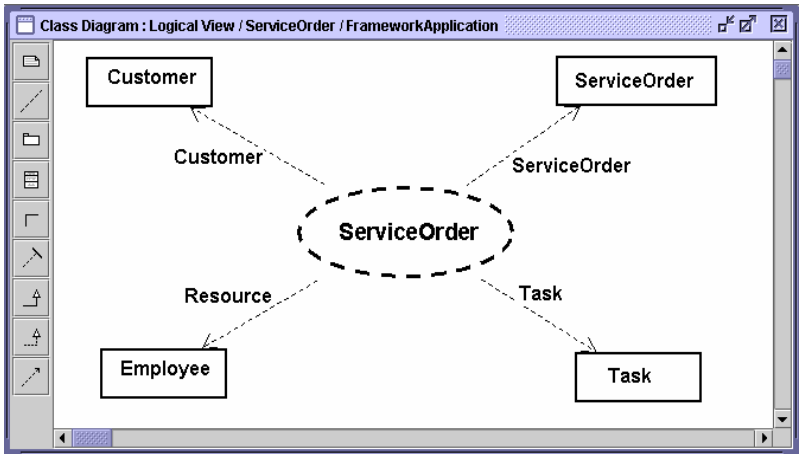

Figure 4. Service Order Framework Application.

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.

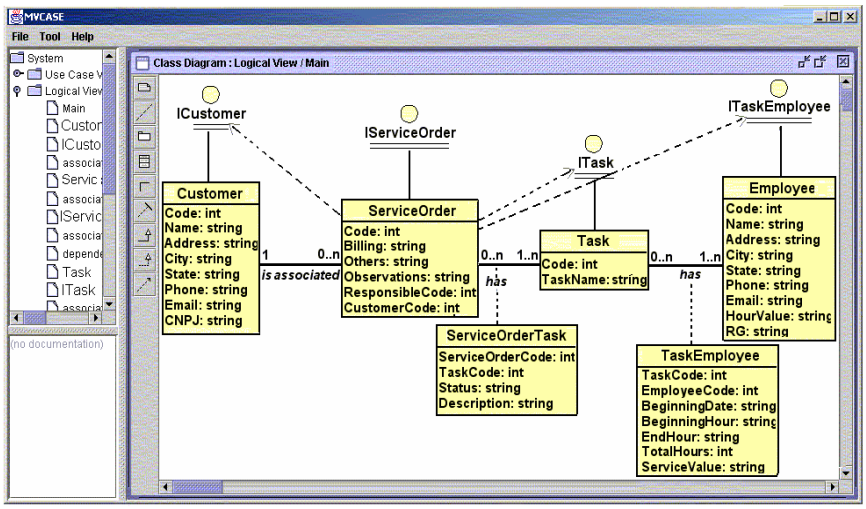

Figure 5. Classes Model obtained from Model of Types.

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.

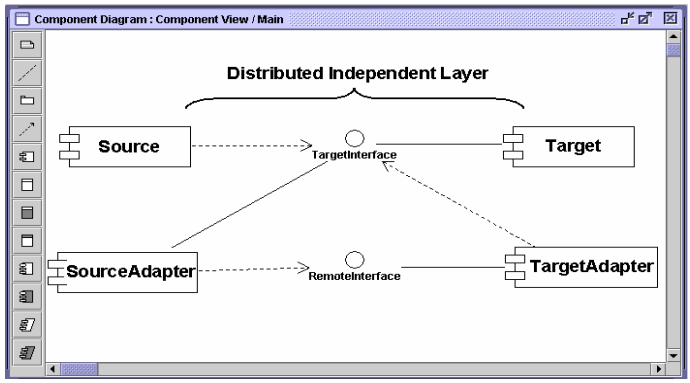

Figure 6. Design Component Model after apply DAP.

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.

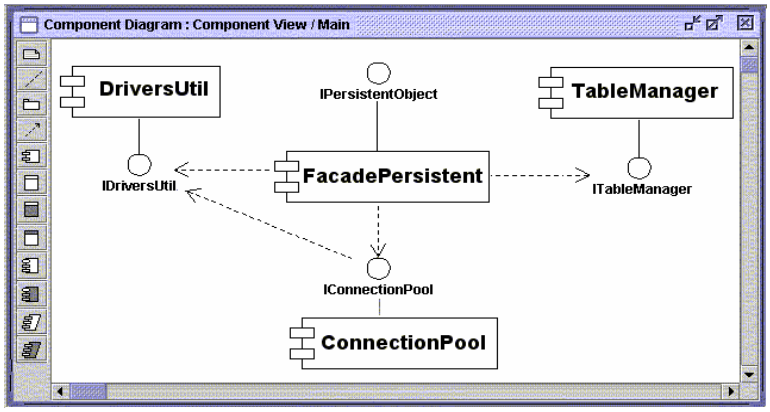

Figure 7. Framework Persistence.

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.

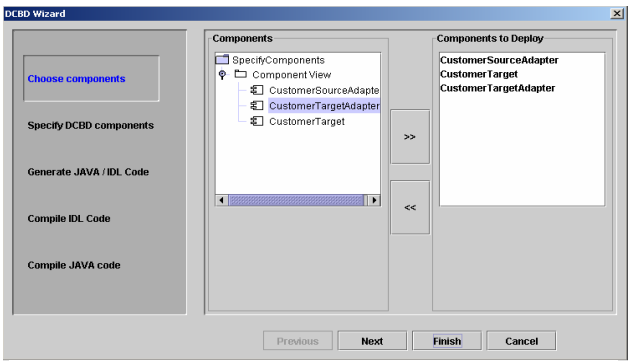

Figure 8. Generate code in MVCase tool.

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.

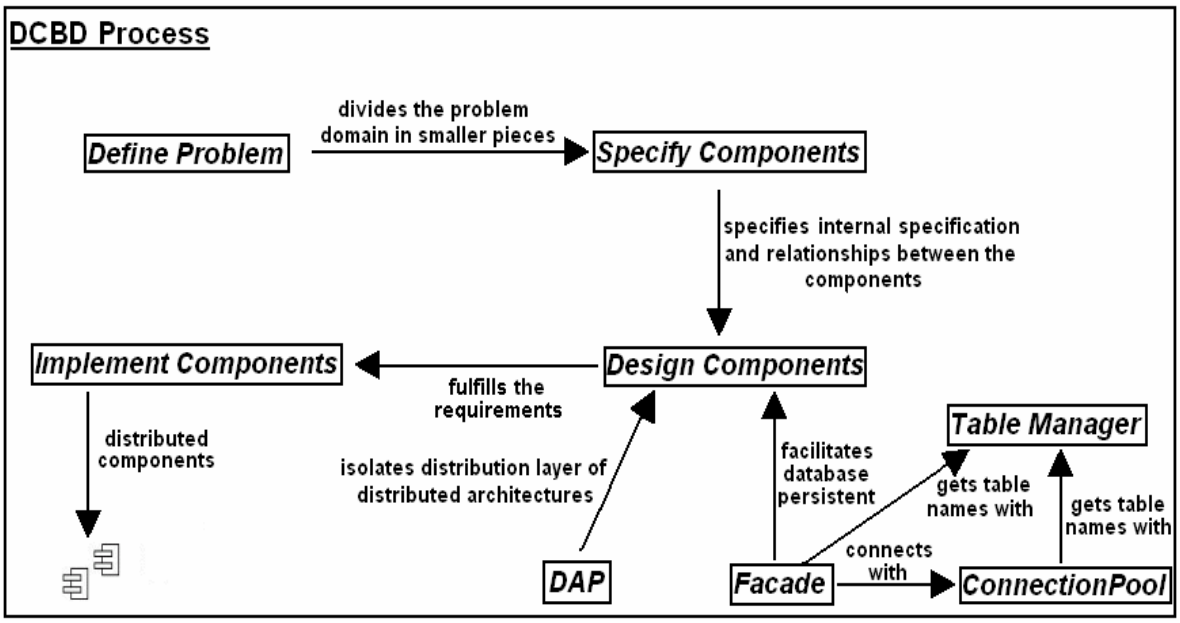

Figure 10. Pattern Interaction Diagram.

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.

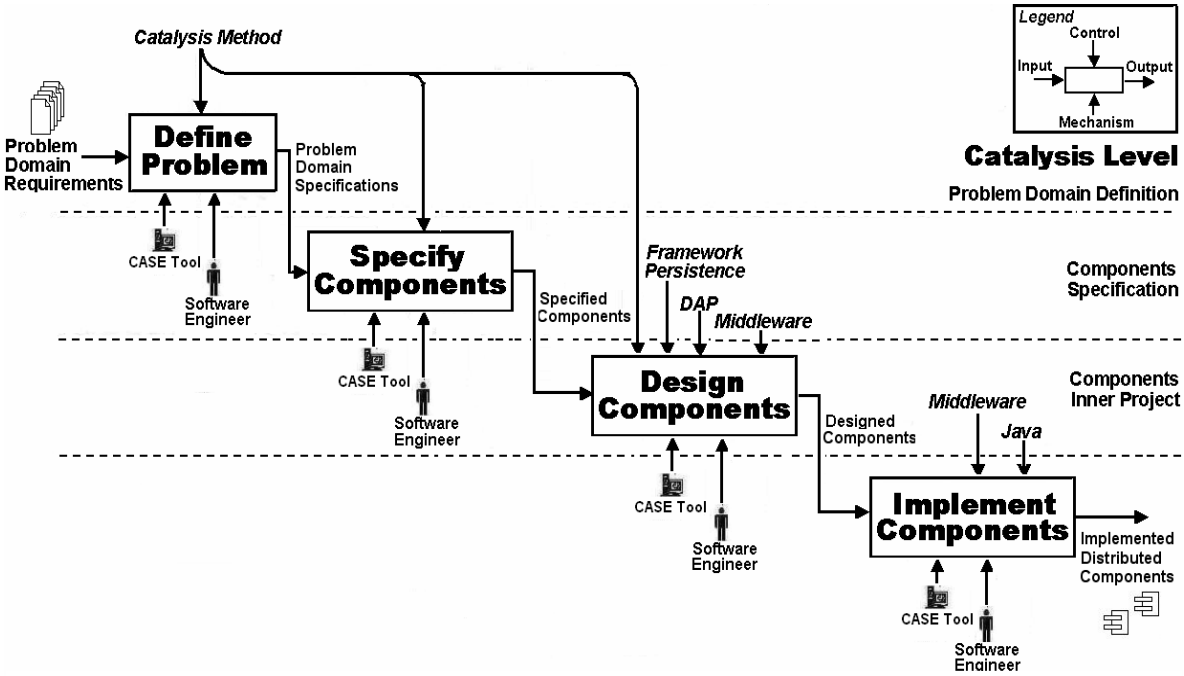

Figure 11. Distributed Components Development Process.

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.