Universidade Federal de Pernambuco
Graduação em Ciência da Computação
Centro de Informática

UNDERGRADUATE CONCLUSION PROJECT

REUSABLE COMPONENT IDENTIFICATION FROM EXISTING OBJECT-ORIENTED PROGRAMS

TOWARDS AN AUTOMATIC COMPONENT LOAD TOOL

Student: Cassio de Albuquerque Melo (cam2@cin.ufpe.br)
Advisor: Silvio Romero de Lemos Meira (srlm@cin.ufpe.br)
Co-Advisor: Eduardo Santana de Almeida (esa2@cin.ufpe.br)

Recife, January/2008

"It is not possible to be a scientist unless you believe that the knowledge of the world, and the power which this gives, is a thing which is of intrinsic value to humanity, and that you are using it to help in the spread of knowledge and are willing to take the consequences."

J. Robert Oppenheimer

Acknowledgements

First, I want to express my gratitude to my advisor, friend and guru, Dr. Eduardo Almeida, for his support to this dissertation who I express my sincere admiration. Eduardo trusted in my potential as a researcher, always entertaining my questions with interest and patience.

To Dr. Alex Sandro, who was my advisor not only in an academic project, but in life, throughout my entire undergraduation course. Alex's influence made deep contributions to this work.

I want to thank my parents, Raimundo and Ana, and my sisters for their unconditional love and affection.

To my aunt Betânia and my friend Fernando, for helping cook, cleaning, and maintaining everything in place in my house while I divided my time working on this dissertation.

Thanks to everyone who I interacted with and participated in projects, as well as all RiSE and CITi members, in special Daniel Arcoverde and Raoni Franco who I had great discussions; and Lica, Yguaratã, Kellyton and Alexandre who reviewed this work over and over giving me worthy suggestions. I've met a lot of innovative, hardworking folks and they all deserve a big kudos.

As always, there are many others who made great contributions to this work but the folks mentioned are those who I worked with day-to-day. Thanks, everyone.

All the expenses related to my under graduation course were paid by millions of Brazilians who I never met and worked hard to pay the bills and taxes. I wish to dedicate this dissertation to them.

Abstract

Software Reuse has grown in maturity and relevance over the past several years and now it is considered one of the most plausible mean to bring to the industry better levels of productivity, quality, time-to-market and hence, competitiveness.

A well succeeded Software Reuse program comprises in many different strategies, varying from technical perspective to the organizational and managerial perspective. Among the technical factors in software reuse, a reusable asset repository plays an important role in reuse programs since it stores valuable, experienced knowledge. Despite its benefits, an asset repository must be populated with reusable artifacts in order to be useful to developers; otherwise, its adoption is definitively compromised. On the other hand, already developed software is available at several open repositories on the internet and at companies' own private repository. The effort needed do identify reusable artifacts from these existing sources must be considered.

This work lies under this motivation. We are trying to answer questions such as, how can we assist engineers in the process of identifying candidates of components from existing source code? What kind of heuristics and metrics should be blend (and how) in order to get better results? How can we make it scalable to large systems?

We have analyzed current approaches on component identification, techniques and tools, mainly focused on software clustering. We specified, designed and implemented a tool to help engineers in indentifying reusable modules from java source code: the CORE Loader. The CORE Loader finds partitions according to the edge strength among the classes.

Keywords: component identification, software modularization, software partitioning, software clustering, software metrics, reverse engineering, software reuse.

Resumo

O Reuso de Software tem crescido em maturidade e relevância ao longo de vários anos e atualmente é considerado como um dos meios mais plausíveis de se trazer à indústria melhores níveis de produtividade, qualidade, *time-to-market* e conseqüentemente, de competitividade.

Um programa de Reuso de Software bem sucedido consiste em várias estratégias diferentes, variando da perspectiva técnica à perspectiva organizacional e de gestão. Entre os fatores técnicos na reutilização de software, um repositório de ativos reutilizáveis desempenha um papel importante no programa de reuso, uma vez que armazena um valioso conhecimento já experimentado. Apesar das suas vantagens, a base de dados do repositório deve ser preenchida com artefatos reutilizáveis para que seja útil para os desenvolvedores; caso contrário, a sua adoção estará definitivamente comprometida. Da mesma forma, existem centenas de softwares disponíveis em repositórios na Internet e em repositórios privados nas próprias empresas. O esforço necessário para se identificar possíveis artefatos reutilizáveis dessas fontes existentes deve ser considerado.

Esta é a motivação deste trabalho. Estamos tentando responder a questões como: de que modo podemos ajudar os engenheiros no processo de seleção dos candidatos a componente a partir do código-fonte existente? Quais tipos de heurísticas e métricas devem ser aplicados (e como), para se obter os melhores resultados?

Nós analisamos as técnicas e ferramentas atuais sobre identificação de componentes, focadas principalmente em clusterização software. Nós especificamos, projetamos e implementamos uma ferramenta para ajudar engenheiros em indentificar módulos reutilizáveis a partir de código fonte java: o CORE Loader. Nossa abordagem está concentrada em encontrar as partições de acordo com a força de relacionamento entre as classes.

Contents

List of Figures

List of Tables

1 Introduction

1.1. Context

According to Krueger [Krueger 1992], “Software reuse is the process of using existing software artifacts rather than building them from scratch”. Reusable software components include not only the generic source code, but also other aspect of the software development lifecycle including design structure, specifications, and documentation.

Software reuse started to be discussed during the NATO Software Engineering Conference in 1968 and the focus was the software crisis – the problem of building large, reliable software systems in a controlled, cost-effective way. An invited paper at the conference: "Mass Produced Software Components" by McIlroy [McIlroy 1968], ended up being the seminal paper on software reuse.

Nowadays, Software Reuse has grown in importance and has become an indispensable requirement for companies’ competitiveness. Some experiences in industry [Bauer 1993], [Griss 1995] have reinforced the idea that software reuse improves productivity and helps to obtain low costs and high quality during the whole software development cycle.

Despite its benefits, an effective application of both technical and non-technical aspects is crucial to the success of software reuse programs. The non-technical aspects include question such as education, training, incentives and organizational management [Sametinger 1997]. On the other hand, technical aspects comprise, among other things, the creation of a component repository that supports software engineers and other users in the process of developing software for and with reuse. A component repository can be seen as a base to

storage, search and recovery software components. There are some academic and commercial repository tools such as [Garcia et al 2006].

However, a main problem with component repositories is that in order to be acquired by a company the one needs to re-organize and re-catalog the available components in order to populate them in the new repository, being sometimes an inhibitor to its adoption. Moreover, already existing software are also available in hundreds of repositories in the internet, such as SourceForge.net [SourceForge 2007]. At these on-line repositories, developers can obtain a large amount of Object-Oriented (OO) program source codes and binary codes. There is a possibility that programs, which partially fulfill the required functionalities, exist among these available OO programs source code. If such parts of existing OO programs could be easily reused as components, programmers could develop software by means of Component-based Development (CBD) by utilizing these programs.

In this context, this work proposes a component identification tool that investigates candidates for components from OO source code repositories. While this goal is easy to state, there have been several barriers. Reengineering can help in extracting reusable components from legacy system, but the efforts needed for understanding and extracting should be considered - sometimes, modifying and adapting software can be more expensive and inefficient than programming the needed functionality from scratch; furthermore, it could not be suitable for reuse [Sametinger 1997].

This project is part of the RiSE[1] project whose main idea is to transfer the state-of-the-art in the area to industrial environments in order to increase the productivity, quality and reduce costs.

[1] The RiSE – Reuse in Software Engineering – group approaches the software reuse area, proposing a framework for systematic software reuse adoption. This framework has been validated in many areas like software component certification, software reuse process, software reuse metrics, domain analysis, software architecture evaluation and software component search and retrieval. See www.rise.com.br

1.2. Objectives

This work aims to specify a tool for reverse engineering, in order to help engineers in identifying reusable parts in object-oriented systems which can be candidates for components, and therefore be stored in a component repository. The extraction of reusable software components from entire systems is an attractive idea, since software objects and their relationships incorporate a large amount of experience from past development. This assumption is supported by the general reuse principle: Re-work avoidance can shorten time, reduce resources consumption and improve quality.

Modularization is unlikely to ever be a fully automated process, but it would be helpful to have a tool that suggests a number of potential modularizations from which a software engineer might reasonably choose [Siff and Reps 1999]. However we are also looking for minimize the need for domain application experts.

A great deal of existing works in this field relies on a "ideal" domain, i.e., considering always commented code, available architecture, grouped functionalities, experienced analysts, and other heavy constraints. Of course there is no absolute solution to the component identification problem. But what we want here is to find the best balance among the selected techniques that brings up the best results in the industrial context. In order to successfully accomplish it, we decomposed this goal in four subparts, which are our tactical goals:

a) We must understand the differences between OO and Components;

b) We must characterize a component in terms of their attributes. Those attributes will serve as criteria for our component identification heuristics;

c) Then, we are able to define a set of heuristics which deals with those attributes in order to identify the candidate for component;

d) We want to automate some part of this process.

To reach a) we will study what makes components different from objects. By analyzing existing components definitions and defining a set of attributes, we accomplish b). We reach c) by doing a critical review on existing approaches on component identification. Finally, we present a specification of a tool to automate the process and hence, satisfying d).

1.3. Overview this Work

This work is organized as follows:

- In Chapter 2 a detailed description of software components issues and characterization;
- Chapter 3 presents a discussion of the existing approaches for component identification; it also summarizes the main features of those approaches;
- Chapter 4 describes a specification for an automatic component extraction tool. This chapter also presents and discusses heuristics that comprise some issues covered by the Chapter 3;
- The conclusions of this research effort and future research directions are stated in Chapter 5.

2 Component-based Software Development

This chapter discusses the problem of characterizing a software component, which is essential to understand what components are and how they can be classified and measured. First, we introduce the notion of component from well-known definitions in the literature. We also discuss about what makes a component different from an object. Then, we classify what should be characterized about a component according to its outstanding characteristics. We end this chapter introducing the Component-based Reengineering, our approach for component identification.

2.1. Introduction

Software components were introduced in 1968 during the Nato Conference by Douglas McIlroy in his article “Mass Produced Software Components” [McIlroy 1968]. At that time, software components were proposed as a mean for improving software quality and reducing development costs to address the so-called software crisis.

Component-based engineering has emerged as a viable alternative to reach productivity gains, accelerated time to market and lower development costs in conventional software engineering. The industry and academia tend to accept the idea of building large computer systems from small pieces called components that had already been built before, increasing productivity during the development phase as well as the quality of the final product [Almeida et al., 2007]. This idea follows one of the basic principles of Computer Science: divide to conquer.

Although much commented, there is a common misunderstanding when distinguishing software components. Many would refer to it as Javabeans, CORBA or COM objects; other would refer to components as fragments of source code or a functional procedure. Traditionally, a software component is defined as a self-contained piece of software with well-defined interface or set of interfaces [Kozaczynski 1998]. It is also important to empathize that the notion of software component is not the same as assets. According to [Ezran et al. 2002], software assets are composed of a collection of related software work products such as documents, architectures and eventually code that may be reused from one application to another. A software component may be an asset, but the contrary is not true. Some consensual definitions for software components are presented later in this chapter.

2.1.1. Similarities and Differences from Object-Oriented Paradigm

The goals of "componentware" are very similar to those of object-orientation: reuse of software is to be facilitated and thereby increased; and software shall become more reliable and less expensive [Lee et al. 2001].

Features of OO paradigm such as encapsulation and abstraction lead to a more flexible and extensible software product. Object-oriented paradigm, however has failed to deliver its expectations with respect to productivity gains in particular related to reuse.

In OO paradigm, a class describes the structure and behavior of objects. Objects encapsulate information (information hiding). This capability is highly relevant for component-based systems: client components are not interested in the internals of server components, only the services that are specified in their interfaces should be required for using those components [Hasselbring 2002].

However, even though object oriented technologies are doing quite well in a number of ways, they also suffer from several drawbacks. Beneken et al. [Beneken et al. 2003] pointed some of them:

- Object interfaces are one-directional: Only the incoming interface (import) is described, the outgoing interface (export) remains implicit;

- Objects cannot be deployed independently;
- Objects are tightly coupled, as they execute in the same process and data space;
- Objects generally only compose and cooperate if written in the same language.

Table 1 summarizes the similarities and differences between objects and components (based on [Szyperski 1998] and [Beneken et al., 2003]).

Table 1 Basic similarities and differences between objects and components

Objects	Components
An object is a unit of instantiation; it has a unique identity.	A component is a unit of independent deployment.
An object has state; this state can be persistent state.	A component has no persistent state.
An object encapsulates its state and behavior.	A component is a unit of third-party composition.

Software components also tend to be easier to reuse and maintain than objects, because objects have their functionality scattered throughout many entities. This implies in a bigger effort to deal with their relationships in order to find reusable parts that can be used independently.

Indeed, OO programming can serve as a good basis for component-based development, by combining functionalities of the objects and offering them as a single service, but it should not be seen as the only one. A well-organized library of functions in a procedural setting is a good basis too.

2.1.2. Business Components and System Components

There are two types of components in a component-based system. One of them is a system component, also referred as technical component, which

encapsulates a set of similar, reusable functionalities. The other is a business component that utilizes the functionalities of the system components [Cheesman 2001].

A system component is a unit of functional reuse or subsystem while a business component embodies a business process. The summary of the First International Workshop on CBSE [ICSE 1998] defines business components as "the software implementation of an 'autonomous' business concept or business process. It consists of all the software artifacts necessary to express, implement and deploy the concept as a reusable element of a larger business system". A further discussion about business components is presented in [Kozaczynski 1998].

In many cases, business components are aggregations of system components – i.e. system components are used as building blocks for logical components. But there is another important relationship between system and system components: Business components are more valuable to be reused, because they are more related to the application domain. A system component generally is more reusable, once it can serve across different business domains.

2.1.3. Classical Definitions

There is still a widely discussion about the meaning of the term component. Many definitions of components have been used throughout the software industry: [Sametinger 1997], [Szyperski 1998], [Brown 1998], [Souza 1998], [Hall], [Kozaczynski 1998], [Yacoub 1999], [Heineman & Councill 2001], [Beneken 2003].

In [Broy et al., 1998] for example, is presented an interesting discussion about the theme with some of the most renamed researchers in the field. Those definitions differ in the kind of aspects covered by a component or the emphasis to a certain issue. In the following, we briefly review some of them.

According to Sametinger [Sametinger 1997]: "Reusable software components are self-contained, clearly identifiable artifacts that describe and/or

perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status".

The Gartner Group, a very influential organization, provides a definition of a Software Component [Gartner 1997]: "A run-time software component is a dynamically bindable package of one or more programs managed as a unit and accessed through documented interfaces that can be discovered at run-time".

Brown et al. [Brown 1998] stated: "A component can be considered an independent replaceable part of the application that provides a clear distinct function".

In [Souza 1998] Souza complemented Brown's definition: "A component can be a coherent package of software that can be independently developed and delivered as a unit, and that offers interfaces by which it can be connected, unchanged, with other components to compose a larger system".

To [Kozaczynski 1998], "A component is a part of a system that is (at the same time) a unit of design, construction, configuration management, and substitution. A component conforms to and provides the realization of a set of interfaces in the context of well-formed system architecture."

Another definition well accepted in the academy is Heineman's [Heineman & Councill 2001] that says "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a component standard".

Today, Szyperski's [Szyperski 1998] definition is cited most frequently: "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties." Here, Szyperski emphasizes the composition and deployment of components.

The most important thing to note in these definitions is that components have a number of points of interconnection: each point could be termed an 'interface', with some of these interfaces being provided for other components to use, while other interfaces are required from other components. Components

are completely ‘encapsulated’: there is no way of doing anything with a component other than by using its provided interfaces. The kind of ‘unit’ is important: As a technical goal, minimal coupling with the outside world and maximal cohesion inside the unit is desirable.

2.1.4. Commercial Component Models

The reusable component which satisfies the definitions given above, has no dependence on elements outside of itself and can be instantiated and used alone. This assumption is very generic and can refer to many different concepts in practice. Component models provide standards for component implementation and interoperability. Also, they can encapsulate services and provide infrastructure such as a meta-information facility, naming and trading services, and transaction monitors.

According to [Voelter 2003] there are two main uses for technical components. They are either used in client applications, or, on the server, known as distributed components. The most commonly component technology used in client side includes JavaBeans of Sun Microsystems and ActiveX of Microsoft, based on their Component Object Model (COM).

The concept behind JavaBeans is to enable the composition of components, known as beans, through a composition tool called Bean Box. Bean components communicate by sending events to one another using a publish/subscribe model.

COM is a binary standard which interface specifications are assigned a globally unique identifier (GUID) at the time they are created; each revision of that interface is assigned a new GUID. Components are binary implementations that are bound to the interfaces they implement via these interface GUIDs. Similarly, clients are linked to components via interface GUIDs.

In case of server-side components, there are three mainstream examples: Enterprise JavaBeans (EJB), Microsoft's COM+ and CORBA Components. They are used in enterprise business applications. All reside in a container which

takes care of transactions, security, load-balancing, failover and other features. They provide meta-information, mainly for use at build time.

An interesting technology which is not properly a component model, but a specification for assets is the Reusable Asset Specification (RAS). RAS assets cannot be deployed; it just provides a standard way to package and extract a set of related files. In this context, RAS can be useful to encapsulate high-level information.

2.2. Component Attributes

In order to classify components, we first must specify the set of components aspects and attributes we want to analyze. The following attribute definitions for software component are commonly cited throughout the literature: [Broy et al., 1998], [Hall] [Kozaczynski 1998], [Yacoub 1999].

Hasselbring [Hasselbring 2002] divided component attributes in both managerial and technical. For him, the managerial goals for Component-based Engineering are:

- Cost Reduction
- Ease of Assembly
- Reusability
- Customization and Flexibility
- Maintainability

And the technical features of component-based systems are described as:

- Coupling
- Cohesion
- Granularity

Yacoub et al [Yacoub et al. 1999] do not consider managerial issues; instead, they afford a set of features ranging from component internals to the environment where the component is deployed. They distinguished what needs to be characterized about a component under three main categories: the Informal Description, Externals, and Internals. For each one they defined a set of features to characterize a component as it follows:

a) Informal Description

These attributes are related to human-related issues. They help to better understand the component purpose and its context, as well as information for component management.

- Age
- Source
- Level of Reuse
- Context
- Intent
- Related Component

b) Externals

Component external define its interactions with other application artifacts and with the platform on which the component resides. This category encompasses the following characteristics:

- Interoperability
- Portability
- Role
- Integration Phase
- Integration Frameworks
- Technology

- Non-Functional Features

c) Internals

Component's internals aspects are related to the technical data about the component.

- Nature
- Design Components
- Specification Component
- Executables
- Code
- Granularity
- Encapsulation (Decision Hiding)
- Structural Aspect
- Behavioral Aspect
- Accessibility to Source Code

Although we recognize that component-based software engineering has impact from both a managerial and a technical perspective, we will focus here in technical aspects of components because we are assuming that there is too little managerial data about the existing system. On the other hand, we consider that there is technical data available from the system itself.

One straightforward classification is that proposed by Sametinger [Sametinger 1997]. Sametinger's classification describes the essential attributes for CBD as being: Functionality, Interactivity, Concurrency, Distribution, Forms of Adaptation, and Quality Control. The following sections treat each cited attribute. Caldiera and Basili had a similar approach in [Caldiera and Basili 1991] in terms of reusability metrics rather than component metrics. They defined cost, quality and usefulness as reusability factors which should be

addressed by cyclomatic complexity, regularity, reuse frequency and code volume metrics.

2.2.1. Functionality

The concept of functionality is comprised in the concepts of applicability, generality and completeness:

- Applicability is the degree of how much is a component suitable for a domain. Depending on the domain, a component might be highly reusable while on others it is low and even being not reusable at all.
- The generality indicates if a component has more specific functionality or not. Excessive generality leads to complex components and unnecessary overhead in both execution time and resource consumption.
- Completeness is the degree of measure when a component offers the functionality expected by users in its intended reuse scenarios.

2.2.2. Interactivity

Component's interactivity is how a component relates with others and whether everything it needs to work is within. If a component is strongly connected with others, it can discourage the reuse even if it is technically possible, because all the other components on which it depends might have to be incorporated into the design. Therefore, a component should have a high degree of conceptual unit and their dependency on other components should be small.

2.2.3. Concurrency

This attribute is concerned with the simultaneously execution of events in a component. A component should be ready to receive multiples inputs and give the corresponding output correctly. The concurrency used to gain execution speed and to eliminate potential processor idle time. Synchronization is also necessary if two components, for example, share the same kind of resource.

2.2.4. Distribution

A distributed system consists of independently executing components. Distribution is essential for scalability and maintainability of the system, since new components could be added or replaced to the system without affect the overall architecture. It is also important to keep the system independent from individual component vendors.

2.2.5. Adaptability

Adaptability is the capability of a component provides means for easy configuration and modification. This is important once some adaptations are required before a component is going to be inserted into a system. Configuration is the process of adaptation which does not affect the component behavior. In modification the primary functionalities may be changed.

2.2.6. Quality

Quality control is related to component error-proneness and robustness. The process of quality checking is so complex that there is no formal verification of quality even for small components, given the heterogeneity of a component system.

2.3. The Envisioned Component Model

The proposed component guidelines below attempt to capture its important property of being a conceptual unity of design, construction and deployment.

- Component encapsulates business process;
- Stable component is placed into lower level than unstable component;
- The dependency between components must be minimized;
- The inner classes number of components is small enough to be managed;
- The complexity of a component is small enough to be understood;
- Static and dynamic aspects of a software system must be considered;
- It must have scalability to large software systems;

- It should deal with enterprise standards such as Spring[2] and Hibernate[3]

2.4. Component-based Reengineering

The reusable components engineering's goal is to produce an infrastructure for reuse. It covers the identification and the specification of components, their organization and their implementation [Aniorte 2002]. The activity of identification and specification of reusable components is essential in the process because they have the objective to produce or identify potential reusable resources. The CBD approach with reverse engineering is considered as down-top.

[Chikofsky and Cross 1990] define Reverse Engineering as "the process of analyzing a subject system to identify its components and interrelationships in order to create representations in another form or at a higher abstraction level". Today, a great number of techniques and methods have been proposed to face the software reconstruction problem. They can be classified in (i) Source-to-Source Translation, (ii) Object Recovery and (iii) Specification, (iv) Incremental Approaches and (v) Component-Based Approaches [Almeida et al., 2007].

Particularly, we are interested in Component-based approaches. It is characterized with the exploitation of existing development products to provide components. This approach can be declined under several forms as clustering analysis and program slicing. We will treat them deeper in the next chapter.

2.5. Chapter Summary

In this chapter we:

- Introduced the notion of software components and viewed definitions from the literature;
- Discussed the most important differences from the Object-Oriented paradigm;
- Understood the difference between system and business components;

[2] Spring, the application framework. http://www.springframework.org/

[3] Hibernate, the Java persistence framework. http://www.hibernate.org/

- Reviewed component models most frequently used in the software industry;
- Analyzed components attributes proposed by Sametinger [Sametinger 1997];
- Finally, we briefly introduced the Component-based Reengineering as our approach for component identification.

3 Component Identification

We have discussed in previous chapters what characterize a component in terms of their attributes and what the “ideal” component model is. It lightened our comprehension about components and provided criteria for the heuristics.

This chapter presents a review on the related areas of component identification with emphasis to software clustering techniques. It is organized as follows: Section 3.1 presents a literature review on relevant methods, techniques and tools to component identification. A set of requirements for an efficient component extraction tool is discussed in Section 3.2 and Section 3.3 summarizes the review on component identification and its application to the reengineering context.

3.1. Related Research

While a great deal of research over the past years has been devoted to the development of methodologies to create reusable software components, the issue of how to identify and extract reusable components from existing systems has remained relatively unsatisfied.

Currently unresolved issues include reducing the functionalities dispersal and increasing modularity, which assist in the maintenance and the evolution of the reengineered systems [Almeida et al., 2007].

There are several research trends on reverse engineering, but the main methods for component identification are classified into those three categories:

Clustering Analysis, Program Slicing and Component Extraction [Washizaki 2005].

In the sections below, we will focus on Clustering Analysis in order to decompose software systems into meaningful subsystems that can be reused.

3.1.1. Notable Software Clustering Techniques

Cluster analysis techniques have been used in many areas such as biology, economics, information retrieval, pattern matching and so on, to solve a wide spectrum of problems. The objective of clustering techniques is the grouping of items in such a way that the relations between them in the same group are stronger than the relations to items in other groups. Software clustering approaches are being considered in reverse engineering, mainly due to their benefits in reuse and maintainability. In [Lung 2004] is presented some clustering methods to software partitioning.

Traditional methodologies for software clustering can be classified into five different categories, namely domain-model-based, dataflow- based, connection-based, metric- based, and concept-based approaches [Koschke 1999]. This classification is just an approximation; some approaches can eventually belong to more than one category, while others can be combined (e.g., domain-models with metrics). In the next sections we will review the most interesting approaches in those categories.

3.1.1.1. Domain-model-based

The domain-model-based approaches use domain models as inputs that describe the domain concepts and their relationships. The approach proposed for identifying business components described in [Jain 2001] uses an analysis level domain model as input. A clustering method is used to obtain an initial set of components. Then, it considers super-type/subtype relationships and a set of heuristics to enhance and refine the solution obtained from the clustering algorithm. These heuristics can be either manual or automated and work regrouping the classes in the initially identified components. The approach has

been implemented in a tool named 'CompMaker' and was used for identifying components for an auto-insurance claims domain.

In [Chao 2005] is proposed a hierarchical clustering technique based on graph using edge strength notion, i.e. class relationships can be weighed according their types. This strategy attempts to blend the advantages of clustering analysis and graph-based decomposing methods and focus on acquiring component from domain business model.

To identify business object component, it uses the concept of resemblance degree between business objects. The resemblance degree depends on the relationship among the objects either dynamics or statics. The approach is composed of two steps:

1) The object-activity relation matrix (OARM) is used to calculate the dynamic resemblance degree. The dynamic relationship represents the communication of information and the transition of states between business objects through business activities. The relationships between business object and business activity can be mainly classified as create (C) and use (U). The Static relationships between business objects can be classified as general, compound, aggregation and association, and have different weights. The static resemblance degree between business objects is equals to sum of weights of static relationships between them.

Table 2. Object-activity relation matrix

activity / object	BA_1	BA_2	BA_3	BA_4	BA_5
BO_1	C		U		U
BO_2	U	C	U		C
BO_3	U			U	

2) To identify business object components, it uses a hierarchical clustering technique based on business object relation graph according to the edge strength metric. The strength of edge considers both cohesion and coupling between business objects. The hierarchical clustering technique requires inputting the edge strengths for all possible pairs of business objects to be clustered, and produces dendrograms as the output (Figure

1 and 2). The algorithm stops when it reaches the maximum number of business objects that are allowed in a business component (Figure 3).

Different to traditional hierarchical clustering, this approach considers edge strength rather than edge weight. However, it could be impractical to medium-big programs since it relies on the number of possible pairs of business objects.

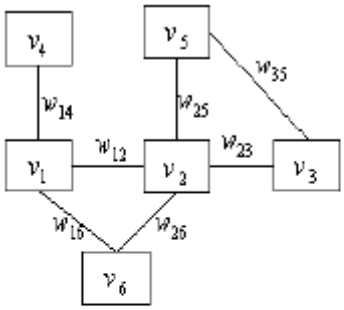

Figure 1. Business object relation graph

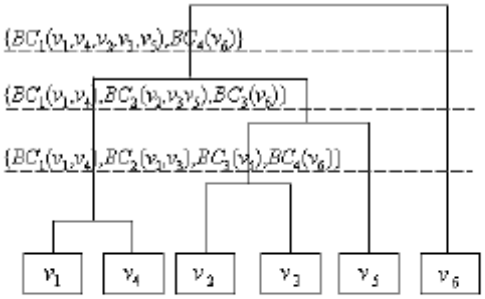

Figure 2. Dendrogram

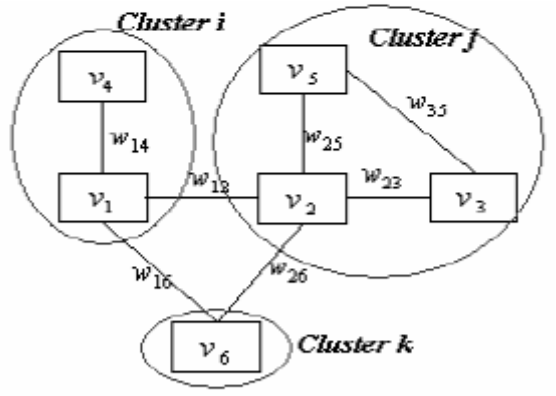

Figure 3. A cluster result

[Choi 2006] classified the characteristics of dependency relationships between classes for an object-oriented domain model in terms of their static and dynamic aspects. The classification method is based on method call types.

The static dependency is classified according to the structural characteristics between classes. Those structural relationships between classes are composition (aggregation), inheritance, and association. The dynamic dependency is related to the type of the message call between classes and its direction. The dynamic relationships are create, delete, write and read while their directions are usually unidirectional and bidirectional. The approach is two-tiered:

1) Firstly, a set of heuristics is applied in order to identify system components. Theses heuristics consists, for instance, in grouping classes that have most similar functionalities in system components candidates. Table 3 shows the use cases and class relationships. Then, is necessary to specify the type of class relationship in terms of their use and direction, assuming that these types represent different connectivity strength (e.g., create > write). As stronger as the relationship is, more they are likely to belong to the same component.

Table 3. Use case diagram and class relationship table

Actor	UseCase	Classes										
		*C*1	*C*2	*C*3	*C*4	*C*5	*C*6	*C*7	*C*8	*C*9	*C*10	*C*11
Actor 1	Usecase 1	√		√								
	Usecase 2	√		√								
Actor 2	Usecase 3		√	√	√							
	Usecase 4		√	√	√							
	Usecase 5			√	√							
	Usecase 6				√	√	√			√	√	
Actor 3	Usecase 7				√	√	√					
	Usecase 8				√	√	√			√	√	√

2) Another set of methods is used to identify business components from these system components. Grouped classes are identified as one candidate business component.

This approach consider identification of system components (constituting a subsystem) prior to business component, using the type of the relationships among the classes. It also considers the direction of message calls which may be relevant information since it changes the dependency relationship.

The drawbacks of this approach are the intensive use of use case, class, and sequence diagrams. An analyst with high knowledge of the domain is necessary since most of the classes reallocation relies on his intuitions. This approach seems more suitable during the development process rather than reengineering purposes.

We have seen that these approaches use business model as input and this assumption is not always possible in some cases. Such dependency relies much on the intuitions of individual developers and it is mandatory the consideration of the entire domain.

3.1.1.2. Dataflow-based

Dataflow-based approaches leverage dataflow information to identify object candidates and transform procedural programs into object-oriented programs.

An approach was described in [Valasareddi 1998]. A program representation model, called Statement Dependence Graph (StDG), is used to identify object state and behavior from the program structure. By representing the program as a graph, the dependences are determined through graph traversal. Code is localized by bringing together all statements with high cohesion which are scattered throughout the program. Statements with high cohesion are merged through graph compaction. The state variables of an object are taken from the exposed “defs” based on their usage information.

3.1.1.3. Connection-based

Connection-based approaches cluster entities based on a specific set of direct relationships between entities to be grouped. The main difficulty in connection-based approaches is the identification of the sub-graphs that can be interpreted as groups of data and related functions.

A method to identify architectural component candidates in a hierarchy of procedural modules has been proposed in [Girard and Koschke 1997]. The dominance analysis of the relation on the call graph is performed to group functions/variables into modules and subsystems as component candidates. It proceeds according to the following steps:

1) It identifies atomic component candidates which are abstract data types (ADT), abstract state encapsulations (ASE), and groups of mutually-recursive routines;
2) This dominance analysis is performed on the call graph to identify support routines, modules, and subsystems. Dominance analysis produces a tree which captures whether a routine implements a service for one or for many callers (Figure 4); and
3) Distribute variables outside atomic components into subsystems.

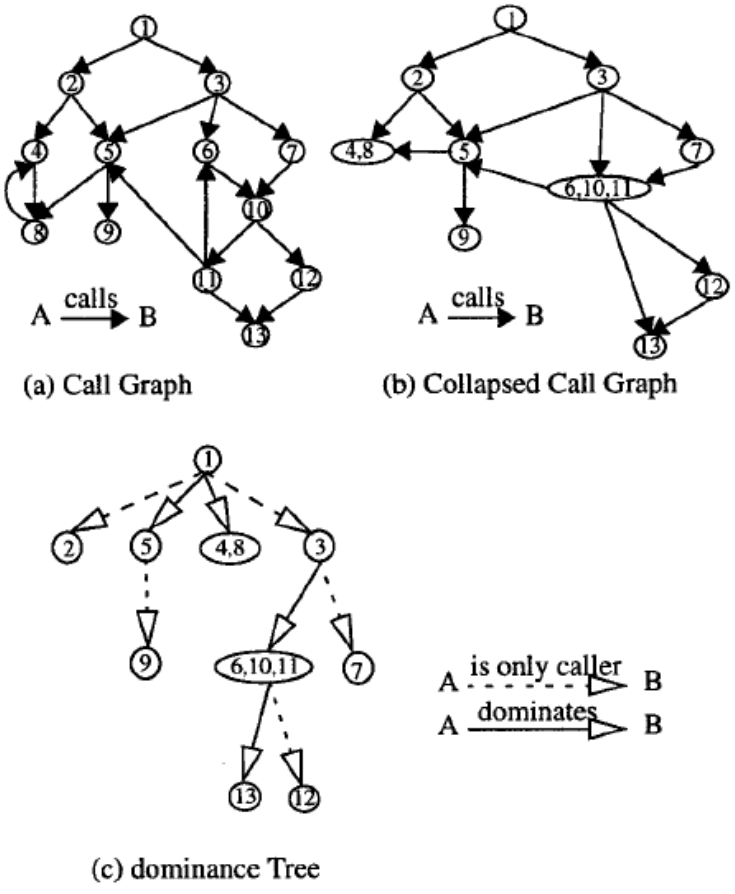

Figure 4. A dominance analysis example

A developer should validate the results of each step before proceeding to the next. A developer should interpret the results of the method. Another application of dominance analysis on call graphs to identify candidates for reusable modules was proposed by Cimitile and Visaggio in [Cimitile and Visaggio 1995]. In their approach they collapsed cycles before dominance analysis gets applied.

An approach to design and analyze software systems is presented in [Jordan 2005]. The code dependencies are extracted from the code by a conventional static analysis, and shown in a tabular form known as the 'Dependency Structure Matrix' (DSM). A variety of algorithms are available to help organize the matrix in a form that reflects the architecture and highlights patterns and problematic dependencies.

The research presents the Lattix Dependency Manager (LDM) tool, the implementation of DSM analysis for software. The LDM tool extracts static information from software systems like packages as well as the flow of information to present them as a dependency matrix.

The analyst can automatically organize the elements and obtain a hierarchical structure based on the flows and compare these dependencies with the rules of the project previously specified.

		1	2	3	4
Task A	1	.		X	X
Task B	2		.	X	
Task C	3	X		.	X
Task D	4				.

Figure 5. A simple DSM example

The DSM seems to scale well and provide a nice view of the system. However, DSM, while inherently more scalable a representation of relations than graphs, are not always easy to read. Also the current implementation does not qualify the relationships among entities.

[Mitchell and Mancoridis 2006] have developed a software clustering tool called Bunch. Bunch produces subsystem decompositions by partitioning a

graph of the entities and their relations in the source code. The process is as follows:

1) The source code is transformed into a Module Dependency Graph (MDG). The MDG is a language independent representation of the structure of the system's source code components and relations;
2) The clustering algorithms treat the graph partitioning as a search problem where their goal is to maximize Modularization Quality (MQ). MQ determines the quality of an MDG partition in terms of inter-connectivity and intra-connectivity. Bunch's hill-climbing algorithms move modules between the clusters of a partition in an attempt to improve MQ. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. It converges when no partitions can be found with a higher MQ.

Unlike other software clustering techniques, this approach evaluates the quality of a graph partition that represents the software structure, and it uses heuristics to navigate through the search space of all possible graph partitions. Such algorithm can perform static and dynamic analysis and language independent for graph representation (Module Dependency Graph - MDG) can be very useful for third-parties applications.

3.1.1.4. Metric-based

A great deal of metric-based approaches found in software clustering literature is focused mainly in coupling and cohesion measures. This lead us to two questions: Are they definitively good for this purpose? Or are we being over-persuaded? [Abreu 2001]

One of the first methodologies involved is the one proposed by Caldeira and Basili [Caldiera and Basili 1991]. They have developed a system that allows identification of reusable components in Ada and ANSI C legacy software based on software metrics.

The CARE system splits the analysis of existing programs into two phases. The first phase identifies some candidate components, and packages them for possible independent reuse. An engineer with knowledge of the

application domain in which the component was developed then analyzes each component to determine the service that component can provide. Finally, components are stored in a repository along with all the information that has been obtained about them.

They defined cost, quality and usefulness as reusability factors which should be addressed by cyclomatic complexity, regularity, reuse frequency and code volume metrics used by the CARE system. These metrics attempts to capture different and relevant aspects of a reusable software component. However when combining unrestrictedly those measures it may bias an eventually better result of a few, well-equalized metrics.

Another work in automated reuse identification is presented in [Dunn and Knight 1993]. They have developed a tool called CodeMiner. CodeMiner assists the programmer in identifying parts of legacy C software systems that might be potential candidates for a reuse library. CodeMiner identifies potentially reusable software in three ways:

1) It identifies functions that are invoked multiple times from multiple sections of code;
2) It identifies functions that are loosely coupled. The types of coupling checked for in Code Miner are:
    a) data coupling: the functions share data through their interfaces;
    b) common coupling: the functions share global data;
    c) external coupling: the functions share data with the outside world, and;
    d) control coupling: the functions share data items upon which control decisions (such as branching) are made.
3) It identifies functions that use the same global data elements and can be grouped together to form abstract data types. Each function and global data is represented as nodes in a graph. Strongly connected component is reported to the user as potential abstract data type.

One of the problems of this approach is the reusable criteria which are based whether a component has at least one of the coupling functions or not. Candidates with varying degrees of coupling are not distinguished and do not

serve as possibly reusable. Furthermore, aspects of the code relevant to reusability, such as functional complexity, have not considered.

[Lee et al. 2001] Proposes a component identification process based on coupling and cohesion measures. Although this approach is combined with domain models, the focus on metrics is the key feature of this work. The approach comprises in six steps:

1) Subsystems and dependencies are defined using sequence diagram;
2) Considering component cohesion: Classify classes in a use case into five class types according to the amount of business logic or whether they are key entity classes. Classes belonging to the same class type are candidates for components.
3) Considering component coupling: The interaction coupling is calculated from the number of methods invoked in a class. Static coupling is represented by class association, composition and inheritance. For each kind of static relationship is assigned a weight corresponding to its coupling strength.
4) Component Interface: The developer determines what use case pair must be not included in a component and what use case pair must be included in a component in order to set the services a component must provide.
5) Considering Dependency among Components: To decrease import dependency, after identified a component, component interfaces are managed in other package. This can remove many components cyclic dependency.
6) Component clustering algorithm: A clustering algorithm is applied to identify components using the suggested metrics. To choose adequate components among many clusters, developer can use the domain knowledge to identify components.

In many steps this approach depends much on subjectivity of the analyst. Moreover, cannot be easily automated.

Because software metrics take into account large volume of source code that must be analyzed to find reusable parts, they provide a way to automate the

first steps of the analysis. The automated techniques can reduce the amount of code that a domain expert needs to evaluate to identify reusable parts. Metrics can capture different and relevant aspects of a reusable software component, however, when combining unrestrictedly those measures it can bias an eventually better result of a few, well-equalized metrics.

3.1.1.5. Concept-based

A concept analysis is a branch of lattice theory that can be used to identify similarities among a set of objects based on their common attributes [Wille 1981]. [Canfora 1999] says “Concept analysis is a very useful tool for gaining an architectural comprehension of the subject system because it decomposes the system into groups of related programs and data stores and outlines the relations between different groups”.

Concept analysis is more general than connection-based methods as it can capture the same kinds of relations depicted in graphs and presents several additional advantages. These include a finer control over the granularity of the obtained modularization and an improved discriminatory factor.

Among the first research works in this direction, [Sahraoui et al., 1997] presents an object identification approach based on Galois lattice [Godin et al., 1995]. The approach consists of four steps:

1) First, an abstract syntax tree (AST) is built from the program. This AST is used by a pattern recognition and transformation program to extract the reference graph information; The AST is represent in a tabular form as shown in Table 4. Representation of binary relation R.
2) Then, we identify candidate objects from the corresponding Galois lattice (Figure 6);
3) In the third step, objects are identified from variables that are simultaneously accessed by a large number of functions;
4) Finally, a set of rules is applied in order to identify the methods of these objects from procedures/functions.

Table 4. Representation of binary relation R

		E'								
	R	a	b	c	d	e	f	g	h	i
	1	1	0	1	0	0	1	0	1	0
E	2	1	0	1	0	0	0	1	0	1
	3	1	0	0	1	0	0	1	0	1
	4	0	1	1	0	0	1	0	1	0
	5	0	1	0	0	1	0	1	0	0

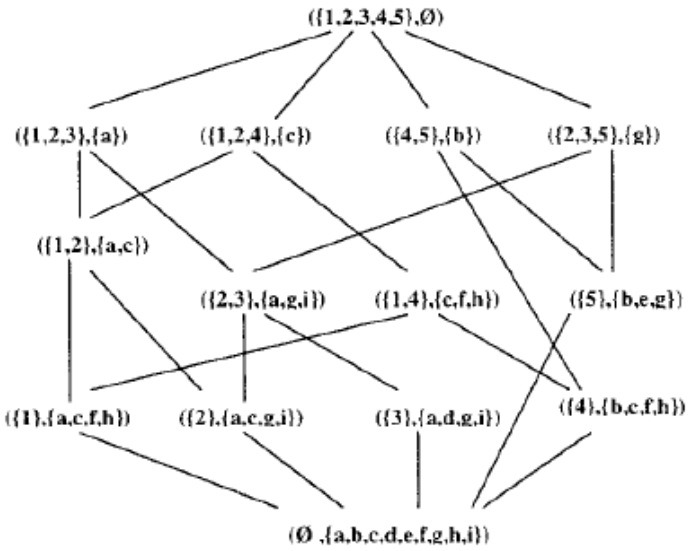

Figure 6. Galois lattice for relation R

It considers the way in which the function uses the variable. It can be write, read and predicate - when the variable is used to control the execution of the function (in a predicate).

Another work involving software components and reengineering may be seen in [Siff and Reps 1999], where Siff discusses a technique by which modules in this C++ classes can be identified from legacy C code. The main idea is to apply concept analysis to find potential modules. An outline of the process is as follows:

1) It is built a context, where objects are functions defined in the input program and attributes are properties of those functions. The attributes could be any of several properties relating the functions to data structures. Examples of attributes are: “uses global variable v”, “return type is r”, “has arguments of type t”, “uses fields of structured type t”.

2) The concept lattice is constructed from the context. In the example illustrated in Table 6. we have possible grouping of concepts varying from the max number of objects to the maximum number of attributes;
3) It identifies concept partitions-collections of concepts whose extents partition the set of objects. Each concept partition corresponds to a possible modularization of the input program. The algorithm to generate partitions from a concept lattice attempts to identify a collection of modules such that every attribute is associated with an atomic concept. An atomic concept is a concept that cannot have the same attributes of other concept or the same attributes of a composition of others concepts.

Table 5. The context for the Stack and Queue example

	returns stack	returns queue	has stack arg.	has queue arg.	uses stack fields	uses queue fields
initStack	✓				✓	
initQ		✓				✓
isEmptyS			✓		✓	
isEmptyQ				✓		✓
push			✓		✓	
enq				✓		✓
pop			✓		✓	
deq				✓		✓

Table 6. The extent and intent of the concepts for a Stack/Queue example

top	(*all objects*, ∅)
c_5	({initQ, isEmptyQ, enq, deq}, {uses queue fields})
c_4	({initStack, isEmptyS, push, pop}, {uses stack fields})
c_3	({isEmptyQ, enq, deq}, {has queue argument, uses queue fields})
c_2	({isEmptyS, push, pop}, {has stack argument, uses stack fields})
c_1	({initQ}, {returns queue})
c_0	({initStack}, {returns stack})
bot	(∅, *all attributes*)

A tool has been developed to support the conversion of a C program to a C++ program using this approach. One of the advantages of using concept analysis is that multiple possibilities for modularization are offered. In addition, the relationships among concepts in the concept lattice also offers insight into the structure within proposed modules.

3.2. Techniques Comparison

Some techniques for component identification are compared in Table 7. Those techniques were chosen according to its relevance in the literature. Each technique was compared according to the following criteria:

Main feature: Describes the kind of strategy of the approach.
Domain Knowledge: Represents how much domain knowledge is necessary in the process.
Static/Dynamic Analysis: It means the capability of the technique to cover static and dynamic aspects of the program. Dynamic analysis can capture dependencies that are not specified in the source code (e.g., asynchronous calls, methods that are linked at runtime).
Automated: It is how much of the process can be performed by a software. It represents also a measure of user interference.
Representation: Corresponds to the kind of visualization of the program structure, if there is.
Input: Input represents what is needed to perform the technique.
Output: It is the result given by the technique.

Table 7. Component Identification Approaches Comparison

Attribute/Work	[Caldiera and Basili 1998]	[Siff and Reps 1999]	[Lee et al. 2001]	[Chao 2005]	[Choi 2006]
Main feature	Cyclomatic complexity, regularity, reuse frequency and code volume metrics	Use Gallois lattice for concept analysis	Refactoring techniques based on heuristics plus coupling, cohesion and complexity metrics	Hierarchical clustering based on business object relation graph	Analyzes the method call types between objects
Static/Dynamic Analysis	Static	Both	Static	Both	Both
Domain Knowledge	Low	Medium	Medium	High	High
Automated?	Fully	Partially	No	No	No
Representation	Textual	Concept lattice	N/A	Business objects relationship; dendrograms	Use case and class diagram relationship table
Input	Procedural code (C)	OO code (C++)	Any OO code	Business Models	Business Models
Ouput	Identified libraries	Concept partition	Identified components	Business Components	Business Components

Those methodologies may be very expensive when applied to a large amount of software, in terms of time consumed by the experts and the precision in identifying reusable components.

Both Chao's and Choi's use domain-models and need high interference of an expert, therefore give more precise results than the other techniques.

Calderas' and Basili's metrics technique does not require an expert and in addition it may be automated completely. However, it does not seem to be very accurate since the intervention of an expert is not possible and only static analysis is performed.

An interesting idea is then to use metrics before either concepts analysis or clustering to reduce the amount of code to be analyzed by the experts. Once potential reusable components have been identified using those methods, experts review them from the perspective of the domain of application. The techniques can be combined, other sources of information can be considered.

3.3. Chapter Summary

In this chapter we:

- Overviewed some of the most relevant research in clustering techniques for component identification: domain-model-, dataflow-, connection-, metric-, and concept-based approaches. We shortly introduced how each of the approaches works and what are their weakness and strengths.
- Summarized and compared the approaches according with defined attributes.

4 CORE Loader

In the previous chapter we have analyzed the existing approaches related to component identification. We believe that combine existing approaches, may overcome the limitation of single approaches and helps to better address the unique aspects of the subject system. Thus far, we have seen just a few industrial tools; several were not implemented in a tool or they are academic prototypes. Automation tools offer valuable help in the reengineering process, saving time, effort and costs.

In this chapter we present the specification, design and implementation of a tool to help engineers to identify components from Java[4] source code in repositories, the CORE Loader. It is so-called because this tool will be integrated with the CORE[5] – Component Repository in order to populate its asset base using existing Java programs.

This chapter is organized as follows: Section 4.1 presents the main requirements of the initial version and section 4.2 shows the CORE Loader architecture and its modules. Section 4.3 introduces some tools and frameworks as well as some implementation details.

4.1. Basic Requirements

The envisioned reusable component has no dependence on elements outside of itself, and can be instantiated and used alone. However, the complexity for

[4] JAVA programming language. http://java.sun.com

[5] CORE – Coponent Repository – is an asset manager developed and commercialized by RiSE. See www.rise.com.br

achieving this could be costly and inefficient. Instead, we want reach a balance between the automated techniques in order to reduce the amount of code that a domain expert needs to evaluate to identify reusable parts.

The requirements were taken from commonalities among the existing approaches found in the literature, described in previous chapters. Some of them, like “Representation” was conceived due to the analysis of graph tools and the kind of interaction they offer. We must emphasize that those requirements are likely to change as the software is used in the real context.

4.1.1 Suggest Candidates for Componentization

We have seen that semi-automated approaches for component identification are good because they join the best of the two worlds: The calculation of several metrics and heuristics in little time; and the domain knowledge and reasoning of a human in the validation of results. Thereby, preliminary results of clusterization should be presented to the user as a suggestion for components.

4.1.2 Dependency Models

Excessive inter-module dependencies have long been recognized as an indicator of poor software design. Highly coupled systems, in which modules have unnecessary dependencies, are hard to work with because modules cannot be understood easily in isolation, and changes or extensions to functionality cannot be contained [Yassine 2004].

Analyzing dependency models of the source code is an attractive idea because it takes advantage of the structural characteristics of class relationships and the subservient nature of static and dynamic relationships stemming from the method call patterns and directions as well as assigning certain weight quantifying degree of dependency to each relationship.

4.1.3 User Interaction

Interaction is related to how the user interferes in the process in order to get specific, better results. Because modularization reflects a design decision that is inherently subjective, it is unlikely that the modularization process can ever be fully automated. Given that some user interaction will be required.

User should be able to tune types of heuristics and values, as well as evaluate the candidates for components suggested by the tool. This evaluation should occur in such a way that the user could manually change classes in the suggested components. It can also publish them in a component repository, for example.

4.1.4 Representation

Representation means the kind of visualization is used to present a view of the program structure. It can be either graph-based or matrix-based. We consider tree view and dendrograms as specifics kinds of graph.

4.1.4.1 Graph-based Representation

In graph based representation, each class represents a vertex in the graph and their relationships are represented as directed edges (Figure 7). Its graphical nature enables a better understanding of the structure; however, in large connected graphs this kind of representation could be even harder. We have chosen directed sparse graph structure as our mainstream representation of the code.

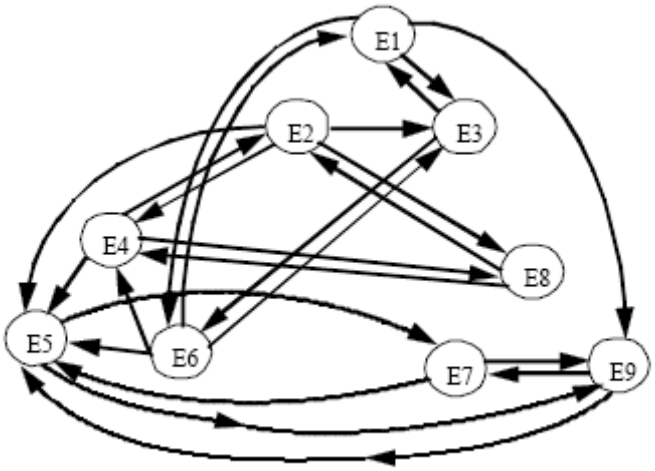

Figure 7. A graph representation example

4.1.4.1 Matrix-based Representation

Matrix-based representations shows the dependencies are extracted from the code in a tabular form. An example of such matrix is the Dependency Structure Matrix (also known as Design Structure Matrix - DSM) [Jordam 2005] as

present by Figure 8. Besides the application in various areas of process engineering, DSM is useful in the analysis of complex software and for representing the interdependencies within the software elements. We have discussed about DSM in the section 3.1.2.3 Connection-based approaches.

	E1	E2	E3	E4	E5	E6	E7	E8	E9
E1	1	0	1	0	0	1	0	0	1
E2	0	1	1	1	1	0	0	1	0
E3	1	1	1	0	0	1	0	0	0
E4	0	1	0	1	1	1	0	1	0
E5	0	1	0	1	1	1	1	0	1
E6	1	0	1	1	1	1	0	0	0
E7	0	0	0	0	1	0	1	0	1
E8	0	1	0	1	0	0	0	1	0
E9	1	0	0	0	1	0	1	0	1

Figure 8. An example of a matrix representing classes dependencies

4.1.5 Metrics

Because software metrics take into account large volume of source code that must be analyzed to find reusable parts, they provide a way to automate some steps of the analysis. The automated techniques can reduce the amount of code that a domain expert needs to evaluate to identify reusable parts [Caldiera and Basili 1991]. The most common metrics relies on coupling and cohesion measurements. As a technical goal, minimal coupling and maximal cohesion is desirable, so that all the elements in one component are closely related for the realization of a certain feature, and changes made to that component will have as little impact as possible on other components. These will be described below.

4.1.5.1 Coupling

Informally, coupling refers to how tightly or loosely bound a set of modules are to each other, a kind of connectivity strength. Functions that are loosely bound tend to be easier to remove and use in other contexts than those that depend heavily on other functions or non-local data. Determining coupling information is complex because there are several different types of coupling. Even harder in some design patterns implementations such as Inversion of Control (IoC) or Dependency Injection. The detailed explanation of how coupling is calculated in this project is done later in this chapter.

4.1.5.2 Cohesion

Cohesion is a measure of the extent to which the various functions performed by an entity are related to one another. Most metrics assess this by considering whether the methods of a class access similar sets of instance variables.

4.1.6 Ranking

Ranking is the process of attributing relevance on an element in a set of elements. This is done because we frequently want to get the most relevant elements prior than irrelevant ones. This work is the first to propose raking algorithms as measurement for modularization. We believe that some objects can be more relevant than others and thus, more valuable for reuse. Since it is relevant, it would be considered to be in a component.

Probably the best known of ranking algorithm for its application in the search engine Google, PageRank [Page et al., 1999] is a mechanism to compute a ranking to each page based on the graph of the web. Its main feature is to explore the hypertext structure of a page to represent associations and hence quantify and propagate the importance of a particular document for the web, a process known as "voting". For instance, an “A” page that has a link, or reference to a page “B”, is counted as a "vote" for page “B”. However, not only this information is taken into account, but also the score or importance of the pages that referenced “B”, in this case, the score of the page “A” influences in the scores of page B.

A PageRank analog algorithm for software objects was proposed by Inoue et al [Inoue et al., 2003] and it is called Component Rank. The idea behind the Component Rank is compute the score of the object from factors called "influence weights" that shows how an object is referenced, similar to PageRank described above.

Each file is initially processed by a mechanism for measuring similarity between objects, where from certain linear similarity of the objects are grouped through the clustering process so that duplicates nodes are removed from the graph.

Then, it is designed the structure of the graph to be mounted from the analysis of the relationships between components, i.e., method invocation, inheritance, implementation of abstract classes that represent the edges and nodes of the graph. The nodes weights are computed propagating the initial values of the nodes through its edges until the system stabilizes and all objects have defined their scores.

The results show that classes often invoked or inherited by other classes had generally the highest ranking and specific and independent ranking had the lowest.

4.1.7 Clustering

Software clustering has been large applied in reengineering to assist grouping of similar components and support partitioning of a system. With clustering, similar components are grouped together to form clusters or subsystems. Many clustering methods have been analyzed in [Lung 2004]. [Chiricota and Melançon 2003] also presents a good set of clustering metrics. Other works in this area were seen in the previous chapter.

In this work, we focus on the Hill-Climbing clustering algorithm adapted by Mitchell in [Mitchell 2002]. The same hill-climbing algorithm was implemented in the Bunch tool [Mitchell and Mancoridis 2006] described in the section 3.1.2.3 Connection-based approaches. Bunch's hill-climbing clustering algorithm starts with a random partition of the dependency graph and attempts to maximize Modularization Quality (MQ), which is a measure relating cohesion and coupling. Modules from this partition are then systematically rearranged in

an attempt to find an improved partition with a higher MQ. If a better partition is found, the process iterates, using the improved partition as the basis for finding even better partitions. This hill-climbing approach converges when no partitions can be found with a higher MQ.

Our approach considers MQ as being the average of clusters MQ. MQ is a measure directly proportional to cohesion and inversely to coupling. As higher the MQ is, better are the partitions.

4.1.8 Refactoring

Refactoring is a structural modification which does not change the program behavior and semantics. It is commonly used to eliminate redundancies, cyclic dependencies, increase modularity, and understandability. Automated refactoring can modify the surrounding parts of OO programs in order to improve component modularization and extraction.

A technique for automatic component extraction was proposed by Washizaki in [Washizaki 2005] called “Extract Component”. The Extract Component Refactoring conduces a static analysis of the dependencies among Java classes/interfaces. Next, a clustering algorithm is applied to detect all possible clusters by determining the reachability on the given program. A cluster is a candidate component, and has no dependence on elements outside the cluster. Finally, this candidate component is easily transformed in a JavaBean component.

4.1.9 Package Identified Components

The tool should have the capability to package, i.e., group identified classes of objects that form a component once it was validated by the user. This package must follows a standard specification and be ready to use. We defined our component model as being the RAS – Reusable Asset Specification[6].

[6] RAS, Reusable Asset Specification.
http://www.ibm.com/developerworks/rational/downloads/06/rsa_ras_assets/

4.1.10 Export to a Component Repository

Although it is possible to immediately reuse an identified component, there is no or very little support in the literature to the exportation of these identified components to assets repositories. The idea is that components could be uploaded to component repositories and thus maximizing the reusability potential of the asset.

4.2. Architecture

The architecture of the CORE Loader was designed to be extensible to different technologies and heuristics, affording the addiction of new ones. This capability is obtained by partioning the system in small encapsulated parts, with well-defined functionalities.

The first version of the CORE Loader is composed of three subsystems, namely, Pre-processor, Heuristics Engine and Graph Engine. An architectural view of the CORE Load system is presented in Figure 9. Each one has a specific purpose in the component identification process. A domain expert software engineer is encouraged to tailor and validate the results.

Pre-processor subsystem will retrieve and format the code from source code repositories. The Heuristics Engine comprises in all modules whose function is to perform calculation in the code structure in order to find possible component candidates. At the end, the Graph Engine will generate a nicely representation of the clustered results to the user. They all share and operate upon the GraphML[7], a language-independent graph representation of the code structure. Finally, CORE Loader provides its functionalities through a web service layer. This enables CORE Loader to communicate with third-party repositories and became possible to export the identified components to these repositories. A brief explanation about the modules goes as follows.

[7] GraphML, the graph file format. http://graphml.graphdrawing.org/

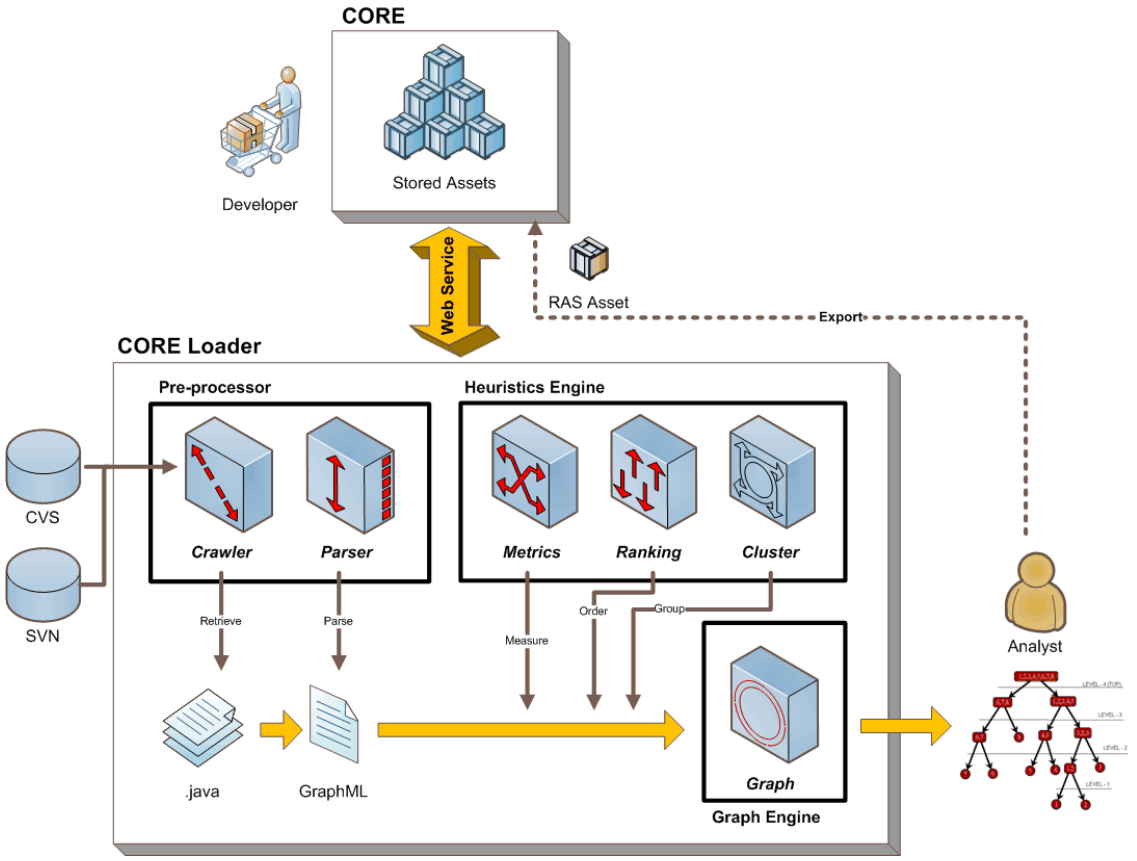

Figure 9. CORE Loader Architecture

Crawler: This module is responsible to check out and retrieve the source code from repositories such as CVS[8] and SVN[9]. The files are thus, passed to the Parser module. This module was already developed by RiSE and it being used in the BART[10] system.

Parser: The Parser receives the code from the Crawler and creates the Abstract Syntax Tree (AST) of the code. The AST is just a tree-form representation of source code. Every element of the source code is mapped to a node or a sub-tree. It will enable the Heuristics Engine to work upon this structure. We have chosen Eclipse's AST Parser[11] for code analyzing and parsing.

[8] CVS, Concurrent Versions System. http://www.nongnu.org/cvs/

[9] SVN, Subversion. http://subversion.tigris.org/

[10] BART, Basic Asset Retrieval Tool. http://www.rise.com.br

[11] Eclipse AST Parser from Eclipse JDT project. http://www.eclipse.org/jdt/

Metrics Module: This module performs the calculation of the Coupling and Cohesion as mentioned previously.

Ranking Module: The Ranking Module calculates the relevance of a class in the context of the graph. It relies on the voting principle, as described previously.

Cluster Module: The Cluster Module is responsible to blend the data from metrics and ranking modules in a cluster algorithm that will perform the grouping. These groups (clusters) of classes will be presented as candidates for componentization. We are implementing the Hill-Climbing clustering algorithm described in [Mitchell 2002]; however, due to time constraints we are currently using JUNG[12] pre-implemented clustering algorithms for preliminary studies.

Graph Module: The graph representation of the results is mounted by this module. To do this task, JUNG framework is being used.

4.3 Implementation

Our technique targets Java language as the OO programming language. Java technologies are widely applied in corporate software [Tiobe 2008]. We consider only programmer-made classes/interfaces.

Due to time constraints imposed on an undergraduate project we have implemented just a subset of the requirements mentioned above. The first prototype of CORE Loader is able to identify components using coupling measure and a clustering algorithm. We are currently working hard on the implementation of more heuristics to be added in the tool.

In this section we present implementation details of this first version of CORE Loader and evaluate a case study.

[12] JUNG, Java Universal Network/Graph Framework. http://jung.sourceforge.net/

4.3.1 Coupling Calculation

A visitor pattern[13] is applied to analyze each statement and retrieve dependencies of a certain class. These dependencies are thus measured according with the following attributes:

- Imports;
- Variable declaration (static, global and local);
- Method call;
- Inheritance;

For each dependency, is assigned a weight which corresponds to the connectivity strength of the relationship. For example, a “variable declaration” relationship has weight X, while “inheritance” has weight Y and Y > X because inheritance has stronger connectivity than variable declaration.

After distributing those weights, we also calculate the frequency in which those relationships occur. The resultant coupling measure is given as the sum of the results of the weight multiplied by the frequency for each relationship.

Note: It is not enough to look for a reference to a variable to determine if there is a class relationship. Retrieving biding information is necessary to characterize the relationship. Bindings provide extended resolved information for several elements of the abstract syntax tree.

4.3.1 Cluster Algorithm

The algorithm used for computing clusters in graphs is based on classes’ coupling. Coupling is the measure of how connected is an entity to the external word and it is given as the sum of edge weights. The premise is that loosely coupled classes should belong to distinct components. The weight of an edge is calculated as shown in section 4.3.1 Coupling Calculation. Edges which have low weights are progressively removed until the clusters have been adequately separated. This algorithm works by iteratively following the 2 step process:

- Compute edge weights for all edges in current graph;

[13] Visitor, the design pattern. http://en.wikipedia.org/wiki/Visitor_pattern

- Remove edge with lowest weight;

We plan to use the Hill-climbing algorithm as described in section 4.1.7 in our future work.

4.3.1 CORE Loader Main Features

An example of an application of CORE Loader executing on a small Java system is discussed throughout this section. Figure 10 shows the initial representation of the target system as a graph. Classes are vertices and the directed edges are the relationships among them.

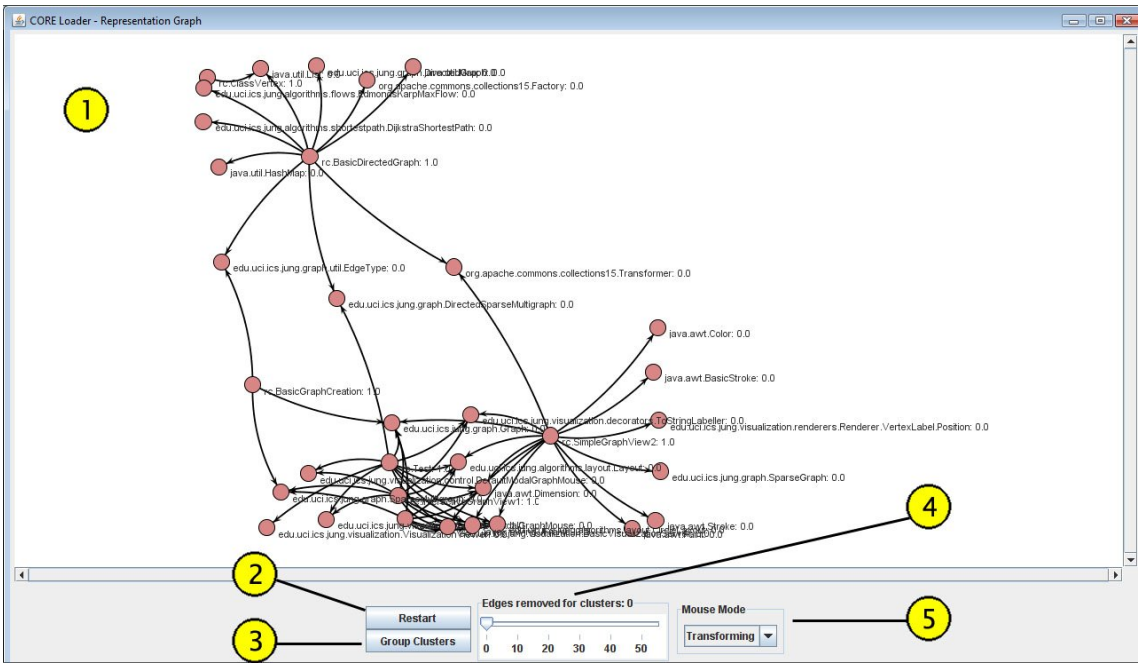

Figure 10. CORE Loader main screen

A brief explanation of CORE Loader main functionalities goes as follows.

1) Graph panel: The place where the graph is shown and user can interact with. Vertices labels are classes' names.
2) Restart: This button reloads the target system, cleaning the operations previously made;
3) Group Clusters: This functionality rearranges the vertices in group form.

4) Remove Edges for Clusters: The number of edges in the graph is show in a slider component and it is possible to remove those edges specifying the quantity. Edges with the lowest weights (loosely coupled) are removed first (painted in gray).
5) Mouse Mode: It corresponds to the kind of interaction the mouse at the moment. It can be either “Transforming” or “Picking”. In transforming mode is possible to change views without modifying the graph structure. In picking mode user can drag and drop selected vertices to another place in the graph panel.

As we increase the number of the vertices removed (the gray edges), we can note that some “islands” begin to appear; those islands represents classes strongly connected (highly coupled). Figure 11 shows the same target system with 24 edges removed.

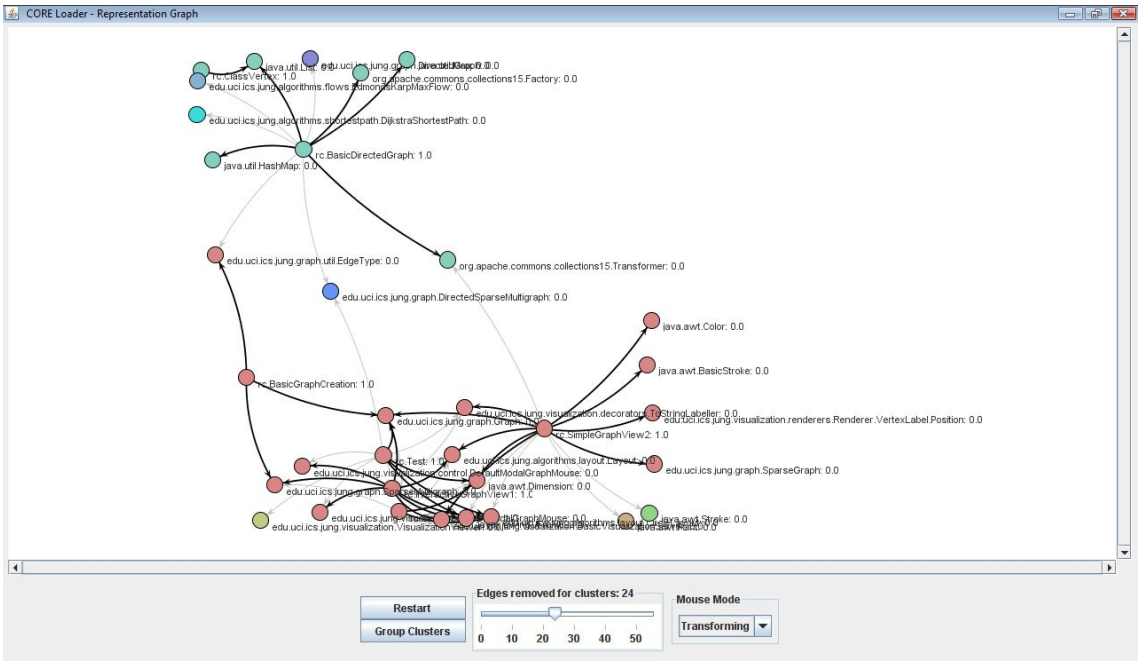

Figure 11. Edges removed from graph

Vertices with similar colors are considered to belong to the same cluster. Finally, we can see possible component candidates as the clusters formed, when we activate “Group clusters” button (Figure 12).

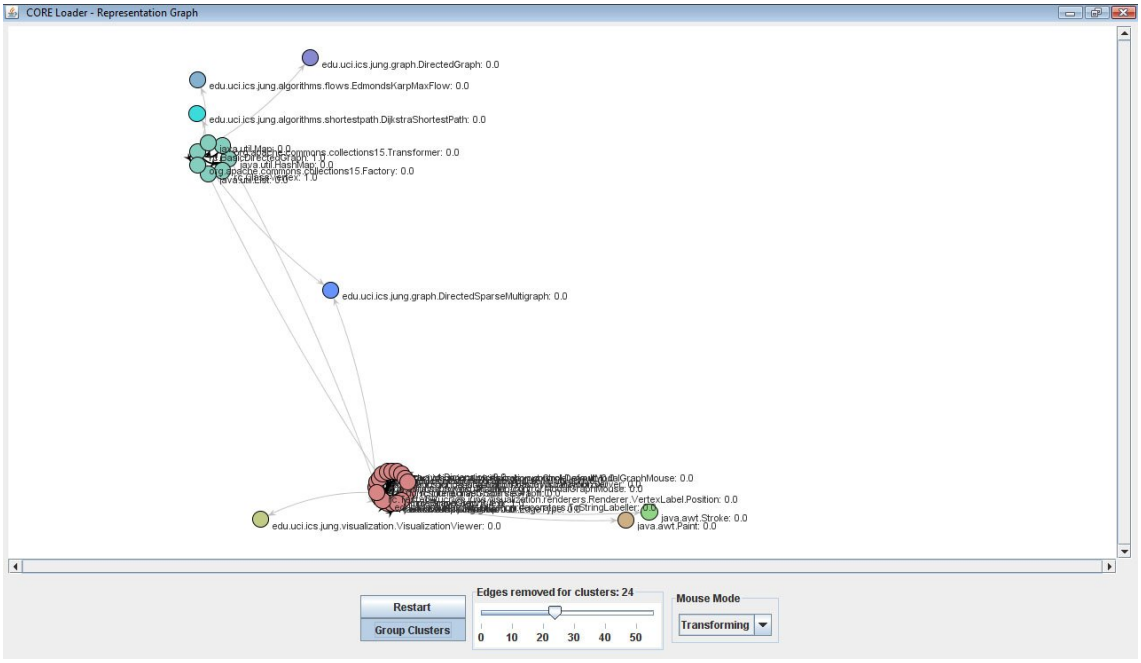

Figure 12. Cluster formation in CORE Loader

The clusters should be then, evaluated by a domain expert, packaged and eventually exported to a component repository. These functionalities are not covered in this version of CORE Loader yet.

4.3.2 Experimental Results

We have applied this initial version of CORE Loader in 15 different target Java projects. The preliminary results are exciting: about 55% of the components identified matched with our intuition. However, we observed some drawbacks:

- There was a high distortion in the granularity of the identified component; some could encapsulate more than one component, and others were too small to be a component. This reflects our coupling measure is not being enough criteria to our clustering algorithm;
- Much of the precision still depends on the number of edges removed for cluster. Unfortunately we cannot conclude that there is a relationship between number of edges removed and precision since not only the size but also the topology of the target systems vary dramatically; an experiment with target systems with similar topologies would give worthier contributions.

- As our purpose is to help engineers in finding reusable modules, we have some difficult in visualizing the labels nodes in large graphs, among other usability deficiencies.

We believe that as more techniques we add to the CORE Loader more aspects of the target system will be covered and hence, we will get more accurate results. Moreover, the clustering algorithm used in this experiment was too simple (section 4.3.1). We comment more on the limitations of this research in section 5.2.

4.3.3 Current Open Issues / Ideas

The current open issues include, but are not limited to:

Usability / Interaction improvements: As we are trying to help engineers to find reusable modules, it is essential that they have a clean, usable interface - not being an obstacle. Colors and shapes could be used to distinguish classes and edges aspects; Enable hiding selected information and so on.

Classify classes in layers: Would be useful if the classes could be at least roughly classified in graphical, business or data classes.

Matrix representation: As an alternative to the graph view, the tool should represent the code structure in a matrix form, facilitating the dependency view in a large system.

View and edit the source code: We could improve user interaction with the tool by enable him to view the source code related to a node in the graph. This would facilitate his comprehension about the system analyzed. It would be also interesting if he could edit and compile a class in the tool itself. This kind of interference would be optional and help the tool to get better results.

Generate metrics and reports: Metrics allow the engineers to define quality goals, measure them, and to monitor the accomplishment of these goals, and

can be used to measure different aspects such design and code. Thus, the definition of a set of metrics, and reports related, can be helpful in reverse engineering tasks.

Suggest libraries as components: In some situations, we found out that some libraries were intensively used by several projects and they were not in the component repository yet. It would be useful if those libraries could be suggested to be exported to the component repository as well.

These questions remains in evaluation and will be developed according to their priorities. We comment more on the future works in section 5.2.

4.4 Chapter Summary

This chapter presented the main aspects of the proposed tool. The requirements were defined and the architecture was showed with some implementation details. We:

- Specified the goals and requirements of the CORE Loader tool;
- Presented and described the CORE Loader architecture and its modules;
- Introduced some technologies and frameworks used in CORE Loader implementation;
- Talked about current open issues and evolution of CORE Loader.

5 Conclusions

We have a long roadmap towards an automatic component load tool. While it is too early to claim a major success, the results outlined in the previous section are encouraging enough to support the idea that using those heuristics for identification of reusable components is a valid approach. However, it is important to understand that even the most successful identification system will require human intervention when evaluating components for reusability. Steps are been taken at the present time to continue the development of CORE Loader.

5.1. Research Contributions

The research done in this work was not intended to be conclusive. Instead, our goal to make an indicative-exploratory study in the field was satisfactorily accomplished. We can outline the main contributions of this work as being:

- Extensive literature review on clustering methods for component identification and a comparison table;
- Proposal of a new approach combining existing methods to the modularization problem;
- Propose ranking as a heuristic that can be added in the process;
- Specification, design and implementation of a tool for component identification.

5.2. Limitations and Future Work

Due to time and resources constraints, this work can be seen as the first step towards the full vision of an efficient reverse engineering tool. As shown in the previous example, our process is applicable to an OO system and is expected to result in a high-quality component-based system. However, because the CORE Loader is still on its first version, the results obtained by using our method roughly matched what our intuition indicated. In the future, we plan to find more salient features of more effectively identifying components and apply them to our component identification approach.

We are also planning to introduce some other heuristics that will be able to capture more information about reusability when using classes, objects, inheritance, and polymorphism. The limitations of the current CORE Loader prototype is summarized in the section 4.3.2 Current Open Issues / Ideas.

A real-world test of the techniques presented in this paper remains for future work. It is likely that such a project would reveal new research problems-for example, it may be fruitful to investigate strategies other than the ones proposed in Section 4 for ensuring the non-existence of a better partition of the target system.

We have plans to use a larger set of systems to assess the performance of the system with respect to industrial size code. In addition, an in depth study is been conducted to verify the predictive power of the measures described previously.

5.3. Concluding Remarks

This work has shown an overview of heuristics to the problem of identifying modules in OO legacy code. The goal of our work is to design a tool to assist engineers in early stages of an asset repository implantation, in particular, with the task of identified possible reusable modules from existing programs in order to populate the base.

The relevance of this work is perceived when it is observed the current gaps in existing methods and tools for component identification: inappropriateness of techniques to the domain, poor user interaction, high domain-knowledge needed and lack of industrial tools are some of them.

Moreover, this work demonstrates that the identification of object-oriented legacy software components can be made significantly easier and more quantifiable which will aid greatly in promoting effective software reuse.

References

[SourceForge 2007] VA Software: SourceForge Product Introduction. Available on http://www.sourceforge.com/, Accessed in 12/12/2007.

[Krueger 1992] Krueger, C.W. Software Reuse, In: ACM Computing Surveys, Vol. 24, No. 02, June, 1992, pp. 131-183.

[McIlroy 1968] McIlroy, M. D. Mass Produced Software Components, In: NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

[Bauer 1993] Bauer D. A Reusable Parts Center, In: IBM Systems Journal, Vol. 32, No. 04, 1993, pp. 620-624.

[Griss 1995] Griss M. L. Making Software Reuse Work at Hewlett-Packard, In: IEEE Software, January, 1995.

[Sametinger 1997] Sametinger, J. Software Engineering with Reusable Components. Springer-Verlag, 1997.

[Garcia et al 2006] Garcia, V. C.; Lucrédio, D.; Lisboa, L. B.; Martins, A. C.; Almeida, E. S.; Fortes, R. P. M.; Meira, S. R. L Toward a Code Search Engine Based on the-State-of-Art and Practice,13th IEEE Asia Pacific Software Engineering Conference (APSEC),Component-Based Software Development Track, Bangalore, India, 2006.

[Almeida et al., 2007] E. S. Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. A. Burégio, L. M. Nascimento, D. Lucrédio, S. R. L. Meira, CRUiSE: Component Reuse in Software Engineering, 2007, pp. 202.

[Ezran el al., 2002] Ezran M.; Morisio, M.; Tully, C. Practical Software Reuse, Springer, 2002.

[Brown 1998] Brown, W., and K. Wallnau "The Current State of CBSE", IEEE Software, October 1998, pp37-46

[Souza 1998] D'Souza, D. F., and Alan C. Wills "Objects, Components, and Frameworks with UML : The Catalysis Approach" , ISBN 0-201-31012-0 Addison-Wesley, 1998.

[Kozaczynski 1998] W. Kozaczynski. Architecture framework for business components. 5th International Conference on Software Reuse, Computer Society Press. 1998, 300~307.

[ICSE 1998] http://www.sei.cmu.edu/cbs/icse98/summary.html . Accessed in 01/22/2008.

[Cheesman 2001] J. Cheesman and J. Daniels, UML Components: A Simple Process for Specifying Component-Based Software, Addison Wesley, 2001.

[Hasselbring 2002] W. Hasselbring. Component-Based Software Engineering. Handbook of Software Engineering and Knowledge Engineering, pp. 289-305, World Scientific Publishing, 2002.

[DePrince 2002] Wayne DePrince Jr. and Christine Hofmeister. Analyzing Commercial Component Models. WICSA, IFIP Conference Proceedings, Vol. 224, pp. 205-219, Kluwer, 2002.

[Emmerich 2001] Wolfgang Emmerich and Nima Kaveh. Component technologies: Java Beans, COM, CORBA, RMI, EJB and the CORBA Component Model. ESEC / SIGSOFT FSE, pp. 311-312, 2001.

[Voelter 2003] M. Voelter. A Taxonomy of Components. Journal of Object Technology, 2(4), pp. 119-125, 2003.

[Szyperski 1998] C. Szyperski. Component Software. Addison-Wesley, Harlow, England, 1998.

[Yacoub 1999] S. Yacoub and H. Ammar and Ali Mili. Characterizing a Software Component. 1999.

[Hall] Pat Hall, Educational Case Study –What is the model of an ideal component? Must it be an object? Third International Workshop on Component-Based Software.

[Broy et al., 1998] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal and C. Szyperski. What characterizes a (software) component?

Software - Concepts and Tools, 19(1), pp. 49-56, 1998. [Beneken 2003] Componentware – State of the Art 2003.

[Sametinger 1997] Sametinger, J. Software Engineering with Reusable Components. Springer-Verlag, 1997.

[Heineman & Councill 2001] Heineman, G. T.; Councill, W. T. Component-based Software Engineering: Putting the Pieces Together. Addison-Wesley, 2001.

[McIlroy 1968] McIlroy, M. D. Mass Produced Software Components, In: NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

[Gartner 1997] The Component Revolution, Tutorials, TU-850-25 1, A. Percy, August 97.

[Aniorte 2002] Seyler F. and Aniorté P. Cooperation of distributed components: From component model to software architecture. In Proceedings of the Workshop on Software Agents - Co-operation - Human Activity (SACHA 2002) - ITS Conference pages 25-33,Biarritz, France, June 5 2002.

[Chikofsky & Cross, 1990] Chikofsky, E. J.; Cross, J. H. Reverse engineering and design recovery: a Taxonomy. In: IEEE Software, Vol. 01, No. 07, January, 1990, pp. 13–17.

[McIlroy 1968] McIlroy, M. D. Mass Produced Software Components, In: NATO Software Engineering Conference Report, Garmisch, Germany, October, 1968, pp. 79-85.

[Lung 2004] Chung-Horng Lung and Marzia Zaman and Amit Nandi. Applications of clustering techniques to software partitioning, recovery and restructuring. The Journal of Systems and Software, 73(2), pp. 227-244, October 2004.

[Gamma et al., 1995] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software Development. Addison-Wesley Professional, 1995.

[Almeida et al., 2007] E. S. Almeida, A. Alvaro, V. C. Garcia, J. C. C. P. Mascena, V. A. A. Burégio, L. M. Nascimento, D. Lucrédio, S. R. L. Meira, CRUiSE: Component Reuse in Software Engineering, 2007, pp. 202.

[Caldiera and Basili 1991] Biggerstaff and Perlis, Software Reusability, ACM Press, 1989. Caldiera and Basili, “Identifying and Qualifying reusable software components” IEEE Computer, February, 1991.

[Koschke 1999] R. Koschke. An Incremental Semi-Automatic Method for Component Recovery. Working Conference on Reverse Engineering, p. 256-, 1999.

[Dunn and Knight 1993] M. F. Dunn and J. C. Knight. Automating the detection of reusable parts in existing software. Proceedings of the 15th International Conference on Software Engineering, pp. 381-390, IEEE Computer Society Press, April 1993.

[Lee et al., 2001] J. K. Lee, S. J. Seung, S. D. Kim, W. Hyun and D. H. Han. Component Identification Method with Coupling and Cohesion. APSEC, p. 79, IEEE Computer Society, 2001.

[Jain 2001] H. K. Jain, N. Chalimeda, N. Ivaturi and B. Reddy. Business Component Identification - A Formal Approach. EDOC, pp. 183-187, IEEE Computer Society, 2001.

[Chao 2005] M. Fan-Chao, Z. Den-Chen and Xu Xiao-Fei. Business Component Identification of Enterprise Information System: A hierarchical clustering method. ICEBE, pp. 473-480, IEEE Computer Society, 2005.

[Choi 2006] M. Choi and E. Cho. Component Identification Methods Applying Method Call Types between Classes. J. Inf. Sci. Eng, 22(2), pp. 247-267, 2006.

[Etzkorn 1997] L. H. Etzkorn. A metrics-based approach to the automated identification of object oriented reusable software components. Ph.D. dissertation. 1997.

[Girard and Koschke 1997] J.F. Girard and R. Koschke. Finding Components in a Hierarchy of Modules: a Step towards Architectural Understanding. ICSM, IEEE Press, 1997.

[Chiricota and Melançon 2003] Y. Chiricota, F. Jourdan and G. Melançon. Software Components Capture using Graph Clustering. HAL - CCSd - CNRS, February 28 2007.

[Jordan 2005] N. Sangal, E. Jordan, V. Sinha and D. Jackson. Using Dependency Models to Manage Complex Software Architecture. Proceedings of OOPSLA'05, pp. 167-176, 2005.

[Mitchell and Mancoridis 2006] B. S. Mitchell and S. Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. IEEE Transactions on Software Engineering, 32(3), pp. 193-208, 2006.

[Sahraoui et al. 1997] H. A. Sahraoui, W. L. Melo, H. Lounis and F. Dumont. Applying Concept Formation Methods to Object Identification in Procedural Code. ASE, pp. 210-218, 1997.

[Linding and Snelting 1997] C. Lindig and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proceedings of the 19th International Conference on Software Engineering, pp. 349-359, ACM Press, May 1997.

[Canfora 1999] G. Canfora, A. Cimitile, A. De Lucia and G. A. Di Lucca. A Case Study of Applying an Eclectic Approach to Identify Objects in Code . International Workshop on Program Comprehension, pp. 136-143, IEEE Computer Society Press, 1999.

[Siff and Reps 1999] M. Siff and T. Reps. Identifying Modules via Concept Analysis. Transactions on Software Engineering, 25(6), pp. 749-768, IEEE Press, November 1999.

[Snelting 2000] G. Snelting and F. Tip. Understanding Class Hierarchies Using Concept Analysis. ACM Trans. on Programming Languages and Systems, pp. 540-582, May 2000.

[Cimitile and Visaggio 1995] A. Cimitile and G. Visaggio. Software Salvaging and the Call Dominance Tree. Journal of Systems Software, 28:117-127, 1995.

[Wille 1981] R. Wille, "Restructuring Lattice Theory: An Approach Based on Hierarchies of Concepts." Ordered Sets, I. Rival, ed., pp. 445-470, NATO Advanced Study Inst., Sept. 1981.

[Abreu 2001] F. Abreu and M. Goulão. Coupling and Cohesion as Modularization Drivers: Are We Being Over-Persuaded?. CSMR, pp. 47-57, 2001.

[Washizaki 2005] H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. Sci. Comput. Program, Vol. 56, Number 1-2, pp. 99-116, 2005.

[Lee et al., 2005] E. Lee, W. Shin, B. Lee and C. Wu. Extracting Components from Object-Oriented System: A Transformational Approach . IEICE Transactions on Information and Systems, Vol. e88-d, Number 6, p. 1178, IEICE, July 05 2005.

[Godin et al., 1995] R. Godin, G. Mineau, R. Missaoui, M. St-Germain and N. Faraj, Applying Concept Formation Methods to software Reuse, International Journal of Knowledge Engineering and Sofiware Engineering, 5(1): 119-142, 1995.

[Gall 1998] H. Gall, R. Kosch, J. Weidl, Resolving uncertainties in object-oriented re-architecting of procedural code, in: Proc. 7th International Conference on Information Processing and Management of Uncertainty in Knowledge-Based Systems, 1998, pp. 726–732.

[Valasareddi 1998] R.R. Valasareddi and D.L. Carver, “A graph-based object identification process for procedural programs,” Proc. of the Fifth Working Conference on Reverse Engineering, October, 1998, IEEE Computer Society Press.

[Yassine 2004] Yassine, A. “An Introduction to Modeling and Analyzing Complex Product Development Processes Using the Design Structure Matrix (DSM) Method”, Quaderni di Management, no. 9, 2004, English version.

[Page et al., 1999] L. Page, S. Brin, R. Motwani and T. Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Number SIDL-WP-1999-0120, November 1999.

[Inoue et al., 2003] K. Inoue, R. Yokomori, H. Fujiwara, T. Yamamoto, M. Matsushita and S. Kusumoto. Component rank: relative significance rank for software component search. Proceedings of the 25th International Conference on Software Engineering (ICSE-03), pp. 14-24, IEEE Computer Society, May 3-10 2003.

[Mitchell 2002] B.S. Mitchell. A Heuristic Search Approach to Solving the Software Clustering Problem. Ph.D. dissertation, Drexel Univ., 2002.

[Tiobe 2008] http://www.tiobe.com/tpci.htm . Accessed in 01/22/2008.

[Mayobre 1991] Guillermo Mayobre. Using Code Reusability Analysis to Identify Reusable Components on a Domain Focussed Reuse Oriented Software Development Process.Proceedings of the Fourth Workshop on Institutionalizing Software Reuse, 1991.

[Abreu et al. 2000] F. B. Abreu, G. Pereira and P. Sousa. A Coupling-Guided Cluster Analysis Approach to Reengineer the Modularity of Object-Oriented Systems. CSMR, pp. 13-22, 2000.

[Lindig and Snelting 1997] C. Lindig and G. Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. Proceedings of the 19th International Conference on Software Engineering, pp. 349-359, ACM Press, May 1997.