Acknowledgements

I would like to thank my all Brazilian colleague, who gave me
a lot of support during the Master.

I thank Prof. Dr. Edson Scalabrin from Pontifícia Católica do Paraná – PUC - PR who
gave me the opportunity to participate in this project.

Special thanks to my mother and my sister that, even so far,
gave me incentive and support in my studies.

TABLE OF CONTENTS

LIST OF FIGURES IV

LIST OF TABLES V

ABSTRACT VI

1 INTRODUCTION 1

2 LITERATURE REVIEW 3

2.1 REUSABLE COMPONENT DEFINITIONS 4
2.2 TYPES OF COMPONENTS 5
2.3 VIEWS ON THE REUSE SOFTWARE PROCESS 7
2.4 EXISTING TECHNIQUES FOR DETECTING REUSABLE COMPONENTS 8
2.4.1 Identifying and Qualifying Reusable Software Components 9
2.4.2 An Expert System Approach 12
2.4.3 Transform Slicing 14
2.4.4 Common Aspects between techniques 15

3 MOTIVATIONS AND DIFFICULTIES ENCOUNTERED 17

4 A GUIDELINE TO DETECT REUSABLE COMPONENTS 19

4.1 THE PURPOSES OF THIS GUIDELINE 19
4.2 TYPES OF COMPONENTS CONSIDERED 19
4.3 THE SET OF STEPS 20
4.3.1 Step 1: Determine the types of reusable components to search for 20
4.3.2 Step 2. Determine the application domain to identify reusable components 21
4.3.3 Step 3. Analyse the Source Code to Detect Redundancy 21
4.3.4 Step 4. Count the Number of Static Calls 22
4.3.5 Step 5: Analyse the Source Code to Detect Dependencies 23
4.3.6 Step 6: Measure the Depth of Inheritance of a Component 24
4.3.7 Step 7: Determine the potential use of the component 24
4.4 OUTLINED METRICS 24
4.4.1 Halstead Metrics 24
4.4.2 McCabe's Cyclomatic number 26
4.4.3 Lack of Cohesion of Methods (LCOM) 27
4.4.4 Coupling between object classes (CBO) 29
4.5 CONCLUSIONS 30

5 APPLYING THE GUIDELINE IN A CASE STUDY 31

5.1 THE CASE STUDY: A BANKING APPLICATION 31
5.1.1 Describing the Application 32
5.1.2 The Architecture of the Application 33
5.2 CHECKING REDUNDANCY 34
5.3 CHECKING DEPENDENCY 38
5.4 THE RESULTS OBTAINED 39
5.5 CONCLUSION 39

6 CONCLUSIONS 41

6.1 CONTRIBUTIONS 41
6.2 FUTURE WORK 41

7 REFERENCES 43

APPENDIX A : UML MODEL AND TRANSLATION TERMS 45

APPENDIX B: SOURCE CODE.. 51

List of Figures

Figure 1: Views of Reuse Process 7
Figure 2: Reuse Process Model 10
Figure 3: Expert System model 13
Figure 4 : Banking Application Context 33
Figure 5: Common Object Request Broker Architecture 33
Figure 6 : Demonstrating a part of a reengineering process 36

List of Tables

Table 1: Methods of Class Account 34
Table 2: Methods of Class Agency 34
Table 3: Methods of Class Bank 35
Table 4: Methods of Class Client 35
Table 5: Methods of Class Saving Account 35
Table 6: Metric LCOM per Classes........ 39

Abstract

The work described in this thesis addresses the field of *software reuse*. Software reuse is widely considered as a way to increase the productivity in software development. Moreover it can bring more quality and reliability to the resultant software systems. Identifying, extracting and reengineering software components from legacy code is a promising cost-effective way to support reuse. Considering this assumption, this thesis deals with the composition of a Guideline that can direct the identification of reusable software components in Object Oriented (OO) legacy system. In order to compose this Guideline some existing techniques to detect reusable components were evaluated. From them the main aspects concerning object-oriented concepts are taken into account. In addition, the basic concerns of object-oriented paradigm are considered since they hold in themselves the goal to tackle reuse. After showing the Guideline composition process and the principles covered in its context, some of the directives stated are applied in a case study to demonstrate their use in practice.

1 Introduction

Reuse has been considered an important solution to many of the problems in software development. It has been claimed to be essential in improving productivity and quality of software development with significant benefits reported by many organizations.

Although the software community does not agree on what software components are exactly, it is recognised that they are the basic unit to practice the reuse. The increased commercial offering of embeddable software components, the standardization of basic software environments and the popularization of Internet have resulted in a new situation for reusability: there are many more accessible reusable components that can achieve a large usage.

Given the high interest in reuse and motivation to the use of available software components, the development software environment is embracing the identification, evaluation and selection of reusable components as important processes, with a high potential impact on the product and project objectives. Since these activities are typically not well defined, each project finds its own approach to perform them often under schedule pressure and without experiences based on previous developments.

This lack of a solid basis for the reuse process causes that each time it is performed it needs to be reinvented. Therefore, for the moment consistency is not yet guaranteed. When a planning for this area can be defined it will be easier to the organizations to follow previous experience gaining from the use of validated methods.

In order to provide a support to facilitate this important identification process of candidate's reusable components in existing software, this work suggests a Guideline to help in the detection of reusable components in OO legacy code.

This Guideline provides a basis for evaluating and identifying candidate reusable parts for software development. It offers oneself a series of steps and suggestions to apply in source code to identify with more facility the possibilities of reuse in the system approached.

The idea in this work is to take advantage of the few techniques encountered in this context. To achieve this purpose the fundamental aspects of each one of them will be considered and combined with new statements discovered during this research.

To validate the Guideline it was necessary to apply it in an example. The results of this experience are explained in the end of this document in order to make clear the Guideline suggestions.

This thesis is organised as follows:
The chapter 2 covers a literature review in the area of reusable components. Chapter 3 presents the motivations and difficulties encountered to compose the Guideline in this unexplored area. In Chapter 4 the Guideline is described considering its purposes, the type of reusable components covered and the steps suggested. The chapter 5 specifically

focuses the evaluation of the Guideline applied in a specific example in order to validate it. The conclusions and further directions are finally given in chapter 6.

2 Literature Review

Rarely software is built completely from scratch [SJ97]. Software reuse is the process of creating software systems from existing software rather than building from scratch. In this way the reuse of software components can considerably reduce the development effort and improve the quality and the reliability of new software systems.

Existing software is widely considered to be the main source for the extraction of reusable assets. To fit new requirements, existing software (documentation, design documents, source code, etc) are adapted composing a great extent. Nowadays, identifying, extracting and reengineering software components from existing system is a promising way to create these reusable assets.

The most basic of the reuse strategies is to develop an application system from scratch only as a last resort. To support this idea it is necessary to find and select parts of source code to compose reusable components that can be used in the development of the various system project deliverables, and this process is not an easy task.

In [JG97] the author affirms that there is a lack of components to reuse. This aspect is due to a host of obstacles: failure to select and strengthen components for reuse in the first place; lack of techniques to identify, classify and package components.

Since object-oriented emerged, organizations have accumulated thousands of lines of code. The concern of developing and identifying reusable components was not apparent until very late and most of the applications were developed following a conventional approach. Instead of recreating component similar to component already existing, and considering the high cost of development, many organizations have realized that reuse parts of applications could be a great advantage.

Building application systems from reusable components is based on the assumption that reusable components exist somewhere, they are reasonably easy to find and understand, and they are of good quality [JH98]. This assumption intends to give motivations to reuse process, but according to what has been explained components are not so easy to deal with them.

In this context of reusable components, there are still few techniques to the identification of reusable components in a repository as well as in legacy system. The techniques encountered will be described in order to demonstrate existing approaches in this direction. To get a clear understanding about the techniques it is necessary before to conceptualize reusable components defining their role, and show the two existing views in the reuse process.

2.1 Reusable Component definitions

The best way to start talking about reusable components is giving a definition for the term reusable component. As the software community does not yet agree on what a component is exactly, there are several definitions for the term software component. Components are well established in all other engineering disciplines, but until recently were unsuccessful in the world of the software.

Many papers and books [BM94, EMT98, JGJ97, and NJ96] try to give a definition for components but it can be realise that each author focuses his definition in the research or the domain application in which the component was used from the usage context.

A general definition that can be given is stated in [JGJ97]:

"Software Component is any element of software life cycle that can potentially be reuse in several contexts".

From this definition one can see that Component and Reusable Component are superposed terms. In the reuse context, according [Mc97], it can be distinguished two different approaches that help us in defining components. Components can be seen as some part of software that is identifiable and reusable. Therefore functions and classes are examples of such components. On the other hand, components can also be seen as the higher level of abstraction, such as patterns, frameworks and specifications.

This work is in the context of software reuse that it focuses the first approach. The intention is to identify in Objet Oriented legacy systems parts potentially reusable.

To be more specific, for the author [SC97], components have the following characteristic properties:
- A component is a unit of independent deployment;
- A component is a unit of third-party composition;
- A component has no persistent state.

These characteristics have various implications but one important aspect to point out is that a component needs to be independent from its environment and from other components. To get this independence, components basically must have <u>an interface</u>; <u>encapsulate internal details</u> and must be <u>documented separately</u>. These three aspects are very important to compose good reusable components.

In [SJ97] the author gives another important definition to consider in order to understand the focus of this work:

"Reusable software components are self-contained, clearly identifiable artefacts that describe and/or perform specific functions and have clear interfaces, appropriate documentation and a defined reuse status".

Searching in software literature it is possible to explain in more details the main aspects of the previous definition:

- **Self-containedness**
  Components must be *self-contained*. In this sense, it must be considered the concepts of packages or modules, that is, the way of programming languages deal with the components. For example, one module can be used as interface for a set of modules which are the entity for reuse, then this component can be reused, or one process can be used as the interface for a set of processes which can even run on different machines.
  For the author [SJ97] it may not always be practical to integrate all parts with a component in order to make it self-contained, but the dependencies have to be clearly documented.

- **Identification**
  Components have to be *clearly identifiable*, through some features such as interface, cohesion and they must be independent and executed with a specific functionality.

- **Functionality**
  Components *describe and/or perform specific functions*, i.e., components must have clearly specified functionality, that can be described through their specification and documentation.

- **Interfaces**
  Components must have clear interfaces and hide details that are not needed for reuse. An interface determines how a component can be reused and interconnected with other components [BM94]. It defines an operation or a set of operations, usually related, defines a service that is available for a component.
  Interfaces of components are crucial for their composition. Components have three different types of interfaces, as stated by [SC97]: a programming interface, a user interface and/or data interface. For reuse all three interfaces are important, although programming interface is certainly the most important one.

- **Documentation**
  It is considered indispensable for reuse. Enough information must be provided in order to allow a component to be reused. This aspect is a problem frequently encountered in majority of the developments.

- **Reuse status**
  Components must be maintained to support systematic reuse [Mc94].
  The reuse status contains information about who maintain them, who can be contacted in case of problems, etc. It becomes a crucial information in the companies.

2.2 Types of Components

Since there are many definitions to reusable components considering different environments and levels of granularity, several types of reusable components have been provided in the literature.

Examples of Types of Reusable Components [Mc97] include software parts or components at varying levels of abstraction and of different sizes what can be understood as the component granularity, as well as documentation deliverables such as:

- Application package;
- Subsystems;
- Data type definition;
- Designs;
- Specifications;
- Code;
- Documentation;
- Test case and test data;

In this work just reusable components in the object-oriented context will be considered. Although the object-oriented approach makes reuse more feasible than with other software development methods because of mechanisms such as encapsulation and inheritance, software reuse is not guaranteed through the use of object-oriented concepts.

Examples of Types of Reusable Components in object-oriented context specifically include:

- Application Framework;
- Use Cases;
- Object Classes;
- Analysis and Design Models;
- Methods;
- Test packages;
- Documentation;
- System Architectures;

2.3 Views on the Reuse Software Process

Some organizations have implemented systematic reuse programs, which have resulted in in-house libraries of reusable components. Other organizations have supported their reuse with own techniques and tools to recover components. Consequently, these organizations are spending much time in recovering reusable components since the choice of the appropriate components depend on several aspects, such as experience, cost and techniques.

Considering these situations there are two faces or sides of reuse, according to [Mc97], that must be incorporated into the development process to support the practice of software reuse:

Consumer Reuse: Activities for using reusable components in the creation of new software systems – building ***with*** reuse view

Producer Reuse: Activities for creating, acquiring or reengineering reusable components– building ***for*** reuse view

Figure 1 shows each view addresses reuse.

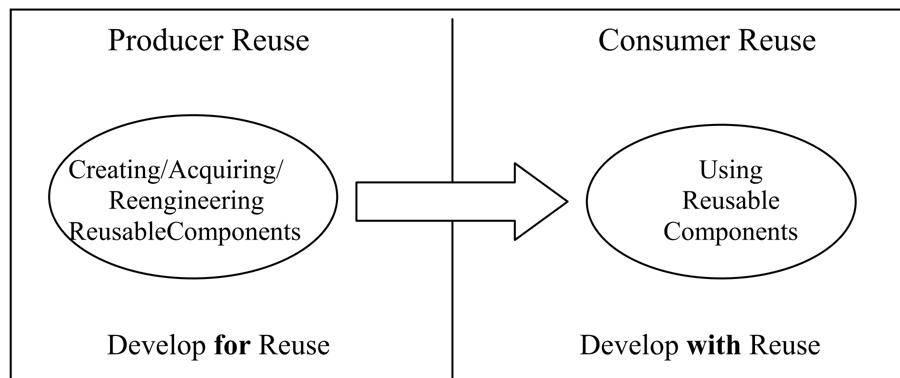


Figure 1: Views of Reuse Process

- **Software development with reuse**

Consumer Reuse is concerned with using reusable software parts to built new systems. Software development with reuse is an approach to development that tries to maximize the reuse of existing components. An obvious advantage of this approach is that overall development cost should be reduced. Fewer software components need be specified, designed, implemented and validated. However, it is difficult to quantify what the cost reductions might be.

In some cases, development costs may not be significantly reduced. The cost of component adaptation might occasionally cost as much as the original component development.
The process of development with reuse is usually developing a system completes a high-level design and specifications of the components of that design. These specifications are used to find components to reuse.

- **Software development *for* reuse**

Producer view is concerned with creating, acquiring and reengineering reusable software parts. A common misconception is that these components are available in existing systems being able to be easily identified. Even the components being created as part of an application system they are unlikely to be immediately reused. These components are geared towards the requirements of the system in which they are originally included. To be effectively reused, they have to be generalized satisfying a wider range of requirements.

In this context of development for reuse is more productivity to try to identify generalized components than to develop components from scratch.

According to the two views of the Reuse Process, it makes clear to affirm that the scope of this work collides with the Producer view.

2.4 Existing Techniques for detecting reusable components

The development of conventional software is supported by many consolidated well-defined techniques, but Reuse is a different paradigm that does not have yet clear techniques that detail of how it can be practice.

There are few techniques that deal with identification and extraction of reusable components. They work in different ways, but contribute to the same objective: the reusability.

The idea is to take advantage of these techniques in order to compose a Guideline, which can help in the detection of components in legacy code. It is important to know how the different steps of each technique interact and how they fit into the development process of detection as a whole.

The act of extracting components is considered a part of a process called Reuse Reengineering. However to consider this action of extraction it is necessary to get techniques that can facilitate this identification of components and after that, extract them.

2.4.1 Identifying and Qualifying Reusable Software Components

This technique can be viewed as a way to reuse experience along the development of software object products.

The basic process of the Identification and Qualification technique [CB91] concerns in how to analyse existing components and to identify ones suitable for reuse. After they are identified, parts could be extracted, packaged in a way appropriate for reuse, and stored in a component repository.

It is very important to emphasize that this technique encompasses an aspect related to the development of a catalog of components, i.e. construct a repository of reusable entities. This repository can be searched every time a component is needed within the software development process. Therefore this environment is not exactly the same considered in my approach. Anyway, the core ideas of identification can be helpful in the composition of the Guideline.

This technique is based on the partition of the traditional life cycle in two parts:

- the project, delivers software systems;
- the factory, supplies reusable software objects to the project;

The factory part concerns are the extraction and packaging of reusable components. Besides it also works with a detailed knowledge of the application domain from which a component is extracted.

This technique to identification and qualification of reusable software component is based on software models and metrics. The interesting way in which these two aspects are treated by the technique was fundamental to take advantage of it.

There are two major steps involved:

- The identification process uses software metrics to search for candidate reusable components, taking into account the large volume of source code;

- and the second step is the “qualification”: the automatically extracted candidates are analyzed more carefully in the context of the semantics of the source application.

The models and metrics allow a feedback and improvement that make the identification parts and extraction process fit in a variety of environments.

The Reuse Framework and Organization

To achieve its purposes the technique suggests an organization framework that demonstrate the project-specific activities and reuse-packaging activities, as showed in figure 2:

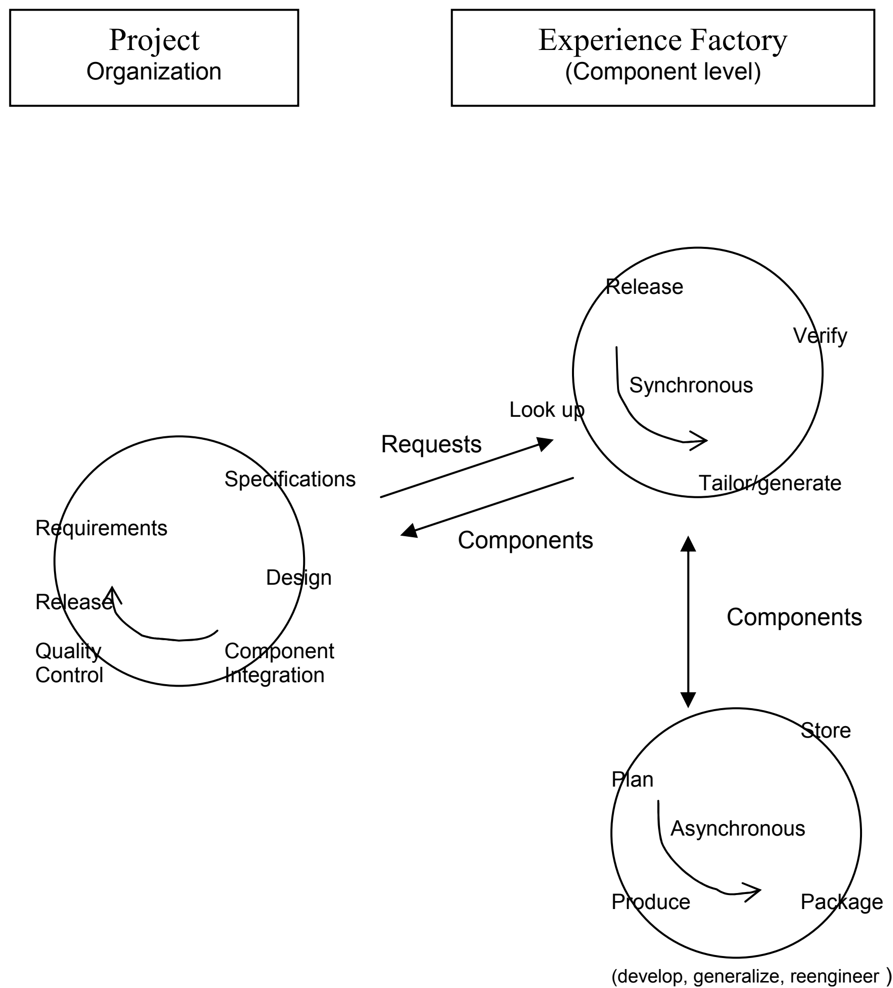


Figure 2: Reuse Process Model

The framework defines two separate organizations: a Project Organization and an Experience Factory.

The project organization develops the product, taking advantage of all sorts of package experience from prior and even current developments. In turn, the project offers its own experience to be packaged for other projects. Within the Experience Factory on the other hand, an organization called component factory develops and packages software

components. It supplies code components to the project on demand, and creates and maintains a repository for future reuse.

As a subdivision of the experience factory, the experience that the component factory manipulates is programming and application experience as present in programs and their documentation. As a result the component factory understands the project context and can deliver components that fit.

The component factory has two sides: it satisfies component requests from the project organization, and it also continuously prepares itself for answering those requests. This represents in fact mixing synchronous and asynchronous behaviour. These activities deal with the reception request and search in the catalog of the available components.

Considering specifically the scope of this work in which it is not approached the repository aspect, the synchronous and asynchronous activities will not be described in details.

- **Extracting Components**

The component factory analyses existing programs in two phases. First, some candidates are chosen and packed for probable independent later reuse. Next, human intervention is required: an engineer with knowledge of the application domain where the component was developed analyses each component to determine the service it can provide. Afterwards, components are stored in the repository, with all information that has been obtained about them.

To achieve the goal of the first phase it is done fully automated using of a tool called Care System. This tool is designed to support this activity of code examination to choose some candidate to package.

The necessary human intervention in the second phase is the main reason for splitting the process in two steps. The first phase reduces the amount of expensive human analysis needed in the second phase by limiting analysis to components that really look worth considering.

Independent program units are automatically extracted and measured according to observable properties related to their potential for reuse. An important part of this process is the measurement of the component potential reuse. To measure this aspect, a technique proposes a family of such measures that can be applied to their reusability attributes model.

The use of a quantitative model for identification of component and a qualitative, partially subjective model for their qualification, provide a continuous improvement of both models using feedback from their application. The reusability attributes model is the key to automating the first phase.

- **Component Identification**

The reusability attributes model for identifying candidate reusable component attempts to characterize the attributes directly through the measures of an attribute, or indirectly through the measures of evidence of an attribute's existence.

The technique establishes a set of acceptable values for each of the metrics. These values can be either simple ranges of values or more sophisticated relationships among different metrics.

The basic reusability attributes model reflects component reusability using the following metrics mentioned. The explanation in details about its formulas can be found in the chapter 4.

Volume and **Regularity**: measured using the Halstead Software Science Indicators. These values are measured through the number of operators and operands used in a program and seeing how well we can predict its length based on some regularity assumptions.

Cyclomatic Complexity: measured using the McCabe measure defined as the Cyclomatic number of the control-flow graph of the program.

Reuse Frequency: measured comparing the number of static calls addressed to a component with the number of calls addressed to a class of components that we assume are reusable.

The calculated values for each of this metrics mean:

- For Volume and Cyclomatic Complexity: it is necessary to establish an upper and a lower limit. In the case of volume, if the component is too small, the combined costs of extraction, retrieval and integration exceed its value. On the other hand if the component is too large, there's a bigger chance of errors in the process what can lead to a low quality component.

- For Regularity: it is calculated a closeness of an estimate by looking for a value close to 1.

- For Reuse Frequency: it is assumed that the program uses some naming convention to be sure that a component with a different name also represents a different functionality. In this case it is considered only a lower limit.

2.4.2 An Expert System Approach

An Expert System [DK93] is another approach that deals with the detection of reusable components. This technique tries to make decisions with a high degree of reliability by identifying design rules that are known to be supportive of reuse.

Such as Identification and Qualification technique, Expert system approach considers the concept of reusable library as a repository to store the components. It is important to emphasize again that this aspect of 'storage' is not considered in the proposed Guideline.

An expert system simulates the behaviour of a human expert. So, having in mind what a human expert would do when searching code for finding reusable components, two things certainly need to be looked for:
- Knowledge of the domain from which the system is being examined comes from;
- and the knowledge of the domain in which the component will possibly be used.

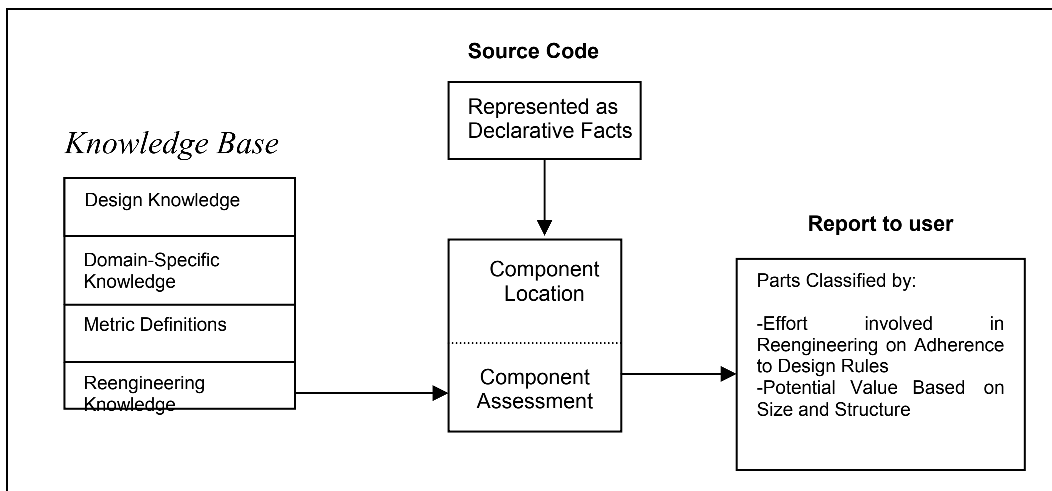


Figure 3: Expert System model

The knowledge is restricted to design knowledge at all levels that are related to reusability. It is supplemented by metrics such as the level of coupling between a set of programs subsets in order to characterize reusable parts.
The Expert System model is depicted in figure 3.
All of this knowledge is captured in different set of logic rules (like in Prolog). These rules can be classified as:

- **Rules that express candidate reuse structures.**
  The rules consider that the developer uses techniques as data abstraction when building a system.

- **Rules that are derived from the application domain.**
  To enhance the searching process as to locate parts likely to be relevant to a spectrum of systems across the problem domain, the rules can be changed as needed.

- **Rules that express different style characteristics that influence reengineering.**
  The purpose is simply to enhance the selection process allowing that a candidate part can be made ready to be put into a repository.

Moreover based in these rules, the Knowledge Base permits the system to identify potentially reusable parts in three ways:

- **By identifying functions that are invoked multiple times**.
  The multiple invoked functions are denoted in logical facts that represent a call graph. The set of Prolog facts denoting the graph would be generated during the program analysis phase.

- **By identifying functions that are loosely coupled.**
  Coupling means how tightly or loosely bound of a set of program subsets with each other is related. A function that is loosely coupled is not easily treated because it must be considered different types of coupling, as Data coupling, Common Coupling, External Coupling and Control Coupling.

- **By identifying functions and global data elements that can be grouped to form abstract data types.**

The activities of this technique are supported by a toolset called Code Miner that assists the programmer in identifying parts of existing software putting these candidates in a reuse library.

This toolset written in C has a Prolog interpreter and an interactive front-end. The Prolog interpreter provides the inference engine in which reusable component candidates are identified.

It is clear in this approach that only the identification part is automated. Human intervention will always be needed for selecting among those candidates which ones are really reusable assets.

2.4.3 Transform Slicing

The transform slicing approach [LV96] aims at extracting reusable functions from ill-structured programs. Transform Slicing is an extension of Program Slicing, one that includes statements, which contributes to transform a set of input variables into a set of output variables.

Program Slicing belongs to a family of program decomposition techniques based on selecting statements that are relevant for a computation, even if they are not grouped together in the program. Transform Slicing includes statements, which contribute to transform a set of input variables into a set of output variables.

To be able to use Transform Slicing two questions must be answered:
- How can we get a list of expected functions to be recovered together with input and output data?
- How will we cope with the difficulty of finding the last statement of an expected function, being the initial statements in the slicing criterion?

Transform Slicing requires the availability of knowledge about the application and programming domain to be able to get in some way a list of expected functions. Domain knowledge suggests the simple tasks being done in the system and these tasks are clearly defined at least in terms of input and output variables.

For data-oriented applications, the reverse engineering process should include a data recovery phase, to be able to produce a data model of the application system. This way, the initial domain representation will be expanded, so the representation evolves to an application data model. At the end of the data recovery phase, the dictionary with all the data will contain the description of the variables and the mapping between the model and the source code.

A further step in getting some expected functions is combining the information in the data dictionary with the functions found in static sources or sources suggested by dynamic sources (like domain experts, programmers…). The result of this should be a list of expected function specifications with a function name, a description of the function, an input and output parameter list.

2.4.4 Common Aspects between techniques

The techniques outlined above have some steps well defined and the continuous improvement of the used metrics model can be considered satisfactory.

All the techniques studying the dependencies among software elements at code level and a determination can be made of the reusability of those elements in other contexts. The theoretical reusability of a component of software is defined as the amount of dependence that exists between that component and other software components as it will see later.

One of the goals of the reuse process is to identify and extract the essential functionality from a program and this extracted essence is not dependent on external declarations, information, or other knowledge. Transformations are needed to derive such components from existing software systems since inter-component dependencies arise naturally from the customary design composition and implementation process used for software development.

Looking at the source code, the technique Identification and Qualification help to identify routines or units that satisfy the metric values typical of components with a high frequency of reuse. The metrics is represented by means of statistical values and they have to be in an allowed interval. The metrics used in this technique will be explained in section 4.3 (The Composition Process) in which the steps to be followed in the Guideline are determined.

The expert system approach is also based on some predefined characteristics of components known to be reusable that are then expressed in logic rules. The metrics are represented decoratively, because a declarative language is used to express the metrics.

The major advantage of the Slicing approach is that they are completely programming language independent because they are based on data flow analysis.

Finally, in order to compose the Guideline the main aspects of each technique that support the identification of potential reusable components are considered. Although all the techniques have well defined steps, it is also possible to conclude that an extraction technique without human intervention is still non-existing.

According [MR97] “A software component is reusable if the effort required to reuse it is remarkably smaller than the effort required to implement a component with the same functions”. Certainly these techniques described above help us in this effort.

3 Motivations and Difficulties encountered

This chapter presents the motivations and difficulties that are encountered to compose this Guideline in the context of software reuse. Several aspects encourage the composition of the Guideline, however some other aspects make difficult the process, considering mainly the few explored area of identification of components by software community.

Many motivations are given to justify the importance of the Guideline. First of all, as stated before, the simplest reason is that does not exist clear planning and deep research in this specific context of reusable component identification what stimulates the approach of this work.

Reuse with reusable components is becoming more and more important in a variety of aspects of software engineering. Recognition of the fact that many software systems contain many similar or even identical components that are developed from scratch over and over again has led to efforts to reuse existing components.

Structuring a system from independent components rather than creating reusable components as an independent activity has several advantages:

- It is easier to distribute the components among various projects to allow parallel development;
- Maintenance is more facilitated when clear interfaces have been designed for the components, because changes can be made locally without having unknown effects on the whole system;
- If components'interrelations are clearly documented and kept to a minimum, it becomes easier to exchange components and incorporate new ones into a system ;
- The resulting components are guaranteed to be relevant to the application area; and
- The cost is low and controllable.

The development of a Guideline to identify components is stimulated in order to give support to achieve these advantages. This Guideline concerns a planning of steps to guide and facilitate the identification of candidate reusable parts in source code.

If the time to search for reusable components is too long in the system builder's point of view, he will opt for building the component from scratch rather than reusing an existing component in spite of how well the reusable components fit the current needs. To facilitate this search, the Guideline must assure that the system approached will be examined following directives based in a planning and therefore it is easier to know the level of reusability of this code.

In contrast to the motivations some difficulties are also showed.

Since the process to compose the Guideline is done taking also advantage of existing approaches, its composition is restricted. After making a research in the literature of this area, it is possible to conclude that there is a lack of techniques to identify reusable components and the existing ones do not approach clearly many steps.

In general the first problem in reusing software arises from the nature of the objects to be reused. The concept for general reuse is simple: use the same object more than once. Although with software it is difficult to define what an object is independently of its context. Normally, the objects are not independent of the context and this aspect is fundamental to consider the idea of component.

In [Mc97] Reusability is defined as *the extent to which a software component can be used with or without changes in multiple software systems, versions or implementations*. The identification of the right reusable parts in the legacy system is often no-trivial task and requires careful consideration of multiple criteria and careful balancing between application requirements, technical characteristics and financial issues. However, the problems and issues associated with the identification of suitable reusable components have rarely addressed in the reuse community.

Some general criteria have been proposed by Object Oriented paradigm to help in the search of potential reusable components. Furthermore, most of the reusable component literature does not seem to emphasize the sensitivity of such criteria to each situation.

Considering the main objective of this guideline is to guide the process of identification of reusable components to be reused obtained from OO source code as existing legacy systems, various levels of difficulties can be outlined:

- Application development is normally designed for special requirement, not as a general abstraction. The goal is to have highly reusable components available in many reuse scenarios, not only for projects developed on/with a certain operating system and/or programming language.

- Generalize the directives in the Guideline that satisfy a wider range of requirements in different contexts is not always possible. Application contexts are diverse and each one holds its own specific features according to the domain.

- Lack of documentation to meet the information needs of potential and actual reusers.

- Useful abstractions are usually created by programmers with an obsession for simplicity and solve problems, who are willing to rewrite code several times to produce easy-to-understand and easy-to-specialize classes.

4 A Guideline to Detect Reusable Components

This chapter will describe properly the essence of the Guideline, its purposes, the type of component considered among the several types existing and the process of its composition.

4.1 The purposes of this Guideline

What is a Guideline? What are its purposes in the specific reuse context?

In a generic way, a definition to Guideline is done in [JG97]: *adapted procedures to be followed when implementing certain actions*. More specifically in the context of reuse, according to [AK98], Guideline can be viewed as *a manual application of the principles and techniques of generic transformation and applications*.

This Guideline tries to reveal several interesting and intuitively reasonable directives related to the identification of reusable components in OO legacy system. It is important to emphasize that the main purpose of a Guideline is not to prove the directives offered, but just to guide users to achieve the proposed goals.

Based in different sort of criteria, the Guideline composition is done combining several aspects of the techniques previous outlined (see section 2.4) and adding some statements. Before showing the Guideline composition process, the type of component considered is explained.

4.2 Types of Components Considered

Through the concepts of encapsulation, inheritance, polymorphism and data abstraction, object-oriented encourages software reuse in a number of ways. Class definition provides the abstraction, modularity and information hiding necessary to be considered in the essence of reusable components. In addition methods represent the behaviour of the classes through which system functionalities are expressed.

According to the types of components explained in section 2.2 and to the ideas stated in the previous paragraph, the type of component considered in this Guideline are class and method. A Class is the main building block in object oriented systems being able to represent strongly by itself some semantics. It determines the structure and behaviour of a group of objects expressing therefore the core concept in the object-oriented context. In addition, the possibility to specialize classes by inheritance without the need to modify their source code brings several advantages to reuse.

There are many other features encompassed by classes that allow them to be viewed as components:

- Classes have names;
- Classes may realize a set of interfaces;
- Classes may participate in dependency, generalization, and association relationships;
- Classes may be nested;

- Classes may have instances;
- Classes may be participants in interactions.

Methods in turn represent the implementation of services that can be requested from any object of the class to affect behaviour. Considering this, a method by itself can be reused since the service encompassed is essential to the system functionality.

Once components can be considered in different levels of granularity, in this Guideline a class or a group of classes, a method or a group of methods can be identified as reusable components since they implement common semantic or functionality.

4.3 The Set of Steps

In this section the steps suggested by the Guideline in order to facilitate the search and identification of reusable components in OO source code will be explained.
These steps determine the directives to be followed by guideline users encompassing metrics, rules and characteristics to be observed in the source code.

4.3.1 Step 1: Determine the types of reusable components to search for

The first consideration that needs to be taken into account in order to identify components is to determine the type of components to be searched. The component type can vary according to the level in which the work is performed as can be seen in the following examples of components in each level:

- Analysis level: use cases, analysis class model, class, etc.
- Design level: design class model, application framework, logic structure, class, etc.
- Implementation level: class, methods, etc.

Through these examples it can be realized that even defining the level in which the identification is to be proceeded, types of components can be still diverse.

Therefore, it is really important not only to establish the level in which to work but also in a specific level to be precise about the types of components to look for. Directives are not always generic and the lack of a clear component definition can lead to the risk of don't achieving the expected results.

According to these statements this Guideline focus on the identification of classes and group of classes, or methods and group of methods as components considering the level of source code to the search. The choice for these types of components is justified in section 4.2. Before achieving the core of the Guideline, step 2 will present generic directive dealing with the application scope.

4.3.2 Step 2. Determine the application domain to identify reusable components

Determine the application domain to which the component belongs is a fundamental step. The application domain establishes the semantics of the components that need to match to its requirements and specific features. Finding generic components that can be applied in any application domain is a next step after identifying components for one domain. This last assumption reflects the goal of this Guideline.

Therefore a strong knowledge of the application domain becomes necessary in order to determine their commonalties (similarities) and variabilities (differences). When the domain is studied, it usually is analysed from two perspectives: (1) aspects that tend to change from system to system (that are the differences or variabilities) and (2) aspects that remain constant in any system of the domain (that are the similarities or commonalities). This evaluation will lead to the identification of the main application domain functionalities that can be covered by predefined components.

Information describing existing systems in the domain approached is fundamental to delimit and understand it. There exist several sources of system information: documentation, requirements, analysis and design models, and source code. Analyse this information becomes essential to determine the common functionalities that are general enough and abstract enough to be reused.

The knowledge of the application domain is also a principle supported by the Expert System Approach explained in section 2.3.2. As the Expert system considers, the interaction with an expert in the application domain is also essential in order to get the knowledge.

In the following steps other specific directives to support the identification of essential functionalities in the application domain are described:

- considering only the system approached, in Step 3 it is stated how functionalities can be discovered through the identification of redundancy in the source code;
- in Step 4 the number of static calls addressed to a component is also suggested as another way to achieve this purpose.

4.3.3 Step 3. Analyse the Source Code to Detect Redundancy

To detect redundant parts in the source code the technique Redundancy Checking is used. This techniques aims to the identification of functionality redundancy and is explained as follows:

Redundancy Checking occurs in a software system or in a set of software systems when multiple software components provide the same function or serve the same purpose, or define the same data. Redundancy can occur at all levels of system abstraction as source code, designs and requirements. When redundancy is identified and when it is feasible, a generic reusable component should be defined and used to replace all the redundant functions.

Considering components as classes and methods (see Step 1), redundancy can be viewed in:

- **For Class description Redundancy:**

  - If multiple classes have common properties, consider generalizing the classes to create a supertype. The differences among the similar classes can be captured as subtypes for the new generalized class. Consider creating an abstract class that defines the common properties, which can then be inherited by its subclasses.

  - Compare all the class descriptions to identify any class redundancies. A class description consists of the class name, specification of the interface, data attributes, object reference attributes, and methods signatures. A signature gives a detailed description of the methods to use for that class.

  - For object classes with the same or similar descriptions, attempt to eliminate these redundant classes by creating a supertype to replace the redundant classes.

- **For Methods Redundancy:**

  To identify processes that may be providing the same or very similar function, look for methods or specifications that have:

  - the same or similar signatures
  - the same or similar complexity value based on the McCabe Complexity Metrics (McCabe Complexity number) and/or Halstead Software Metrics, or
  - meet the same or similar requirements.

Redundant methods can be combined and replaced into one generic method.

The metrics Halstead Software Metrics and McCabe Complexity are explained in sections 4.4.1 and 4.4.2 respectively. These metrics are also used by the technique *Identifying and Qualifying Reusable Components* explained in chapter 2 (see section 2.4.1).

In this step it can be realised that a Reengineering process takes place through the restructuring of class and methods organization. To complete this process, Reverse Engineering must be performed in order to reflect the changes in the design.

4.3.4 Step 4. Count the Number of Static Calls

The idea in this step is to propose a ratio between the number of static calls addressed to a given component and the average number of static calls computed in the system. If the result shows an expressive amount of message sending to a component then this component can be considered reusable in the context of the application domain.

The principles suggested are based on the Reuse Frequency metric presented in section 2.4.1 (Identifying and Qualifying Reusable Software Components).

4.3.5 Step 5: Analyse the Source Code to Detect Dependencies

The goal of this step is the evaluation of the dependencies between software components. *The theoretical reusability of a software component is defined as the amount of dependencies that exists between that component and other software components* [JI94].

Ideal examples of reusable software components can be defined as those, which have no dependencies on other software components. In other words, the reusability of a component can be thought of as inversely proportional to the amount of external dependencies required by that component.

By studying the dependencies between software elements at code level, a determination can be made of the reusability of those elements in other contexts. For example, if a component of a program uses or depends upon another component, then it would not normally be reusable in another system where that other component was not also present. On the other hand, a component of a software program that does not depend on any other software component can be used, in theory at least, in any arbitrary context.

Two main principles can support the evaluation of the dependency level of a component: cohesion and coupling. They can be defined as follows:

Cohesion *is an integral part of modular design and represents the strength of the relationship between module elements* [CK94]. A cohesive module performs a single task and requires minimal interaction with procedures in other parts of a program. Ideally a cohesive module should do only one thing. It is desirable to have modules which are highly cohesive and strongly related to one another. The elements of one module should not be strongly related to those of another module as this leads to tight coupling which is undesirable. Good abstraction typically exhibits high cohesion.

Coupling *two objects are coupled if and only if at least one of them acts upon the other* [MM95]. Since objects of the same class have the same properties, two classes are coupled when methods declared in one class use methods or instance variables defined by the other class. Excessive coupling between object classes prevents reuse. The more independent a class is, the easier it is to reuse it in another application.

To measure the coupling and cohesion of a component some metrics are suggested. They are explained in sections 4.4.3 and 4.4.4.

Another way to evaluate the level of dependency of a component is by measuring its Depth of Inheritance as will be explained in the next step.

4.3.6 Step 6: Measure the Depth of Inheritance of a Component

Class Inheritance Depth can be described in: Depth of the class in the inheritance tree. Large nesting numbers might indicate a design problem and usually results in subclasses that are not specializations of all the superclasses. A subclass should ideally extend the functionality of the superclass [CK94].

As it can be realized in the previous assumption, a deep class inheritance points out a design problem that probably prejudice potential class reusability. This fact indicates that a reengineering in the design structure might be necessary. Moreover as much as deep is the class inheritance more dependent is the class on the behaviour and structure of the others. Therefore, less chances it has to be reused in other contexts.

Depth of Inheritance is a measure that is easy achieved and provided by many tools supporting metrics.

4.3.7 Step 7: Determine the potential use of the component

Taking advantage of the commonalties explained in the step 2 it is possible to determine the potential use of the component through the comparison of this component identified in a similar context.

When a component is common, that is likely to be needed frequently in projects or frequently included in a system, it can be searched in other software projects to determine if the same or similar component was or is being developed for those projects. If so, attempt to reuse the component or to plan its development jointly with those projects to enable the component to be used in this project, in the other project(s), and in possible future projects.

4.4 Outlined Metrics

This section describes the metrics suggested by this Guideline.

The following two sets of Complexity metrics are used to measure the complexity of a program and also can be used to detect redundancies in software programs. If two or more components have the same or similar complexity characteristics, they are likely to be providing the same function and may be redundant.

Afterwards, metrics to measure coupling and cohesion are explained.

4.4.1 Halstead Metrics

Halstead’s Software Science Complexity Metrics was developed by M.H.Halstead [Mc97] to measure the complexity characteristics of software programs, principally to attempt to estimate the programming effort.

Halstead's Metrics are based on counting the number of unique operators and operands in a program. The measurable and countable properties are:

n1 = number of unique or distinct operators appearing in that implementation
n2 = number of unique or distinct operands appearing in that implementation
N1 = total usage of all of the operators appearing in that implementation
N2 = total usage of all of the operands appearing in that implementation

From these metrics Halstead defines:

i. the vocabulary n as n = n1 + n2
ii. the implementation length N as N = N1 + N2

Operators are reserved programming language words such as ADD, GREATER THAN, MOVE, READ, IF, CALL; arithmetic operators such as +,-,*,/; and logical operators such as GREATER THAN or EQUAL TO. The number of operands consists of the numbers of literal expressions, constants and variables.

For e.g., the assignment statement

p = q

has one operator and two operands

The Halstead measures are calculated using these four parameters. These measures are listed below.

Program length

N = n1 log n1 + n2 log n2

Actual Halstead length

Halstead length = N1 + N2

Program's vocabulary

Program vocabulary = n1 + n2

Volume

Volume = (N1 + N2) * log (n1 + n2)

Level

Level = (2/n1)*(n2/N2)

Difficulty

Difficulty = 1/Level

Effort

Effort = (Volume/Level)/(18*60) Minutes

Bug Predicted

Bugs predicted = Volume/3000

Advantages of Halstead :

i. Do not require in-depth analysis of programming structure.
ii. Predicts rate of error.
iii. Predicts maintenance effort.
iv. Useful in scheduling and reporting projects.
v. Measure overall quality of programs.
vi. Simple to calculate.
vii. Can be used for any programming language.
viii. Numerous industry studies support the use of Halstead in predicting programming effort and mean number of programming bugs.

Drawbacks of Halstead:

i. It depends on completed code.
ii. It has little or no use as a predictive estimating model. But McCabe's model is more suited to application at the design level.

4.4.2 McCabe's Cyclomatic number

Cyclomatic number proposed by McCabe [HS96] is one of the widely used measures to understand the structural complexity of a program and it can be used to detect software redundancies. This number, based on a graph-theoretic concept, counts the number of linearly independent paths through the program. In practice it is a count of the number of test conditions in a program. They can be calculated by hand or by automatic complexity metrics tools.

If G is the control flowgraph of program P, and G has e edges (arcs) and n nodes, then Cyclomatic number $V(G) = e - n + 2$

Or, more simply, if d is the number of decision nodes in G, then Cyclomatic number
$V(G) = d + 1$

The value of d for the Java constructs is given below

if..then	1
if..then..else	1
for	1
while	1
do..while	1
case statements	1

On the basis of empirical research, McCabe claimed that modules with high values of V (G) were those most likely to be fault-prone and unmaintainable.

Advantages of McCabe Cyclomatic Complexity :

i. It can be used as a ease of maintenance metric.
ii. Used as a quality metric, gives relative complexity of various designs.
iii. It can be computed early in life cycle than of Halstead's metrics.
iv. Measures the minimum effort and best areas of concentration for testing.
v. It guides the testing process by limiting the program logic during development.
vi. Is easy to apply.

Drawbacks of McCabe Cyclomatic Complexity:

i. The Cyclomatic complexity is a measure of the program's control complexity and not the data complexity
ii. The same weight is placed on nested and non-nested loops. However, deeply nested conditional structures are harder to understand than non-nested structures.
iii. It may give a misleading figure with regard to a lot of simple comparisons and decision structures. Whereas the fan-in fan-out method would probably be more applicable as it can track the data flow

4.4.3 Lack of Cohesion of Methods (LCOM)

The original object-oriented cohesion metric, LCOM, is given by Chidamber and Kemerer [CK94]. Lack of Cohesion of Methods (LCOM) is a measure of the structural cohesion of classes.

LCOM is defined as a count of the number of method pairs whose similarity is zero minus the count of method pairs whose similarity is not zero. The degree of similarity, between two methods is given by :

If there are no common properties then similarity = 0.

Consider a class C with n methods M1,M2...Mn. Let {Ij} = set of instance variables used by method Mi. There are n such sets {I1}...{In}.

If P = {(Ii,Ij), Ii intersection Ij is equal to nullset} and
Q = {(Ii,Ij), Ii intersection Ij is not equal to nullset} then

LCOM(Chidamber - Kemerer) = P - Q, if (P > Q)
= O, Otherwise.

For a perfectly cohesive class the value of LCOM(Chidamber-Kemerer) is 0, and for a totally non-cohesive class the LCOM(Chidamber-Kemerer) value equals (n(n-1))/2 where n represents the total number of methods present in the class.

LCOM (Li - Henry):

Li and Henry defined LCOM as the number of disjoint sets of methods, where any two methods in the same set share at least one local instance variable.

In LCOM(Li-Henry), a value of 1 represents a perfectly cohesive class, whereas for a totally non-cohesive class the value equals the total number of methods present in the class.

LCOM (Henderson - Sellers):

Consider a set of methods {Mi} (i=1,...m) accessing a set of attributes {Aj} (j=1,...a). Let the number of methods which access each datum be Mu(Aj). Then

LCOM(Henderson-Sellers) = (((1/a)sigma j = 1 to a Mu(Aj)) - m) / (1 - m).

The value of LCOM(Henderson-Sellers) is 0 for a perfectly cohesive class and greater than 0 for non-cohesive classes.

Example :

Given, member variables: I,J,K,L and member functions A,B,C,D

Member function A accesses variables {I,L}
Member function B accesses no variables
Member function C accesses variables {J,L}
Member function D accesses variables {K}

Then the value of Lack of Cohesion of Methods based on these three methods is given below:

LCOM(Chidamber-Kemerer) = 4 (Since P=5, Q=1 and P > Q)

LCOM(Li-Henry) = 3 (Disjoint sets are {A,C}, {B}, {D})

LCOM(Henderson-Sellers) = 0.916(m=4, a=4, Mu(Aj) = 5)

4.4.4 Coupling between object classes (CBO)

Coupling between object classes (CBO) for a class is a count of the number of other classes to which it is coupled [CK94 and MM95]. CBO relates to the notion thatt an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another.

This metric can be calculated by *Analytical Evaluation of Coupling Between Objects (CBO*) [CK94] . CBO for a class is a count of the number of other classes to which it is coupled. CBO relates to the notion that an object is coupled to another object if one of them acts on the other, i.e., methods of one use methods or instance variables of another. Two classes are coupled when methods declared in one class use methods or instance variables defined by the other class.

As per assumption 1, there exist classes P, Q and R such that $\mu(P) \neq \mu(Q)$ and $\mu(P) = \mu(R)$ thereby satisfying properties 1 and 2. Inter-class coupling occurs when methods of one class use methods or instance variables of another class, i.e., coupling depends on the manner in which methods are designed and not on the functionality provided by P. Therefore, Property 3 is satisfied. Let P and Q be any two classes with $\mu(P) = \eta_P$ and $\mu(Q) = \eta_Q$. If P and Q are combined, the resulting class will have $\eta_P + \eta_Q - \partial$ couples, where η_P is the number of couples reduced due to the combination. That is $\mu(P+Q) = \eta_P + \eta_Q - \partial$, where ∂ is some function of the methods P and Q. Clearly, $\eta_P - \partial \geq 0$ and $\eta_Q - \partial \geq 0$ since the reduction in couples cannot be greater than the original number of couples. Therefore,

$$\eta_P + \eta_Q - \partial \geq \eta_P \text{ for all P and Q and}$$
$$\eta_P + \eta_Q - \partial \geq \eta_Q \text{ for all P and Q}$$

i.e. $\mu(P+Q) \geq (P)$ and $\mu(P+Q) \geq (Q)$ for all P and Q. Thus, Property 4 is satisfied. Let P and Q be two classes such that $\mu(P) = \mu(Q) = \eta$, and let R be another class with $\mu(R)=r$.

$$\mu(P+Q) = \eta + r - \partial, \text{ similarly}$$
$$\mu(P+Q) = \eta + r - \beta.$$

Given that ∂ and β are independent functions, they will not be equal, i.e. $\mu(P+R)$ is not equal to $\mu(Q+R)$, satisfying Property 5. For any two classes P and Q, $\mu(P+Q) = \eta_P + \eta_Q - \partial$.

$$\mu(P+Q) = \mu(P) + \mu(Q) - \partial \text{ which implies that}$$
$$\mu(P+Q) \leq \mu(P) + \mu(Q) \text{ for all P and Q}$$

Therefore Property 6 is not satisfied.

4.5 Conclusions

This chapter has presented the core of this dissertation by explaining the set of steps that can be followed in order to detect reusable components in OO legacy code.

In this context there are still some aspects that need to be considered. Detect components is not enough to guarantee their reuse. To be reusable they need to be completed with some essential features :

Well defined Interface - According to [BG98] "*an interface is a collection of operations that are used to specify a service of a component"*. Interfaces are the glue that binds component together determining what each component can require to another. Besides interfaces break the direct dependency between components since they make the role of a bridge allowing their communication. It is important to state that to be used, a component needs to be in a context in which all the services it needs are provided by the interfaces of other components.

Documentation – Information is needed to understand, identify, compare, modify (specialize) and integrate the component. A very basic reuse truism is that to identify a component to reuse and then reuse it in the creation of a new system, the reuser must understand what it does. A primary inhibitor to reuse is component understanding. In addition, information more specific to reuse, such as under what conditions it can be reused and how to go about reusing is needed.

Certification – Reusers will not want to utilize reusable components unless they are confident in the component's validity. Every reusable component should be subject to a certification process. The process will depend upon the types of reusable components. Certification information should be "carried along" with the reusable component as part of its documentation. As stated in [SM97]: The idea behind the certification of component is to guarantee that a specific set of guidelines has been met.

5 Applying the Guideline in a Case Study

In order to demonstrate the application of the Guideline, some of the directives suggested by each step are used in practice. The case study chosen is a banking application developed by a student at Ecole des Mines de Nantes, France.

This chapter stars explaining the case study through its environment and design, going then deep in the source code in order to support the Guideline application.

5.1 The Case Study: A Banking Application

The application used to demonstrate the Guideline is a banking application implemented in JAVA programming language and making use of the CORBA (Common Object Request Broker Architecture) concepts. The application [GP99] was developed by a student at Ecole des Mines de Nantes during his graduation period.

This application was suggested as the case study in this work because of three main reasons.

- This application is an OO-legacy system. Therefore it is in accordance with the purposes of this Guideline allowing its use.
- Banking application can also be considered as a generic application. Consequently it allows an easy understanding of its functionalities inside a well-known context.
- It is really simple and considering the time constraint it would be not possible to analyse a more complex example.

Banking applications deal with a set of financial aspects for a rapidly growing electronic market. Since this market evolves quickly, bank services must compete in providing efficient innovative services to end users and commercial customers. Besides, by its nature, the banking application applies distributed computation.

The norm CORBA allows the creation of distributed applications in a heterogeneous environment. CORBA has also an another important feature that is the Interface Definition Language (IDL). This application utilizes the logical ORBACUS 3.1 to support every feature of CORBA IDL.

Java is an object-oriented programming language which programs can be executed on many different platforms without recompilation, even when they have a graphical user interface. Therefore, it has interesting features to support software reuse. The portability feature not only increases widespread use of such programs, but also provides a platform for increased reusability of components.

Among CORBA's advantages are: the use of an object-oriented paradigm, the hiding of the programming language and the operating system differences supporting distributed systems. Moreover CORBA IDL is used in order to ensure programming language independence. It describes a component interface being then mapped to the programming language in which the component is implemented.

5.1.1 Describing the Application

According to the step 2 (see section 4.3.2), a deep study of the application becomes essential to get a strong knowledge of the domain in which the application is inserted. In this work, the study was based on the various sources of information available about the system, as the design in a UML Class Diagram, a textual description and the source code. Now some explanation of the application resultant from this study is presented.

This application is based mainly on the following classes:

Class **Agency** – This class has all features that characterize a bank agency, such as: its address, reference of server bank, list of clients, list of accounts and the persistence of the clients. The Class Agency also provides the basic operations to create, delete, search and list its clients.

Class **Account** – This class encompasses the basic operations to deposit and to withdraw money. Besides it provides the control on the maximum limit authorised, the type and the number of account. It has a reference to a client and an agency, offering in this way the possibility to recover rapidly the related information.

Class **Saving Account** –This class inherits from class Account. It adds to the Account structure just two attributes exchange and limit_Max. The notion of polymorphism is used in the methods deposit and withdrawal that are overridden in Saving Account. It is not authorised withdrawals that can lead to negative balance.

Class **Bank** – This class covers a reference to all the agencies connected to Bank Server (see section 5.1.2).

Class **Client** – This class expresses all the information about client, such as: identification number, name, and first name, date of birthday, address and sex. It has a reference to the agency in which the client has accounts. This class provides the basic operations to create, delete, list and search accounts of a client.

The original UML Class diagram of this application showing classes structure, their operations and relationships is depicted in Appendix A. Considering that the Case Study is in French, in the same Appendix following the diagram, a translation of terms from French to English is presented.

The source code of these main classes analysed is provided in Appendix B.

5.1.2 The Architecture of the Application

The banking application is distributed in workstations divided in various sites connected through a Local Area Network (LAN). The LAN's are connected through a Wide Area Network (WAN). A Bank server, some Agency servers and shared clients of Agency servers are showed in figure 4.

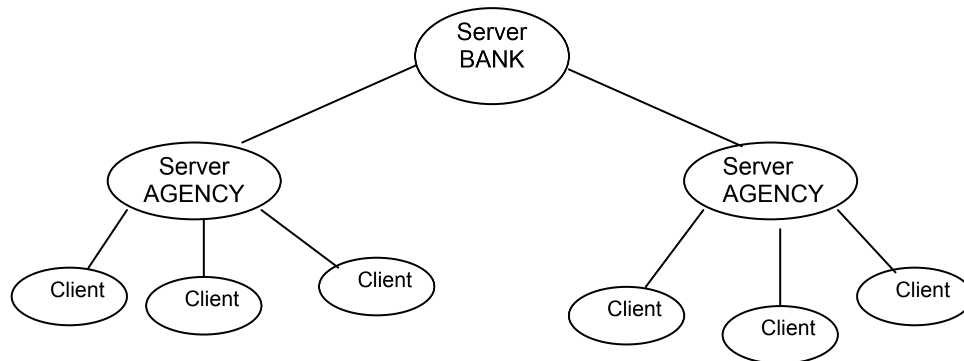


Figure 4 : Banking Application Context

The clients can be clerk, financial service, and commerce, cash dispenser, etc.

The application developed implemented only server classes in Java and a graphic interface to the client part. The bus proposed by OMG (Object Management Group) allows a transparent communication between clients and servers as represented in figure 5.

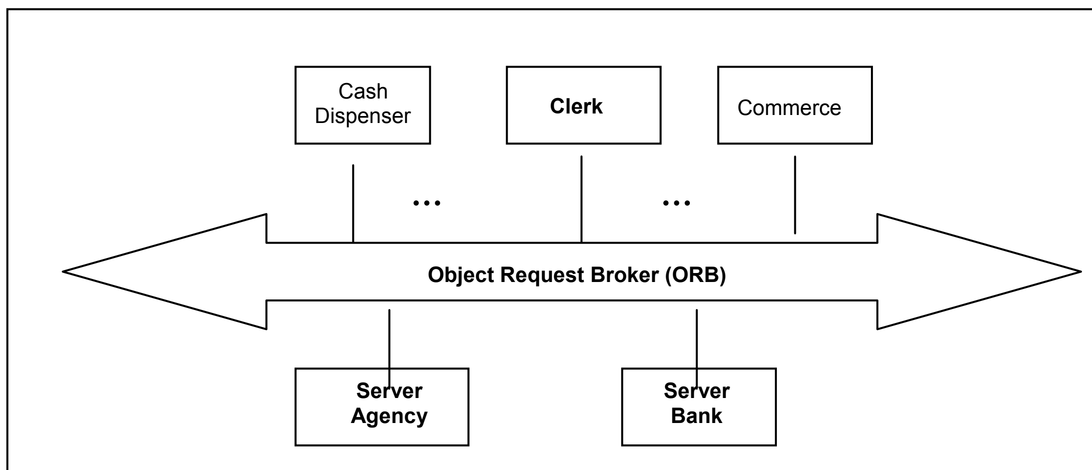


Figure 5: Common Object Request Broker Architecture

After explaining the main functionalities and architecture of the banking application, in the following sections the use of Guideline directives in order to detect reusable components in the source code will be presented.

5.2 Checking Redundancy

According to the step 3 (see section 4.3.3) the goal of this directive is to detect redundancy among classes and methods. It is necessary to focus in the functionality aspect to achieve this purpose. Some metrics are suggested in this direction.

As an automatic way to calculate metrics the Tool ***JStyle*** is used. ***JStyle*** analyses Java source code to offer comments on the code and to generate useful metrics. This tool does a static analysis of Java source code to uncover flaws like naming convention violations, improper exception handling and efficiency impeding coding practices. Among the metrics calculated by ***JStyle*** tool, the ones used to apply the Guideline are: Halstead Measures, Cyclomatic Number and LCOM (Lack of Cohesion of Methods).

Reminding the statements outlined in Step 3, it must be analysed the same or similar complexity value based on the McCabe Complexity Metrics and/or Halstead Software Metrics to detect method redundancy. JStyle computes Cyclomatic Number for every method in a class, not to the module as a whole. Therefore the values produced by JStyle for these metrics for the main methods of the banking application are as follows:

Class Account

Methods/ Metrics	Cyclomatic Number	Program Length	Volume	Development Effort
operations	2	54.7052	179.307	2.34806
deposit	1	154.257	584.738	3.92533
withdrawal	2	198.054	759.677	8.46284

Table 1: Methods of Class Account

Class Agency

Methods/ Metrics	Cyclomatic Number	Program Length	Volume	Development Effort
search_account	***4***	***133.662***	***393.463***	***6.55771***
list_agency_account	***2***	***57.7052***	***187.647***	***2.59379***
create_a_client	5	202.922	702.957	13.556
del_a_client	4	106.606	312.478	3.61664
search_a_client	4	117.593	353.296	5.77503
list_clients	2	57.7052	187.647	2.59379
add_Account	1	9.50978	15.5098	0.0215414
del_Account	3	91.1253	260.056	2.44497
recreate	2	93.4869	260.056	1.41643
change	5	243.525	856.536	11.783

Table 2: Methods of Class Agency

Class Bank

Methods/ Metrics	**Cyclomatic Number**	**Program Length**	**Volume**	**Development Effort**
insert_an_agency	13	134.544	1296.41	41.7432
delete_an_agency	7	128.362	685	13.5661
search_an_agency	7	144.546	754.005	19.1216
delete_Adress_Agency	7	128.362	625	11.4776
recreate_Adress	2	93.4869	260.056	1.41643
change_Adress	2	113.93	354.633	2.76517
list_Agences	2	57.7052	187.647	2.59379
list_Adress_Agences	2	57.7052	187.647	2.59379

Table 3: Methods of Class Bank

Class Client

Methods/ Metrics	**Cyclomatic Number**	**Program Length**	**Volume**	**Development Effort**
create_an_account	1	48.7291	104	0.404444
delete_an_account	4	122.603	351.748	4.274471
list_ client_account	***2***	***57.7052***	***187.647***	***2.59379***
Deposit	9	372.12	2582.89	103.709
search_account	***4***	***128.09***	***375***	***5.70602***

Table 4: Methods of Class Client

Class Saving Account

Methods/ Metrics	**Cyclomatic Number**	**Program Length**	**Volume**	**Development Effort**
exchange	1	2	4.75489	0.00440267
type_account	1	2	4.75489	0.00440267
limit_Max	1	2	4.75489	0.00440267
deposit	2	106.709	278.827	2.79687
withdrawal	2	106.709	278.827	2.79687

Table 5: Methods of Class Saving Account

** Development Effort in mins*

The methods ***search_account*** of class Client and ***search_account*** of class Agency**,** as well as ***list_account_agency*** and ***list_account_client*** of Agency and Client respectively, present redundant functionality according to the assumptions of step 3: similar names, similar requirements and similar values in metrics Halstead and McCabe. It can be realized that considering the Halstead metrics (see section 4.4.1) just the most significant measures are taken into account.

As stated in the Guideline, the solution to this situation can be the creation of one generic component (method) to each tuple of redundant ones that can replace their functionality in the application. Therefore, a new method ***list_account*** with a parameter account[] (an object list of accounts) is created. It is placed in the class Account which better meets its

functionality. In the same way, a generic ***search_account*** method with a parameter account[] (the list of accounts to be searched) is created in the class Account. The substituted methods are extracted from the original classes. Objects of classes Agency and Client can then require this service to objects of type Account.

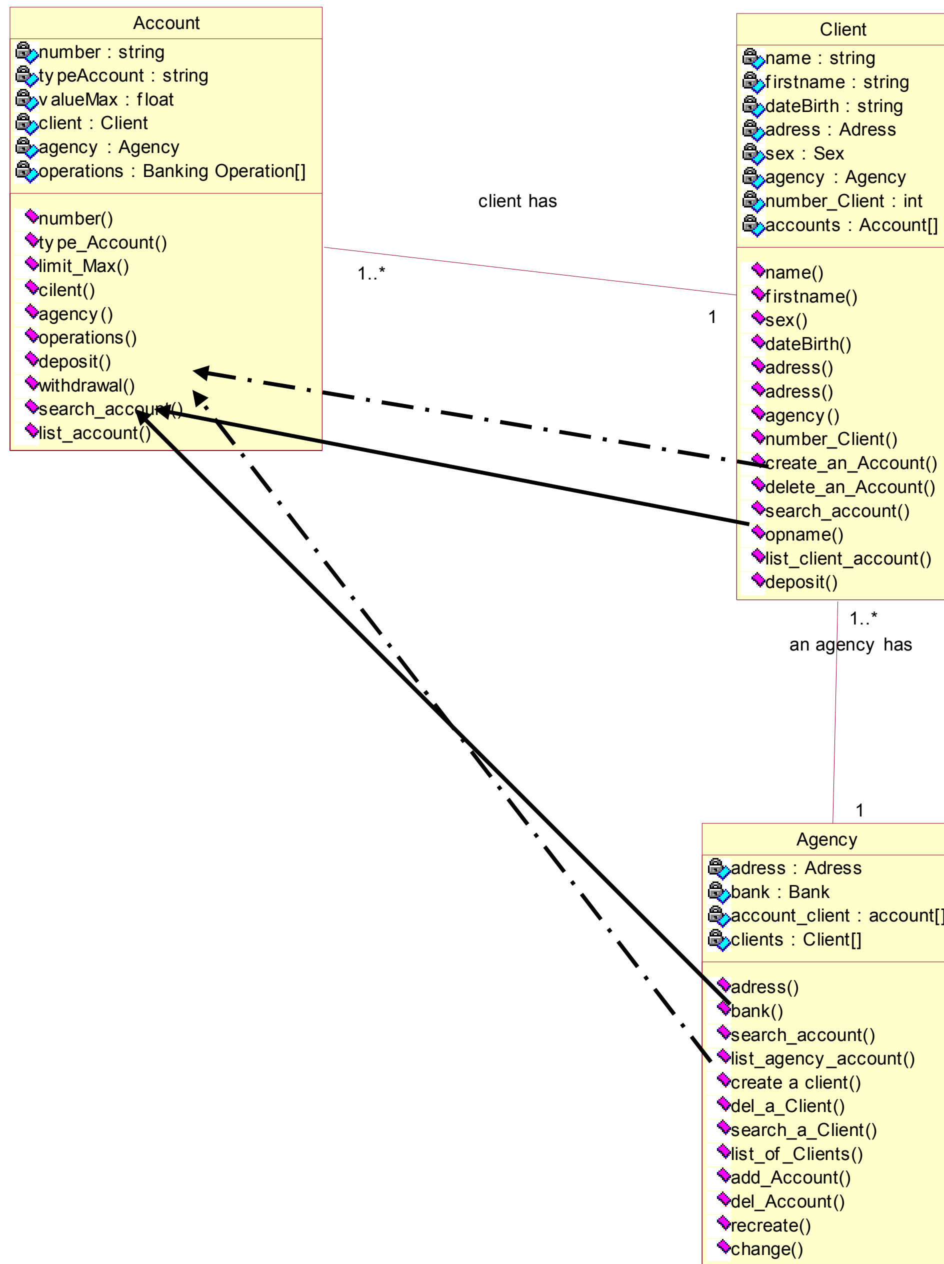


Figure 6 : Demonstrating a part of a reengineering process

Through snippets of the source code, presented as follows, it is possible to confirm this redundant functionality between those pairs of methods.

Class Client – Method list_Client_Account

```
      public Compte[] liste_des_Comptes_de_client(){

                  int i=0;
                  Compte[] aux_desComptes;
                  aux_desComptes= new Compte[lesComptes.size()];

            while(i<lesComptes.size()) {
                  aux_desComptes[i]=(Compte)lesComptes.elementAt(i);
                  i++;
            }
                  return aux_desComptes;
      }
```

Class Agency – Method List_Agency_Account

```
      public Compte[] liste_des_Comptes_de_agence() {
                  int i=0;
                  Compte[] aux_desComptes_Client;
                  aux_desComptes_Client= new
Compte[desComptes_Client.size()];

            while(i<desComptes_Client.size()) {

      aux_desComptes_Client[i]=(Compte)desComptes_Client.elementAt(i);
                  i++;
            }
                  return aux_desComptes_Client;
      }
```

Class Client - Method search_account

```
      public Compte rechercher_le_Compte(String numero_Compte)
                                    throws ProblemeClient{

                  boolean ok=false;
                  int i=0;

            while(((i<lesComptes.size()) && (ok==false))) {
                  if((((Compte)lesComptes.elementAt(i)).numero()).equals
(numero_Compte)) {
                        ok=true;
                  }
                  i++;
            }

                  if(ok==true) {
                  return (Compte)lesComptes.elementAt(i);
                  }
                  else {
```

```
                  throw new ProblemeClient("Le compte avec le numero
"+numero_Compte+
                                                " n'exite pas !");
                        }
```

Class Agency - Method search_account

```
      public Compte rechercher_le_Compte(String numero_Compte)
                              throws ProblemeAgence {
                  boolean ok=false;
                  int i=0;

            while(((i<desComptes_Client.size()) && (ok==false))) {

      if((((Compte)desComptes_Client.elementAt(i)).numero()).equals(nume
ro_Compte)) {
                        ok=true;
                        i--;
                        }
                  i++;
                  }
                  if(ok==true) {
                  return (Compte)desComptes_Client.elementAt(i);
                  }
                  else {
                        throw      new      ProblemeAgence("Le      compte
"+numero_Compte+
                                                " n'exite pas !");
                  }
      }
```

Concluding, it can be reliased that the classes Client and Agency present some redundant funtionality, however their structures and main functionalities are not equal. Therefore, the similar funtionality can not be factored out to a superclass, but just extracted and placed in another correct class.

5.3 Checking Dependency

Dependency between classes can be checked through the evalutaion of their level of coupling and cohesion as stated in step 5 of the Guideline (see section 4.3.5). To get coupling and cohesion measures some metrics can be applied to the classes. The ones used here are LCOM (Lack of Cohesive of Methods) to calculate cohesion, see section 4.4.3, and CBO (Coupling between Object Classes) to calculate coupling, see section 4.4.4. LCOM is given by the JStyle tool which was mentioned in the previous section. Three values of LCOM are obtained as can be seen in the following table.

Classes / LCOM	Account	Agency	Bank	Client	Saving Account
LCOM (Chidamber-Kemerer)	45	91	66	136	15
LCOM (Li-Henry)	10	14	12	17	6
LCOM (Henderson-Sellers)	0.944444	0.846154	0.840909	0.945313	0.933333

Table 6: Metric LCOM per Classes

Through this table it can be realized that the values of the metric LCOM are really showing totally none cohesive classes in the system. For example, taking the classic Chidamber-Kemerer LCOM metric, a totally cohesive class is considered when the result value is close to 0, while totally non cohesive classes present the LCOM value equal (n(n-1))/2, in which n represents the total number of methods present in the class. Taking class Bank, for example :

n = 12 => (n(n-1))/2 = (12(12-1))/2 = 66

Totally non cohesive classes imply high level of coupling between them. Therefore, CBO metric does not need necessarily to be calculated in this example. According to the directives of step 5, no classes in this case study can be considered as reusable components. They are not independent enough, or they are totally dependend on others to describe a clear functionality that can be resued in any other context.

5.4 The results obtained

As determined in step 2 a deep study of the banking application domain was taken in order to apply Guideline directives. After applying some directives two methods were identified as components in the application (see section 5.2) according to the types of components determined in step 1.

Considering the example is really simple, and the level of coupling between classes is too high, no class could be pointed out as a candidate reusable component. The depth of inheritance, another measure that can be used to determine class dependency (see step 6 of the Guideline – section 4.3.6) was not possible to be applied in the example. There was, considering the functionality of the application and not CORBA aspects, just one inheritance with one level in the system.

5.5 Conclusion

The experience with this case study indicates that the Guideline is feasible in an operational context, and that its use in complex systems must be encouraged supporting

further validations. The directives applied were just to exemplify the use of the Guideline. It was not possible to apply all the directives suggested because of the limited time available.

Considering the use of CORBA in the case study, once a component is identified CORBA can be helpful in the definition of its interface through the IDL. As stated before in section 4.5, the relationship between component and interface is very important. CORBA as the most common component-based operating system facilities the use of interfaces as the glue that binds components together.

6 Conclusions

In the software industry nowadays one of the main issues considered is how to evolve the existing softwares in order to support the rapidly changing world we live in. Political, economic and social factors are always changing and the softwares need to be able to efficiently follow this evolution in order to continue producing expected results. Companies compete on small in quickly introducing innovative services. In order to continue in the marketing, this innovation needs to be supported with an efficient response time. Considering these aspects, software reuse has become the key part of companies software engineering strategy in order to support software improvement and evolution.

The hardware industry has walked in this direction for many years realising that reuse in fact facilitates an engineering process. Software industry in turn has been devoting efforts in this direction just in the last years. Object Oriented paradigm tries to address this issue but not with complete practical results in large-scale software. Many results are still expected in this sense.

The thesis of this research has been that detecting components in legacy systems is the first and fundamental step in order to support reuse. Therefore the work presented focused on the description of a Guideline that can help in identifying reusable components in OO legacy systems. This Guideline was developed to consolidate some of the best existing practices.

6.1 Contributions

Considering the detection of reusable components is still an unexplored area, this thesis has also the purpose to contribute to the research in this context opening new ways for its investigation. Moreover, taking an industrial context in which reusable components have been receiving a great importance, the use of this guideline can be a way to help in taking advantage of source code making them more reusable. Therefore more quality is ensured in systems evolution and new development processes, since existing components have been tested and approved before.

In addition companies have been investing lots of money in developing software systems from scratch for many years. Simply replacing these systems by new ones because of requirements change is not cost-effective. Therefore taking advantage from legacy code has been accepted as a good alternative to save money.

6.2 Future work

We are still far from the ideal scenario of composing the most software systems from existing components. To be able to use components, it is essential to know early in the development process which components are available. Then, a good knowledge of the application domain becomes essential in order to match the existing components to the expected requirements.

The Guideline presented covers the main aspects researched in order to support the identification of components in OO legacy code. It is really not a complete Guideline considering time constraint and the fact that each time it is applied more statements can be concluded. Composing a Guideline can be seen as a recursive process. Therefore extending this Guideline can be considered as a continuous investigation process in the area of reusable software components.

The case study reported in this work provided initial results and practical feedback on the main aspects of this Guideline to detect reusable components. It seems that the Guideline addresses important and often ignored aspects in the source code. Although this initial experience from the use of the Guideline is encouraging, further and more formal experiments are required to validate the process.

Detecting components as proposed in this thesis is not the only step necessary to make them reusable. Another fundamental aspect that needs also to be considered is the form to be given to a component in order to allow its application in any context. This form is mainly based on the interface aspect. Components are linked to a system through the services it can offer.

There exist many emerging component-based software technologies that try to reach this goal of establishing well-defined and clear component interfaces. The main aspect is to separate component implementations from the interfaces. This is an important aspect that can be considered as extensions of existing Guidelines in the area of reusable components.

Since there is still a long way to go until systematic reuse of software components becomes concrete, it is required that more investigation and research in this area is undertaken achieving many of the points suggested as future extensions. It is hopped that as soon as these results are achieved, they can be widely spread helping industries and software community in general to progress in the software engineering process.

7 References

[AK98] Anish Arora and Sandeep S. Kulkarni. *Component Based Design of Multitolerant Systems*. *IEEE Transactions on software Engineering*. Vol.24 №1pages 63-78 January 1998.

[BB90] John W.Bailey and Victor R. Basili. Software Reclamation: *Improving Post-Development Reusability. 8th Annual National Conference on Ada Technology* 1990 pages 477-498.

[BG98] Grady Booch., James Rumbaugh and Ivar Jacobson, *The unified modeling language user guide*1998.

[BM94] Bertrand.Meyer. *Reusable Software: The base oriented component libraries*, 1994.

[CB91] Gianluigi Caldiera and Victor R. Basili. *Identifying and qualifying reusable software components. IEEE Computer* pages 61-70, February 1991.

[CK94] Shyam R. Chidamber and Chris F. Kemerer. *A Metrics Suite for Object Oriented Design. IEEE Transactions on Software engineering*, Vol. 20, nr. 6, june 1994.

[DW98] Desmond F. D'Souza and Alan Cameron Wills. *Objects, components, and Frameworks with UML – The Catalysis Approach*, 1998.

[DK93] Michael F. Dunn and John C. Knight. *Automating the detection of reusable parts in existing software. In Proceedings of the 15th International Conference on Software Engeeniring*, pages 381-390, 1993.

[EM98] Michel Ezran, Maurizio Morisio and Colin Tully. *Pratical Software Reuse: The essential guide*, 1998.

[GO95] David Garlan, Robert Allen and John Ockerbloom. *Why it's hard to build systems out of existing parts. Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle WA, April 1995.

[GP99] Pascal Grolleau. *Projet: Service Bancaire avec CORBA et JAVA – Ecole des Mines de Nantes*, February, 1999.

[HS96] Henderson-Sellers, Brian. *Object-oriented metrics: measures of complexity*, 1996 Chapter 1 pages 1-25.

[JG97] Ivar Jacobson, Martin Griss and Patrik Jonsson. *Software Reuse – Architecture, Process and Organization for Business Success, ACM Press Books*, 1997.

[JH98] Christian Damsgaard Jensen and Daniel Hagimont. *Protection Reconfiguration for Reusable Software. IEEE Computer*, 1998 pages 74-80

[JI94] Ivar Jacobson. *Object-Oriented Software Engineering – A Use Case Driven Approach*, 1994.

[KA96] Rick Kazman, Paul Clements, Len Bass and Gregory Abowd. *Classifying Architectural Elements as a Foundation for Mechanism Matchin*g. USA 96

[LV96] Filipo Lanubile and Giuseppe Visaggio. *Extracting reusable functions by Flow Graph-Based Program slicing. IEEE Transactions on software Engineering.* Vol.23 №4 pages 246-259, April 97.

[Mc94] Carma McClure. *Reuse: Re-engineering the Software Process. Extended Inteligence, Inc* 1994.

[Mc97] Carma McClure. *Software Reuse Techniques – Adding Reuse to the Systems Development Process*,1994.

[MM95] Martin Hitz and Behzad Montazeri – *Measuring Coupling and Cohesion In Object-Oriented System. Symposium on Applied Computing*. Oct, 1995.

[MN91] Maiden, Neil. *Analogy as a paradigm for specification reuse. Software Engineering Journal*, January 1991.

[MR97] Melody M. Moore and Spencer Rugaber. *Domain Analysis for Transformational Reuse. IEEE Computer,*1997 pages 156-162.

[NJ96] James M. Neighbors – Bayfront Technologies, Inc. *Finding Reusable Software Components in Large Systems. IEEE Proceedings of WCRE'96* pages 2-9

[SC97] Clemens Szyperski. *Component Software Beyond Object-Oriented Programming*, 1997. Chapters 1 and 4.

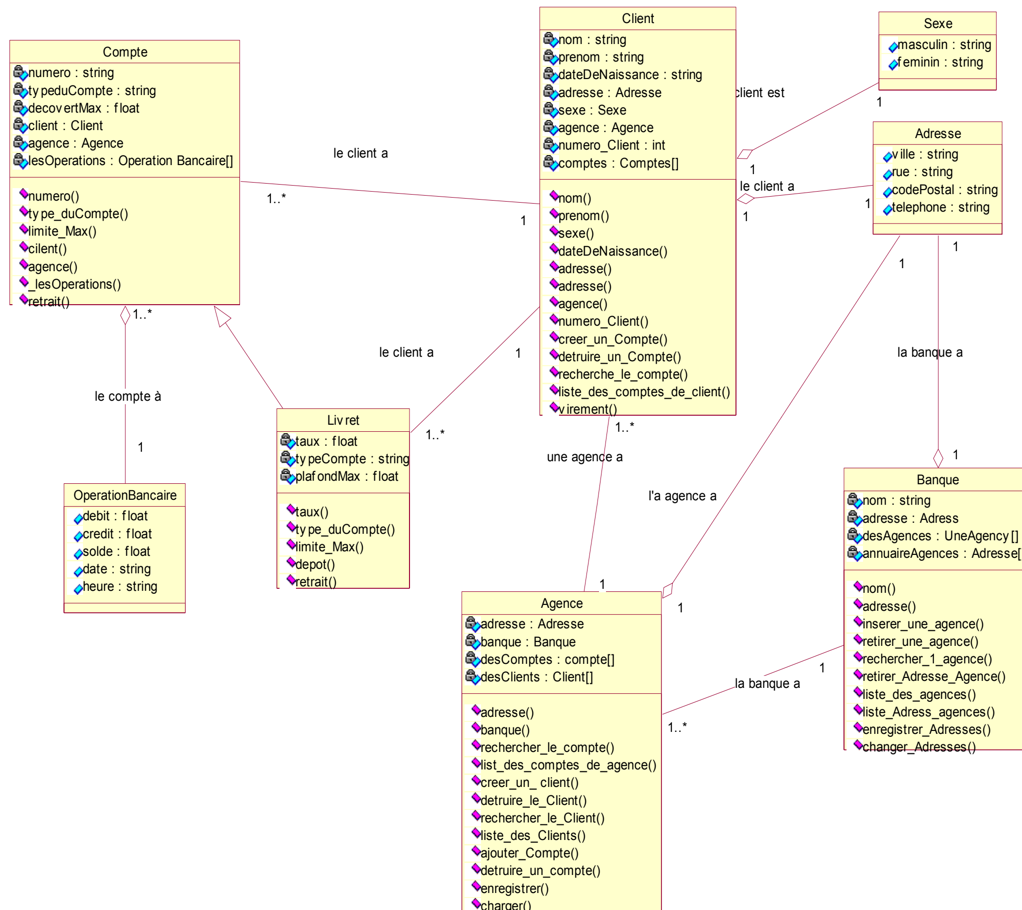
[SJ97] Johannes Samentinger. *Software Engineering with Reusable Components*, 1997.

[RA93] Robert S. Arnold. *Software Reengineering*, 1993.Chapter 11 pages 475-516.

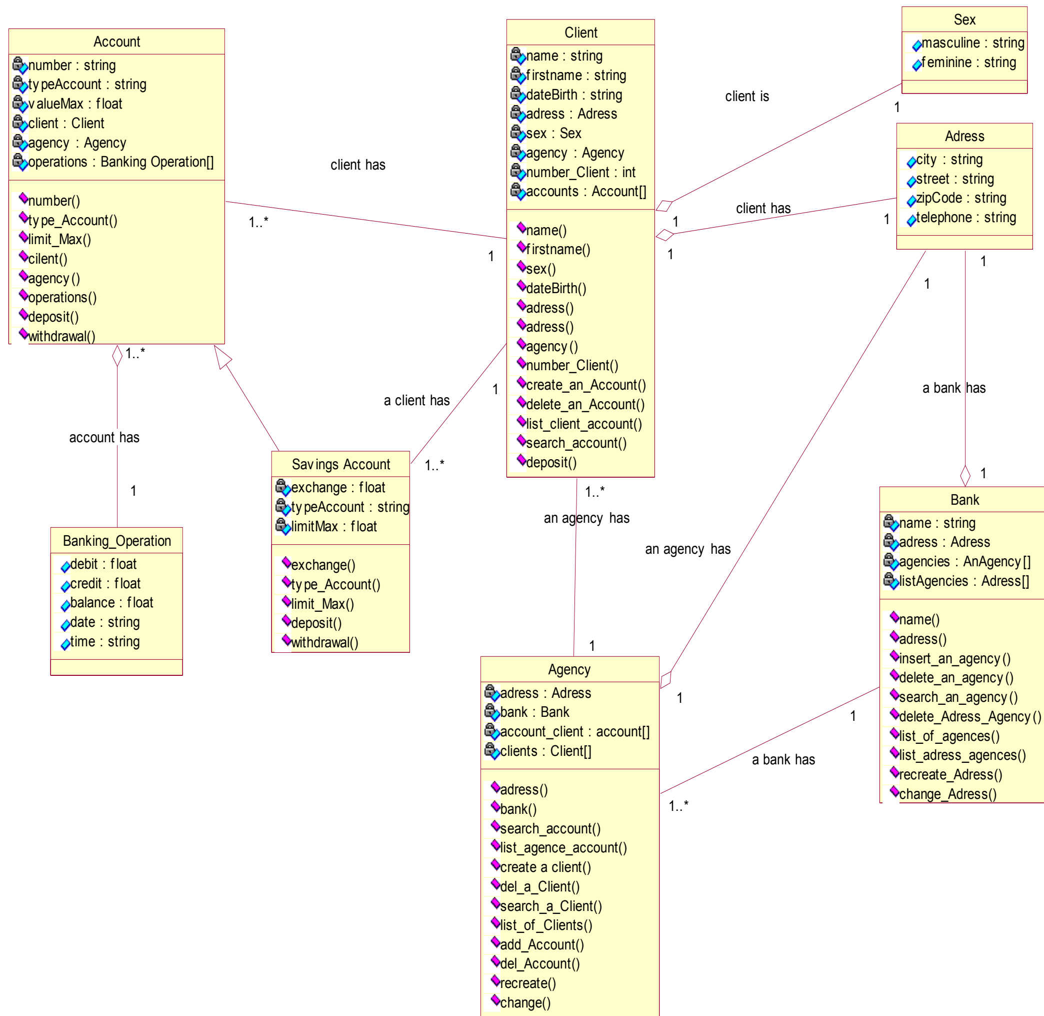
[WB97] Nelson H. Weiderman, John K. Bergey, Dennis B. Smith and Scott R. Tilley. *Approaches to Legacy System Evolution.Technical Report – Software Engineering Institute*, Carnegie Mellon University Pittsburgh, December 1997.

Appendix A : UML Model and Translation Terms

The original UML diagram (in French) provided in the Case Study documentation.



The same UML diagram in English.



Translation Terms

Class Compte : Account

Attributes	
numero	number
typeducompte	typeaccount
decouvertMax	valueMax
client	client
agence	agency
lesOperations	operations
Methods	
limite_Max	limit_Max
retrait	withdrawal

Classe Agence : Agency

Attributes	
adresse	address
banque	bank
desComptes_Clients	account_client
desClients	clients
Methods	
rechercher_le_Compte	search_account
liste_des_Comptes_de_agence	list_agency_account
creer_un_client	create_a_client
detruire_un_client	del_a_client
rechercher_le_Client	search_a_client
liste_des_Clients	list_clients
ajouter_Compte	add_account
detruire_Compte	del_account
enregistrer	recreate
charger	change

Classe Client : Client

Attributes	
nom	name
prenom	firstname
dateDeNaissance	dateBirthday
adresse	address
numero_Client	number_client
lesComptes	accounts
Methods	
creer_un_compte	create_an_account
creer_un_livret	create_saving_account
detruire_un_compte	delete_an_account
recherche_le_compte	search_an_account
liste_des_Comptes_des_clients	list_client_account
virement	deposit

Classe Livret : Savings Account

Attributes	
taux	exchange
typeducompte	typeaccount
plafondMax	limitMax
Methods	
depot	deposit
retrait	withdrawal

Classe Banque : Bank

Attributes	
nom	name
adresse	address
desAgences	agencies
annuaireAgences	listAgencies
Methods	
inserer_une_agence	insert_an_angency
retirer_une_agence	delete_an_agency
rechercher_une_agence	search_an_agency
retirer_Adress_Agence	delete_an_agency
enregistrer_Adresses	recreate_Addresses
charger_Adresses	change_Addresses

Classe Operation Bancaire : Banking Operation

Attributes

debit	Debit
credit	Credit
solde	Balance
date	Date
heure	Time

Classe Sexe : Sex

Attributes

masculin	Masculine
feminin	Feminine

Classe Adresse : Adress

Attributes

ville	city
rue	street
codePostal	zipCode
telephone	telephone

Associations Names

le client a	client has
le compte a	account has
une agence a	an agency has
le client est	client is
l'agence a	agency has
la banque a	bank has

Appendix B: Source Code

In this Appendix it will be found the main classes of source code of the Banking Application

- ✓ AgenceImpl.java
- ✓ BanqueImpl.java
- ✓ ClientImpl.java
- ✓ CompteImpl.java
- ✓ LivretImpl.java

```
//==========================================================================
// Implementation des méthodes de la classe Agence
//==========================================================================

//Pour la gestion des entrées et des sorties
import java.io.*;
import java.util.*;
import java.lang.String;
import bancaire.*;

public class AgenceImpl extends _AgenceImplBase implements Serializable{

	//==================================================================
	//Les variables d'instance
	//==================================================================

	private Adresse adresse_Agence;
	private Banque la_Banque;
	private Vector desComptes_Client= new Vector();
	private Vector desClients= new Vector();
	private int dernier_Numero_Client=0;

	//==================================================================
	//constructeur de la class ClientImpl
	//==================================================================

	public AgenceImpl(String ville, String rue,  String codePostal,
				String telephone, Banque banque) {

			adresse_Agence= new Adresse(ville, rue, codePostal,
telephone);
		la_Banque=banque;
	}
	public AgenceImpl() {
	}

	public void ajouter_Compte(Compte unCompte) {
		desComptes_Client.addElement(unCompte);
	}

	public Adresse adresse() {
			return adresse_Agence;
	}
```

```
	public Banque banque() {
			return la_Banque;
	}

	public Compte[] liste_des_Comptes_de_agence() {

			int i=0;
			Compte[] aux_desComptes_Client;
			aux_desComptes_Client= new
Compte[desComptes_Client.size()];

		while(i<desComptes_Client.size()) {

	aux_desComptes_Client[i]=(Compte)desComptes_Client.elementAt(i);
			i++;
		}
			return aux_desComptes_Client;
	}

	public Client[] liste_des_Clients() {

			int i=0;
			Client[] aux_desClients;
			aux_desClients= new Client[desClients.size()];
		while(i<desClients.size()) {
			aux_desClients[i]=(Client)desClients.elementAt(i);
			i++;
		}
			return aux_desClients;
	}

	public void detruire_un_Compte(String numero_Compte)
					throws ProblemeAgence {

			boolean ok=false;
			int i=0;

		while(((i<desComptes_Client.size()) && (ok==false))) {

	if((((Compte)desComptes_Client.elementAt(i)).numero()).equals(nume
ro_Compte)) {
				desComptes_Client.removeElementAt(i);
				ok=true;
			}
			i++;
			}
	}

	public Compte rechercher_le_Compte(String numero_Compte)
					throws ProblemeAgence {
			boolean ok=false;
			int i=0;

		while(((i<desComptes_Client.size()) && (ok==false))) {

	if((((Compte)desComptes_Client.elementAt(i)).numero()).equals(nume
ro_Compte)) {
				ok=true;
```

```
                        i--;
                        }
                i++;
                }

                if(ok==true) {
                return (Compte)desComptes_Client.elementAt(i);
                }
                else {
                        throw new ProblemeAgence("Le compte
"+numero_Compte+
                                                " n'exite pas !");

                }
        }

        public Client creer_un_Client(String nom_Client, String
prenom_Client,
                String date_Nais_Client, Sexe s, Adresse a) throws
ProblemeAgence {

                String nomClient=nom_Client.toUpperCase();
                        boolean ok=false;
                int resultat=0;
                        int i=0;

                dernier_Numero_Client= dernier_Numero_Client+1;

                        Client unClient= new ClientImpl(nomClient,
prenom_Client,
                         date_Nais_Client, s, a, this, dernier_Numero_Client);

                if(desClients.size()==0) {

                        desClients.addElement(unClient);
                }
                else {

                        while(((i<desClients.size()) && (ok==false))) {


        resultat=(((Client)desClients.elementAt(i)).nom()).compareTo(nomCl
ient);
                                if(resultat > 0) {
                                desClients.insertElementAt(unClient, i);
                                ok=true;
                                }
                        i++;
                        }
                        if(i>=desClients.size()) {
                                desClients.addElement(unClient);
                        }
                }
                        return unClient;
        }

        public void detruire_un_Client(int numero_du_Client) throws
ProblemeAgence {
```

```
                  boolean ok=false;
                  int i=0;

            while(((i<desClients.size()) && (ok==false))) {

      if((((Client)desClients.elementAt(i)).numero_Client())==numero_du_
Client) {
                        desClients.removeElementAt(i);
                        ok=true;
                  }
                  i++;
                  }

                  if(ok==false){
                  throw new ProblemeAgence("Le client n'exite pas !");

                  }

      }

      public Client rechercher_le_Client(int numero_du_Client) throws
ProblemeAgence {

                  boolean ok=false;
                  int i=0;

            while(((i<desClients.size()) && (ok==false))) {

      if((((Client)desClients.elementAt(i)).numero_Client())==numero_du_
Client) {
                        ok=true;
                        i--;
                  }
                  i++;
                  }

                  if(ok==true) {
                  return (Client)desClients.elementAt(i);
                  }
                  else {
                        throw new ProblemeAgence("Le client n'exite pas
!");
                  }
      }

      public void charger() {

            ObjectInputStream in;
            String lesClients= "Clients.age";
            Object unClient;
            int aux_numero;

            try {
                  in= new ObjectInputStream(new
FileInputStream(lesClients));
                  try {
                        while((unClient= in.readObject()) != null) {
                              desClients.addElement(unClient);
```

```
	aux_numero=((Client)unClient).numero_Client();
				if(aux_numero > dernier_Numero_Client) {
					dernier_Numero_Client=aux_numero;
				}
			}
		}
		catch(IOException ex0) {
			ex0.printStackTrace();
		}
		catch(ClassNotFoundException ex1) {
			ex1.printStackTrace();
		}
		finally {
			in.close();
		}
	}
	catch(IOException ex) {
		ex.printStackTrace();
	}

	Enumeration enum_Clients= desClients.elements();
	Compte [] les_Comptes_duClients=null;

	while(enum_Clients.hasMoreElements()) {


	les_Comptes_duClients=((Client)enum_Clients.nextElement()).liste_d
es_Comptes_de_client();
		int i=0;

		while(i<les_Comptes_duClients.length) {

	desComptes_Client.addElement(les_Comptes_duClients[i]);
			i++;
		}
	}
	}

	public void enregistrer() {

		Enumeration enum_Clients= desClients.elements();

		ObjectOutputStream out;
		String lesClients= "Clients.age";

		try {
			out= new ObjectOutputStream(new
FileOutputStream(lesClients));
			while(enum_Clients.hasMoreElements()) {
				out.writeObject(enum_Clients.nextElement());
			}
			out.close();
		}
		catch(IOException ex) {
			ex.printStackTrace();
		}
	}
}
```

```
//===========================================================================
// Implementation des méthodes de la classe Banque
//===========================================================================

import java.io.*;
import java.util.*;
import java.lang.String;
import bancaire.*;

public class BanqueImpl extends _BanqueImplBase {

	//=====================================================================
	//Les variables d'instance
	//=====================================================================

	private String nom;
	private Adresse adresse_Banque;
	private Vector desAgences= new Vector();
	private Vector annuaireAgences= new Vector();

	//=====================================================================
	//constructeur de la class BanqueImpl
	//=====================================================================

	public BanqueImpl(String nom, String ville, String rue,
				String codePostal, String telephone){
			this.nom=nom;
		adresse_Banque= new Adresse(ville, rue, codePostal,
telephone);
	}
	public BanqueImpl(){
	}

	public String nom() {
			return nom;
	}

	public Adresse adresse() {
			return adresse_Banque;
	}

	public void inserer_une_Agence(Adresse adresse, Agence agence)
throws ProblemeBanque {

			boolean ok=false;
		int i=0;

		while(((i<desAgences.size()) && (ok==false))) {


if(((((UneAgence)desAgences.elementAt(i)).adresse).ville).equals(adresse
.ville)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).rue).equals(adresse.r
ue)) {
```

```
if(((((UneAgence)desAgences.elementAt(i)).adresse).codePostal).equals(ad
resse.codePostal)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).telephone).equals(adr
esse.telephone)) {
                        ok=true;
                }
              }
            }
          }
          i++;
        }

              if(ok==false) {
              UneAgence uneAgence= new UneAgence(adresse, agence);
                    desAgences.addElement(uneAgence);
              }

              ok=false;
        i=0;

        while(((i<annuaireAgences.size()) && (ok==false))) {


if((((Adresse)annuaireAgences.elementAt(i)).ville).equals(adresse.ville)
) {

if((((Adresse)annuaireAgences.elementAt(i)).rue).equals(adresse.rue)) {

if((((Adresse)annuaireAgences.elementAt(i)).codePostal).equals(adresse.c
odePostal)) {

if((((Adresse)annuaireAgences.elementAt(i)).telephone).equals(adresse.te
lephone)) {
                        ok=true;
                }
              }
            }
          }
          i++;
        }

              if(ok==false) {
              annuaireAgences.addElement(adresse);
              }
      }

      public void retirer_une_Agence(Adresse adresse) throws
ProblemeBanque {

              boolean ok=false;
              int i=0;

        while(((i<desAgences.size()) && (ok==false))) {


if(((((UneAgence)desAgences.elementAt(i)).adresse).ville).equals(adresse
.ville)) {
```

```
if(((((UneAgence)desAgences.elementAt(i)).adresse).rue).equals(adresse.r
ue)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).codePostal).equals(ad
resse.codePostal)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).telephone).equals(adr
esse.telephone)) {
                        desAgences.removeElementAt(i);
                        ok=true;
                  }
                 }
                }
               }
               i++;
             }

                  if(ok==false){
                  throw new ProblemeBanque("L'agence n'est pas connectee
!");
               }
      }

      public Agence rechercher_l_Agence(Adresse adresse) throws
ProblemeBanque {

                  boolean ok=false;
                  int i=0;

            while(((i<desAgences.size()) && (ok==false))) {


if(((((UneAgence)desAgences.elementAt(i)).adresse).ville).equals(adresse
.ville)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).rue).equals(adresse.r
ue)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).codePostal).equals(ad
resse.codePostal)) {

if(((((UneAgence)desAgences.elementAt(i)).adresse).telephone).equals(adr
esse.telephone)) {
                        ok=true;
                        i--;
                  }
                 }
                }
               }
               i++;
             }

                  if(ok==true) {
                  return ((UneAgence)desAgences.elementAt(i)).agence;
                  }
                  else {
                        throw new ProblemeBanque("L'agence n'est pas
connectee !");
```

```
                }
      }

      public void retirer_Adresse_Agence(Adresse adresse) throws
ProblemeBanque {

                  boolean ok=false;
                  int i=0;

            while(((i<annuaireAgences.size()) && (ok==false))) {


if((((Adresse)annuaireAgences.elementAt(i)).ville).equals(adresse.ville)
) {

if((((Adresse)annuaireAgences.elementAt(i)).rue).equals(adresse.rue)) {

if((((Adresse)annuaireAgences.elementAt(i)).codePostal).equals(adresse.c
odePostal)) {

if((((Adresse)annuaireAgences.elementAt(i)).telephone).equals(adresse.te
lephone)) {
                        annuaireAgences.removeElementAt(i);
                        ok=true;
                 }
                }
               }
              }
             i++;
            }

                  if(ok==false){
                  throw new ProblemeBanque("L'adresse n'existe pas !");

             }

      }

      public UneAgence[] liste_des_Agences() {

                  int i=0;
                  UneAgence[] aux_desAgences;
                  aux_desAgences= new UneAgence[desAgences.size()];

            while(i<desAgences.size()) {
                  aux_desAgences[i]=(UneAgence)desAgences.elementAt(i);
                  i++;
            }
                  return aux_desAgences;
      }

      public Adresse[] liste_Adresse_Agences() {

                  int i=0;
                  Adresse[] aux_AnnuaireAgences;
                  aux_AnnuaireAgences= new
Adresse[annuaireAgences.size()];

            while(i<annuaireAgences.size()) {
```

```
      aux_AnnuaireAgences[i]=(Adresse)annuaireAgences.elementAt(i);
                  i++;
            }
                  return aux_AnnuaireAgences;
      }

      public void charger_Adresses() {

            ObjectInputStream in;
            String lesAdresses= "AnnuaireAgences.ban";

            Object uneAdresse;

            try {
                  in= new ObjectInputStream(new
FileInputStream(lesAdresses));
                  try {
                        while((uneAdresse= in.readObject()) != null) {
                              annuaireAgences.addElement(uneAdresse);
                        }
                  }
                  catch(IOException ex0) {
                        ex0.printStackTrace();
                  }
                  catch(ClassNotFoundException ex1) {
                        ex1.printStackTrace();
                  }
                  finally {
                        in.close();
                  }
            }
            catch(IOException ex) {
                  ex.printStackTrace();
            }
      }

      public void enregistrer_Adresses() {

            Enumeration enum_Adresses= annuaireAgences.elements();

            ObjectOutputStream out;
            String lesAdresses= "AnnuaireAgences.ban";

            try {
                  out= new ObjectOutputStream(new
FileOutputStream(lesAdresses));
                  while(enum_Adresses.hasMoreElements()) {
                        out.writeObject(enum_Adresses.nextElement());
                  }
                  out.close();
            }
            catch(IOException ex) {
                  ex.printStackTrace();
            }
      }

}
```

```
//=======================================================================
// Implementation des méthodes de la classe Client
//=======================================================================

//Pour la gestion des entrées et des sorties
import java.io.*;
import java.util.*;
import bancaire.*;
import org.omg.CORBA.*;

public class ClientImpl extends _ClientImplBase implements Serializable{

	//=================================================================
	//Les variables d'instance
	//=================================================================

	private String nom;
	private String prenom;
	private String dateDeNaissance;
	private Sexe sexe;
	private Adresse adresse;
	private Agence agence;
	private int numero_Client;
	private Vector lesComptes=new Vector();

	//=================================================================
	//constructeur de la class ClientImpl
	//=================================================================

	public ClientImpl(String nom_Client, String prenom_Client,
			String dateDeNaissance_Client, Sexe s,
			Adresse a, Agence ag, int le_numero_Client) {

		nom=nom_Client;
		prenom=prenom_Client;
			dateDeNaissance=dateDeNaissance_Client;
		sexe=s;
		adresse=a;
		agence=ag;
		numero_Client=le_numero_Client;
	}

	public ClientImpl() {
	}

	public String nom() {
		return nom;
	}

	public String prenom() {
			return prenom;
	}

	public String dateDeNaissance() {
			return dateDeNaissance;
	}
```

```
      public Sexe sexe() {
                  //remarque:
                  //Sexe sexe= Sexe.from_int(Sexe.FEMININ); //Faux
                  //Sexe sexe= Sexe.from_int(1); //Vrai
                  //Sexe sexe= Sexe.FEMININ; //vrai
                  return sexe;
      }

      public Agence agence(){
                  return agence;
      }

      public Adresse adresse(){
                  return adresse;
      }

      public void adresse(Adresse adresse){
                  this.adresse=adresse;
      }

      public int numero_Client(){
                  return numero_Client;
      }

      public Compte creer_un_Compte(String numero, Client client,
                               Agence agence, float solde){

                  Compte nouveau_Compte= new CompteImpl(numero, client,
                                          agence, solde);
                  lesComptes.addElement(nouveau_Compte);
                  agence.ajouter_Compte(nouveau_Compte);
                  return nouveau_Compte;
      }

      public Livret creer_un_Livret(String numero, Client client,
                        Agence agence, float solde, float taux){


                  LivretImpl nouveau_livret= new LivretImpl(numero,
client,
                                    agence, solde, taux);
            _LivretImplBase_tie nouveau_livret_tie=new
_LivretImplBase_tie(nouveau_livret);

                  lesComptes.addElement(nouveau_livret);
                  agence.ajouter_Compte(nouveau_livret);
                  return (Livret)nouveau_livret_tie;
      }

      public void detruire_un_Compte(String numero_Compte)
                                    throws ProblemeClient {

                  boolean ok=false;
                  int i=0;

            while(((i<lesComptes.size()) && (ok==false))) {
```

```
      if((((Compte)lesComptes.elementAt(i)).numero()).equals(numero_Comp
te)) {
                        lesComptes.removeElementAt(i);
                        ok=true;
                  }
                  i++;
            }

                  if(ok==false){
                  throw new ProblemeClient("Le compte avec le numero
"+numero_Compte+
                                                " n'exite pas !");

                  }
      }

      public Compte recherche_le_Compte(String numero_Compte)
                               throws ProblemeClient{

                  boolean ok=false;
                  int i=0;

            while(((i<lesComptes.size()) && (ok==false))) {

      if((((Compte)lesComptes.elementAt(i)).numero()).equals(numero_Comp
te)) {
                        ok=true;
                  }
                  i++;
            }

                  if(ok==true) {
                  return (Compte)lesComptes.elementAt(i);
                  }
                  else {
                  throw new ProblemeClient("Le compte avec le numero
"+numero_Compte+
                                                " n'exite pas !");

                  }
      }

      public Compte[] liste_des_Comptes_de_client(){

                  int i=0;
                  Compte[] aux_desComptes;
                  aux_desComptes= new Compte[lesComptes.size()];

            while(i<lesComptes.size()) {
                  aux_desComptes[i]=(Compte)lesComptes.elementAt(i);
                  i++;
            }
                  return aux_desComptes;
      }

      public void virement(String numero_Compte1, String numero_Compte2,
                        float montant, StringHolder message1,
                        StringHolder message2) throws ProblemeClient {
```

```
            boolean okj=false;
            boolean okk=false;
            StringHolder aux_message1=new StringHolder();
            StringHolder aux_message2=new StringHolder();
            String aux_Message="OK";
            message2.value=new String(aux_Message);
            message1.value=new String(aux_Message);
            int i=0;
            int j=0;
            int k=0;

            while(((i<lesComptes.size()) && !((okj!=false) &&
(okk!=false)))) {

      if((((Compte)lesComptes.elementAt(i)).numero()).equals(numero_Comp
te1)) {
                        j=i;
                        okj=true;
                        }

      if((((Compte)lesComptes.elementAt(i)).numero()).equals(numero_Comp
te2)) {
                        k=i;
                        okk=true;
                        }
                  i++;
            }

            if((okj==true) && (okk==true)) {

                  OperationBancaire [] lesOperations1;
                  OperationBancaire [] lesOperations2;


      lesOperations1=((Compte)lesComptes.elementAt(j))._lesOperations();
                  OperationBancaire derniereOperation1=

      (OperationBancaire)lesOperations1[lesOperations1.length-1];
                  float dernierSolde1=derniereOperation1.solde;
                  float solde1=dernierSolde1-montant;


      lesOperations2=((Compte)lesComptes.elementAt(k))._lesOperations();
                  OperationBancaire derniereOperation2=

      (OperationBancaire)lesOperations2[lesOperations2.length-1];
                  float dernierSolde2=derniereOperation2.solde;
                  float solde2=dernierSolde2+montant;

                  if(solde1>= -
(((Compte)lesComptes.elementAt(j)).limite_Max())) {
                        if(solde2<=
(((Compte)lesComptes.elementAt(k)).limite_Max())) {


      ((Compte)lesComptes.elementAt(j)).retrait(montant, aux_message1);
                              String aux1=""+aux_message1.value;
                              message1.value=new String(aux1);
```

```
	((Compte)lesComptes.elementAt(k)).depot(montant, aux_message2);
	                        String aux2=""+aux_message2.value;

	                        message2.value=new String(aux2);

	                    }
	                    else {

	aux_Message=""+((Compte)lesComptes.elementAt(k)).limite_Max();
	                        message2.value=new String(aux_Message);
	                    }
	                }
	                else {

	aux_Message=""+((Compte)lesComptes.elementAt(j)).limite_Max();
	                        message1.value=new String(aux_Message);
	                }
	            }
	            else {
	                    if(j==0) {
	                    throw new ProblemeClient("Le compte avec le
numero "+numero_Compte1+
	                                            " n'exite pas !");
	                    }
	                    if(k==0) {
	                    throw new ProblemeClient("Le compte avec le
numero "+numero_Compte2+
	                                            " n'exite pas !");
	                    }
	            }
	}

	public void interets_Bancaires() {

	        int i=0;
	        float aux_Taux=0;
	        float aux_Solde=0;
	        float aux_Interet=0;
	        OperationBancaire [] lesOperations;

	        while(i<lesComptes.size()) {

	if((((Compte)lesComptes.elementAt(i)).type_duCompte()).equals("LIV
RET")) {

	                    StringHolder message= new StringHolder();


	aux_Taux=((LivretImpl)lesComptes.elementAt(i)).taux();

	lesOperations=((Compte)lesComptes.elementAt(i))._lesOperations();
	                    aux_Solde=lesOperations[(lesOperations.length)-
1].solde;
	                    aux_Interet=aux_Solde*aux_Taux;

	((Compte)lesComptes.elementAt(i)).depot(aux_Interet, message);
	                }
```

```
                  i++;
            }
      }
}

//=========================================================================
// Implementation des méthodes de la classe Compte
//=========================================================================

//Pour la gestion des entrées et des sorties
import java.io.*;
import bancaire.*;
import java.util.*;
import java.util.Calendar;
import org.omg.CORBA.*;

public class CompteImpl extends _CompteImplBase implements Serializable{

      //===================================================================
      //Les variables d'instance
      //===================================================================

      protected String numero;
      private String typeduCompte;
      private float decouvertMax;
      protected Client client;
      protected Agence agence;
      protected Vector lesOperations=new Vector();

      //===================================================================
      //constructeur de la class ClientImpl
      //===================================================================

      public CompteImpl(String numero, Client client, Agence agence,
float decouvertMax){
            this.numero=numero;
            typeduCompte="COMPTE";
            this.client=client;
            this.agence=agence;
            this.decouvertMax=decouvertMax;
            GregorianCalendar date=new GregorianCalendar();
            String heure="à "+date.get(Calendar.HOUR_OF_DAY)+":"
                        +date.get(Calendar.MINUTE);
            String laDate="le "+date.get(Calendar.DAY_OF_MONTH)+":"
                        +date.get(Calendar.MONTH)+":"
                        +date.get(Calendar.YEAR);
            OperationBancaire uneOperation=
                  new OperationBancaire(0,0,0,laDate,heure);
            lesOperations.addElement(uneOperation);
      }
      public CompteImpl(){
      }

      public String numero() {
            return numero;
      }

      public String type_duCompte() {
            return typeduCompte;
```

```
    }

    public float limite_Max() {
          return decouvertMax;
    }

    public Client client() {
          return client;
    }

    public Agence agence() {
          return agence;
    }

    public OperationBancaire[] _lesOperations() {

                int i=0;
                OperationBancaire[] aux_OperationBancaire=
                new OperationBancaire[lesOperations.size()];

          while(i<lesOperations.size()) {
                aux_OperationBancaire[i]=
                      (OperationBancaire)lesOperations.elementAt(i);
                i++;
          }
                return aux_OperationBancaire;
    }

    public void depot(float montant, StringHolder message) {

          OperationBancaire derniereOperation=
                (OperationBancaire)lesOperations.lastElement();
          float dernierSolde=derniereOperation.solde;
          float solde=dernierSolde+montant;
          GregorianCalendar date=new GregorianCalendar();

          String heure="à "+date.get(Calendar.HOUR_OF_DAY)+":"
                      +date.get(Calendar.MINUTE);
          String laDate="le "+date.get(Calendar.DAY_OF_MONTH)+":"
                      +date.get(Calendar.MONTH)+":"
                      +date.get(Calendar.YEAR);
          OperationBancaire uneOperation=
                new OperationBancaire(0,montant,solde,laDate,heure);

          lesOperations.addElement(uneOperation);
          message.value=new String("OK");
    }

    public void retrait(float montant, StringHolder message) {

          OperationBancaire derniereOperation=
                (OperationBancaire)lesOperations.lastElement();
          float dernierSolde=derniereOperation.solde;
          float solde=dernierSolde-montant;
          GregorianCalendar date=new GregorianCalendar();
          String lemessage;

          if(solde>=-decouvertMax) {
```

```
                  String heure="à "+date.get(Calendar.HOUR_OF_DAY)+":"
                        +date.get(Calendar.MINUTE);
                  String laDate="le "+date.get(Calendar.DAY_OF_MONTH)+":"
                        +date.get(Calendar.MONTH)+":"
                        +date.get(Calendar.YEAR);
                  OperationBancaire uneOperation=
                        new
OperationBancaire(montant,0,solde,laDate,heure);

                  lesOperations.addElement(uneOperation);
                  message.value=new String("OK");
            }
            else {
                  String aux_Message=""+decouvertMax;
                  message.value=new String(aux_Message);
            }
      }
}

//=========================================================================
// Implementation des méthodes de la classe Livret
//=========================================================================

//Pour la gestion des entrées et des sorties
import java.io.*;
import bancaire.*;
import org.omg.CORBA.*;

public class LivretImpl  extends CompteImpl
                        implements LivretOperations, Serializable{

      //===================================================================
      //Les variables d'instance
      //===================================================================

      private String typeduCompte;
      private float plafondMax;
      protected float taux;

      //===================================================================
      //constructeur de la class ClientImpl
      //===================================================================

      public LivretImpl(String numero,Client client,Agence agence,
                              float plafondMax, float taux){
            super(numero, client, agence, 0);
            this.plafondMax=plafondMax;
            typeduCompte="LIVRET";
            this.taux=taux;
      }

      public String type_duCompte() {
            return typeduCompte;
      }

      public float limite_Max() {
            return plafondMax;
      }
```

```
	public float taux() {
		return taux;
	}

	public void depot(float montant, StringHolder message) {

		OperationBancaire[] lesOperations = super._lesOperations();
		float dernierSolde=lesOperations[lesOperations.length-
1].solde;
		float solde=dernierSolde+montant;

		if(solde<=plafondMax) {
			super.depot(montant, message);
		}
		else {
			String aux_Message=""+plafondMax;
			message.value=new String(aux_Message);
		}
	}

	public void retrait(float montant, StringHolder message) {

		OperationBancaire[] lesOperations = super._lesOperations();
		float dernierSolde=lesOperations[lesOperations.length-
1].solde;
		float solde=dernierSolde-montant;

		if(solde>=0) {
			super.retrait(montant, message);
		}
		else {
			String aux_Message=""+0;
			message.value=new String(aux_Message);		}
	}
}
```