NEWCASTLE UNIVERSITY
SCHOOL OF COMPUTING SCIENCE

Exploiting the Architectural Characteristics of Software Components to Improve Software Re-use

by
Basem Yousef Alkazemi

PhD Thesis

January 2009

Abstract

Software development is a costly process for all but the most trivial systems. One of the commonly known ways of minimizing development costs is to re-use previously built software components. However, a significant problem that source-code re-users encounter is the difficulty of finding components that not only provide the functionality they need but also conform to the architecture of the system they are building. To facilitate finding re-usable components there is a need to establish an appropriate mechanism for matching the key architectural characteristics of the available source-code components against the characteristics of the system being built. This research develops a precise characterization of the architectural characteristics of source-code components, and investigates a new way to describe how appropriate components for re-use can be identified and categorized.

Acknowledgment

I would like to express my gratitude to several people who have supported me over the period of doing this work. Firstly my supervisor Professor Peter Lee, I would like to thank him for his continual support and encouragement, and also for his effort in reviewing my thesis. Thanks also to the supervisory committee members Dr Christina Cacek and Dr John Fitzgerald for their constructive inputs and directions.

Secondly, I would like to thank all members of staff in the school of computing science. To my internal examiner, Dr Neil Speirs, and my external examiner, Professor David Budgen, for a fruitful viva session. To Shirley Craig, thank you for helping me finding many relevant references in this thesis.

Thirdly, I would like to thank my friends Dr Saad Almajnoni, Mr Christiaan Lamprecht, and Mr Carl Gamble for their encouragements and opinions.

Finally, I would like to thank my wife, Huda Alaama, for being patient and supportive all along. To my brothers, Khalid and Majed, and to my sisters, Amel, Dalal, and Ghada for calling me regularly every occasion from Saudi Arabia. To my respected mother in law, Afaf Khayat, for the prayers. This thesis is for all of you. To may parents who passed away years ago (may God be pleased with them), I hoped you were alive to witness this achievement.

The work reported here would have not been possible without the grants from Umm Al-Qura University.

Table of Content

List of Figures

List of Tables

Chapter 1 - Introduction

1.1 Vision of the Future

Adam is a software engineer who works for the *EasySoftMicrosystems* company, developing graphical environments for software development. He has been assigned a task of building a new and novel integrated development environment (IDE) that helps software developers to build their systems using graphical representations rather than pure source-code writing. So, developers can construct their systems simply by selecting the right *components* from a list, customizing as required, and dropping them into the system being developed.

Adam has gone through the necessary processes to build the system, starting from collecting the requirements and writing the specification of the new system, to establishing the overall design of the system intended to be built. The design identified the basic building blocks of the system as being composed of several components such as editors, source-code generator, compiler, builder and debugging tools, in addition to a shared library (i.e. a repository system) that stores and organises re-usable components. After Adam documented the design of the new system he decided to start implementing the real working IDE system.

While Adam was programming the various parts of the system he came to a point at which he needed to add a component to perform the parsing functionality to the system being built, so he started looking for something to re-use in the hope of saving his time and effort. He searched for some within his company's repository system but couldn't find anything re-usable for his system. Then he started looking externally to find a repository where the required component might be found.

Adam searched several repositories that offer different kinds of components, some free of charge while others were subject to some cost. Finally, he came across a repository system called *GeniusComponents* that offered a wide range of components freely. He tried the repository by providing some information to search for matching components. The repository system started searching for the component that Adam was looking for and returned some results. The repository listed a parser that provided the exact functionality that Adam was looking for. Adam was very pleased that he managed to get the component he needed.

However, there were two problems with the found component. Firstly, it was written in FORTRAN but what Adam needed was a component written in Java. The second problem, which seemed more challenging, was that the found component was a part of another system, so it must be extracted from its original system before re-using it in Adam's system. Adam was very concerned about how the extracted component might impact on his own system as his experience suggested that the component might cause system failure if, for example, some of its required dependencies had been missed during extraction. Adam also wondered how he could modify the component to fit into his system as there was the major language dissimilarity. His frustration was rising as he started to believe that building one's own component is much easier than re-using one, as found components often cannot be re-used without major modifications. What he got from the repository demonstrated that. The modification itself was hard to achieve as it required a thorough understanding of the component's architectural aspects in addition to its functional aspects.

While Adam was looking sadly at the listed component wondering what to do, he noticed something flashing in the top right corner on the screen where the component he found was listed. It was a button labelled with the word "*modify*". Adam clicked on that button and what he got was an extensive list of automatic conversions that could be applied. One option was to convert a component into a stand-alone application. Adam tried that option hoping that it could extract the component he needed and the extracted component would work satisfactorily when converted. The repository system advised that the conversion was completed successfully. So, Adam obtained an application that provided the functionality that his system required, however, the application was still written in FORTRAN, hence still not re-usable in his system as the application must to be connected to his system prior to being able to re-use it as a component. Adam found another option in the list of possible conversions that converted FORTRAN applications to Java applications, so without any further thinking, he clicked on that option. The result was stunning; the application was converted, somehow, to Java.

The next step was to wrap the application to fit as a component into Adam's system. An option for accomplishing that conversion was also provided by the repository system, so Adam simply selected that option and, somehow, the application was wrapped as a component that satisfied his system's requirements. Adam was happy that he had finally got what he needed but he was worried that the modification might have broken the component. Adam found another listed option in the repository to automatically check the component to ensure its validity against some specified requirements. There were numbers

of pre-defined requirements of other systems recognized by the repository that he could use if his system requirements were similar, but Adam's system was novel, hence he needed to supply his system's requirements to the repository in order to be utilized by the checking tool. The repository system assisted Adam in providing the required details of his system's requirements by asking him about the different characteristics of his system. So, Adam specified his system's requirements to check the component against and tried that test. The result was positive indicating that the converted component was fine and it was conforming to the requirements of his system. Adam then requested that the component be delivered to him from the repository so that he could re-use it in his system, and the repository system successfully delivered it. Adam noticed that the delivered component also came with a help file describing everything he needed to know in order to utilize the component.

So finally Adam obtained the component that fitted perfectly into his system, provided the required functionality and satisfied his system's requirements. Adam was very pleased that he found such a super repository system that assisted him to obtain re-usable components and encouraged him to keep re-using components rather than building them from scratch.

1.2 Thesis Context

It is obvious from the above "vision" story that a repository system called *GeniusComponents* has helped Adam to build his intended system (i.e. the novel IDE). Without such a repository system, re-use would have been very unlikely and Adam would have been faced with writing a new component from scratch.

The above described repository system is an example of an ideal repository that every software re-user is hoping to find. That repository would speed up the development process, save time and effort, and encourage the software engineer to practice re-use. However, the state of current repository systems is in reality far behind such an envisaged level of support. Today's repository systems have many limitations that, in one way or another, discourage software development from re-use. Therefore this thesis is primarily motivated by the fact that the current support provided by repository systems is not sufficient and may hinder re-use, and the research to be described is a step towards achieving the ideal case of software re-use described in the story above. Therefore, the hypothesis of this research is that re-use would be highly advantageous for software development, however the inadequate support for re-use is preventing these advantages

from being realised. The corresponding aim and objectives of the research to be described in this thesis are listed in section 1.10.

1.3 Software Re-use

Software re-use is commonly described as the process of re-using previously built components to drive the building of a new system. So a system might be built from re-usable components instead of reinventing components every time a new system is needed. Re-using previously generated components has the potential to significantly lower the cost of developing new software systems, speed the development process, and improve the quality of the final software product [138].

At a first glance, re-using components might seem a natural approach that software developers should follow, comparable to the well-established practice of re-use when building an electronic system. However, it seems that software developers usually prefer to build their own components rather than harnessing ones generated previously. This reluctance to re-use components is caused by many reasons, often based on the perspective that the developer has. Software developers can be classed into three groups with respect to re-using components:

i) those who would never re-use components;

ii) those who have never thought about re-use; and

iii) those who would exploit re-use if the process was better supported.

The first group of developers may not trust components that were generated by others as they believe those components are vulnerable to unauthorized access, or they might think that the available components are inaccurate, or of inappropriate quality to re-use. Others might prefer to take the intellectual challenge of solving the problem by themselves rather than re-using existing solutions. The second group of developers may not be aware of the availability of re-usable components, so they always work on generating their own components without considering any re-use attempt. The third group of developers might be aware that re-usable components are available somewhere, but the problem that distracts them from re-use is the difficulty of obtaining them. The first group of developers are not interested in re-use in the first place. The second group may or may not be interested. However, the third group of developers are interested in practising re-use as they are aware of its advantages, but they suffer from not being able to obtain the

components easily. This research is targeting primarily the third group of developers by addressing the difficulties they currently encounter.

Obtaining re-usable components involves the activities of searching, finding, and retrieving components from places where they are stored. Components stores are commonly known as repositories. If components are not organized and represented in a repository in a precise manner then obtaining them will not be trivial. Organizing components for re-use is a significant feature that a repository system must exhibit to support re-use.

1.4 Repository Systems

A repository system in very general terms is a place where data are stored and maintained for subsequent access. In software development, a repository is a commonly known place where re-usable components can be found. Practising re-use in software development requires the availability of a repository system to help obtaining components for re-use.

Deciding whether a repository system is really necessary or not is, primarily, based on the number of components available for re-use [43]. Small numbers of components (e.g. 10 components) may not need a complex repository system, a shared folder could be sufficient. However, if the number of components is large then a properly organized repository system is essential.

Some repository systems are built privately within organizations and do not provide external access; only employees within that organization can access the repository. Other repository systems are in the public domain and open to more general access. Repository systems may vary from one organization to another, based on the needs that organizations want to fulfil. However, all repository systems must exhibit some common key characteristics including:

- **Organize components**: if the number of components within a repository is large then a mechanism to organize components systematically is required.
- **Mechanisms to facilitate searching and browsing**: a repository must employ a mechanism to search for and locate components of interest. It is desirable also to provide browsing mechanisms.
- **Adding new components**: a repository system must be able to accept and organize new components.

- **Provide descriptions for components:** every component in a repository must be described by some means (e.g. textual description) to reflect the main purpose of generating them in the first place. The provided descriptions can be utilized to help re-users search for their desired components.
- **Provide a version control mechanism**: components in a repository system may differ in versions. So a mechanism to control versions and differences between versions is useful in a repository system.

Before further details of software repository systems are examined, it is appropriate to examine the nature of the re-usable software components that may be stored in repository systems. Many types of components can be kept in a repository system; these are discussed briefly in the next section.

1.5 Re-usable Components

Several types of components are produced throughout the process of software development: from requirements documents, specification documents, designs, plans, through to source code, binary libraries, and even test harnesses and test data. These types of component can be classified as high-level and low-level. High-level components include all components produced before the implementation stage of a software system (e.g. requirements, specification, and design). The low-level components are those produced in the implementation stage and in the stages thereafter (i.e. testing and maintenance).

Theoretically, many components produced in a software development process could be re-used. Specification documents, for example, could be re-used in a development process that intended to build a similar system with some added functionality. Design patterns [55], for instance, are a type of component that can be re-used in software development within a design stage. Design patterns describe solutions to recurrent problems at the design level. So, one might re-use design patterns to help overcome system design problems. Source code can also be re-used from an existing system that do similar functionality as the one needed by re-users. Re-using source code can reduce the overhead of building a system as re-users will not need to write source code from scratch, but re-use the code that is already available to them.

Re-using any components produced during the software development process could be beneficial, but the main focus in this research is in re-using source-code components. The term software components will now be used to mean source-code components throughout

the thesis. The term *artefact* will be used to refer to the different types of components in general (e.g. requirements, specification, design). The term component will be used to refer conceptually to a part of a system (e.g. library of functions, web service, design element, source-code component).

1.6 Software Component Repository Systems

Obtaining software components from a software repository system is one of the obstacles encountered by re-users [138]. This problem is caused by the lack of sufficient categorization for software components inside repository systems. Obviously, software components cannot be re-used unless they can be obtained, and they need to be found with less effort than it would take to implement the desired functionality from scratch.

Of course, currently available repositories employ some mechanisms to organize the software components they contain. However, the categories used to organize software components are very abstract in nature and can cause a large number of results to be listed under a single category. For example, in *sourceforge.net* (one of the leading open-source repositories) 3,351 open-source projects were listed on 12/2007 under the *utilities* category. One obvious problem indicated by this example is that the number of potentially re-usable candidates is large because the categorisation is too general, which would require the re-user to spend extensive time in locating suitable components for re-use. Another problem is that this category (i.e. *utilities*) is very abstract and may not reflect any useful meaning to re-users. This indicates that unless re-users are aware of the specific categories that classify software components, it can be extremely hard to locate re-usable source code, especially given the limited searching support provided by the current open-source repositories (usually limited to free-text searching and category browsing). The existence of sufficient and effective ways to group and organize software components within software repositories is a key aspect that eases finding source code, hence encouraging their re-use.

Precise categorization of software components inside a repository system is necessary to achieve logical structuring and organizing of source-code components within a repository system, and hence to facilitate re-users in obtaining them. However, categorizing source-code components is not always trivial in software because the discriminating characteristics of source code identities are ill defined compared to other engineering disciplines. For example, in electronics, a simple transistor can be identified by its I/O pins. Whenever an electronic component is described as having an Emitter, a Collector, and a

Base pins then it can be categorized as a transistor. This degree of precision, that specifies what a component "must have" to fit within a category, is not obvious in software, and there appears to be no standard mechanism for describing their characteristics. For example, what are the distinguishing characteristics by which components can be identified as JavaBeans? Or what are the characteristics that a parser component must have to achieve its parsing behaviour? Thus a means for identifying the characteristics that may allow discrimination of components, and hence form the basis for a categorization mechanism that a repository can use, is needed.

Prior work (which will be discussed in Chapter 3) has attempted to establish a categorization for software components, although mostly based on trying to capture a component's functional characteristics. Categorizing components by their functional characteristics is one of the obvious ways to organize software components in a repository, as functionality is likely to be the characteristic that re-users consider first, when searching for software components to re-use. Re-users might further apply filtering mechanisms to identify the most relevant components, but their primary searching criteria are likely to be based on functional characteristics. So, precise descriptions of component functionality are desirable in order to establish an organizational basis for a repository's categorization mechanism.

However, capturing and understanding complex functionality of software components is a significant challenging to software developers [48]. For high-level artefacts (e.g. specification) there are several attempts to capture descriptions of functionality in a formal manner, but they are not in widespread use. For source-code components, there is no appropriate way of obtaining key functional characteristics from source code. In addition, understanding the functionality of software components requires the presence of good documentation, whereas having well-written documentation is not guaranteed all the time especially in the case of software components in open-source repositories. Moreover, finding components that provide the required functionality is not enough alone to re-use the found components successfully as their architecture is also of importance.

One requirement of an ideal repository system is that it should depend primarily on identifying components' characteristics from the source code of the deposited components to facilitate their easy finding and subsequent re-use. The characteristics that can be identified from a component's source code relate to the functionality that the component provides and also the architecture it conforms to. While capturing key functionality from

the source code is not possible at the moment, a first step towards achieving the optimal solution that the ideal repository system can provide, however, is trying to capture a component's architecture from its source code. This research assumed that software components can be re-usable into a re-user's system if they provide the required functionality and also conform to the system's architecture at hand. This assumption is motivated by the fact that re-users might find, somehow, components that provide the right functionality (fit at functional level) and re-use them in their system but soon they discover that components raised compile-time, link-time, or run-time errors due to missing some required methods, for instance, or being written in a different programming language than the one required by a re-user, and hence caused an architectural mismatch (i.e. not fit at the architectural level). Recall the "vision" story at the beginning, Adam had found the required "parser" component but he could not re-use it directly as it needed some modification in order to satisfy his system's architectural requirements (e.g. converted from FORTRAN to Java).

Satisfying the functional requirements of a system is named as *functional fit* while satisfying the architectural requirements of a system is referred to as *architectural fit*. Achieving both functional fit and architectural fit is referred to as the perfect fit. Components can perfectly fit into a re-user's system if they provide the required functionality (i.e. functional fit) and also conform to system's architecture (i.e. architectural fit). This separation between functional fit and architectural fit can help the more accurate understanding of a component's re-usability and, as a result, allow re-users to plan their modifications to components. For example, if developers are aware of what is required to modify a component conforming to the "Net-Beans" architecture to an "Eclipse" architecture then that would help re-using the component, as a result, the component can be fit architecturally after applying the necessary modification, and consequently is considered re-usable. An additional advantage of the separation between functionality and architectural aspects is that a tool might be used to perform the modification automatically either at the architectural level or at the functional level as both levels of fit are described precisely. This is what has been described in the story at the beginning when a FORTRAN component needed to be modified and converted to a component written in Java instead, also converting a component into a stand alone application.

While satisfying the functional fit is not possible at the moment, this research concerns the characteristics of the architectural fit as the distinguishing characteristics that an ideal

repository system can utilize in order to identify and organize components automatically for subsequent re-use. The research also concerns addressing the architectural fit of software components as a starting point towards achieving the complete solution, in the future, by which components can perfectly fit into systems where they are re-used. The architecture that a component conforms to is named as its architectural type. The architectural type defines the characteristics that a component must satisfy in order to fit architecturally into a system. For example, one of the characteristics of a Java application architectural type is that it must have a method called "`public static void main()`". The characteristics that are defined by an architectural type are distinct and different from one architectural type to another. As a result, it is useful to exploit the characteristics defined by an architectural type to build the basis for organizing software components inside a repository system. Therefore, addressing the architectural fit was considered as the topic in this research while the functional fit is left for future work.

1.7 Aim and Objectives

This research is aimed to address some of the problems that hinder components re-use, and investigate potential solutions to optimise the support that can be provided to components re-users. With that aim in mind the set of objectives for this research are:

1. To identify the design of an ideal repository system.
2. To investigate the possibility of characterizing components at the source-code level.
3. To uncover the architectural characteristics and dimensions that correspond to fitting components architecturally into a system in order to address the use of these characteristics within a repository.
4. To propose an approach, namely ArchInt, that formalizes the architectural interface in a low level of abstraction that reflects the precise characteristics of software components.
5. To investigate the applicability of ArchInt in automating the process of categorizing and modifying software components.

1.8 Thesis Outline

The remainder of this thesis is organized into six chapters as follows. Chapter 2 presents the background work conducted in this research to set the context for the thesis. The chapter describes the potential problems encountered by components re-users and the main reason behind the presence of these problems and how they can be solved. The overall context of the thesis is described towards the end of the chapter.

Chapter 3 presents a classification and analysis of the main characteristics of the existing software repository systems – forming taxonomy of software repository systems. Using the taxonomy, a survey of software repository systems is provided. Toward the end of Chapter 3, an analysis of the current repository systems is presented which provides the requirements and justification for proposing the approach of this research. Also a discussion about the components categorizations available in the literature is given in Chapter 3.

Chapter 4 draws upon related work into supporting software re-use, and describes a number of use-cases to gather the requirements of the approach proposed in this research. The chapter then proceeds and discusses concepts related to component fit and identifies the characteristics of architectural types.

Chapter 5 introduces a prototype of a specification language namely ArchInt that formalizes some of the characteristics of architectural interface. The chapter also presents some studies for evaluating the generated ArchInt prototype and also to give a spin on the overall value of architectural interface.

In Chapter 6, the overall notion of architectural interface is evaluated based on the experience gained from the studies conducted in the previous chapter. Also, the chapter presents the assessment of ArchInt prototype with a discussion of its observed limitations.

The conclusion of this research is given in Chapter 7, where the achievements, contributions and problems are summarized. Additionally, the main research aim and objectives are revised and the usefulness of architectural interface as a paradigm to support software re-use is discussed and areas for future work are identified.

Chapter 2 - Background Research

The previous chapter set the scene of the research by establishing the importance of re-use in general to enhance software development, and identified some of the obstacles encountered by re-users that undermine the predicted advantages of re-use.

This chapter describes the background work to set the context for this research. The chapter starts in Section 2.1 with a general discussion about the definition and history of re-use. After that, Section 2.2 identifies various artefacts that could be re-used to build a system. Section 2.3 describes the software components and identifies their characteristics. Then, Section 2.4 discusses the notion of component-based software development (CBSD) to identify the area that is the main focus of this thesis. Section 2.5 discusses the notion of software architecture and describes the significant impact of such architecture in tackling the problems encountered in the field of software re-use. After that, Section 2.6 describes the notion of open-source software and identifies the current state of re-use in that area. Finally, Section 2.7 sets the context for the research under consideration by identifying the key points from the background work that has been discussed in this chapter and mapping some of the terminology used in the literature to that which is used in this research.

2.1 The History of Re-use

Re-use is one of the old paradigms that were commonly practiced in many different professions. In car assembly lines for example, motors, body parts and many other components are re-used from one model to another. Rarely are new parts built from scratch. Electronic engineers assemble their integrated circuits from resistors, transistors, diodes and many other re-usable components. They simply search for the required component on the corresponding data-sheets that explain the detailed specification of each type of component so that they can re-use them.

In software, the concept of software re-use has existed since the beginning of programming in that programmers were re-using algorithms, sub-routines and pieces of code from previously created programs. The idea of re-use was firstly formalized by McIlory [103] who emphasized the need to componentize software systems. So, applying McIlory's idea has led to us thinking about building software systems in a similar manner to building hardware systems (e.g. electronic circuits). Later on, more advanced research work emerged that discussed re-use and its possible directions, emphasising the significance of re-use. Nowadays, re-use has become one of the standard paradigms that

most of the leading software development corporations such as HP [62], IBM [154] and Motorola [79] have practiced in their software development processes while many others have reported successful experience with practicing re-use in their software development projects such as the examples provided in the C.R.U.I.S.E book [6].

The term, re-use, in its most basic meaning, indicates obtaining some of the already built parts with the intention of using them in the building of a new product. A diversity of descriptions are available in the literature about software re-use. Some of the commonly known descriptions are:

Software re-use is a process by which organizations describe a set of systematic operation to generate, organize, and locate re-usable components for future development [111]. Sametinger [127] described software re-use as the process of re-using some already built components to construct a new system. Software re-use has also been described by Krueger [85] as the process of using existing components to build systems instead of building them from scratch.

Software re-use is among the significant software attributes that permits software artefacts to be taken from one project and incorporated into another that shares similar characteristics [109]. Many types of artefacts can be re-used. The next section describes the different types of artefacts that can result from a software development process.

2.2 Types of Re-usable Artefacts

Artefacts might be described as pieces of formalized knowledge that can contribute to the software development process [33]. Artefacts might be a complete solution to a problem (e.g. a whole system) or they might be part of a complete solution. While re-using both might of interest to re-users, it is observed that re-using part of a system always raises problems as will be see in the coming sections. Therefore, this thesis is concerned with investigating the problems encountered when re-using parts of a system.

Many artefacts are produced during the different developmental stages of a software system. The following describes the different stages normally found in most development processes (e.g. Waterfall [136]) and identifies the possible artefacts produced within these stages:

Requirements & specification stage: the main objective of the requirement and specification stage is to identify the problem to be solved and also to describe the possible solution to that problem. Many artefacts might be produced during this stage including

analysis, data dictionaries, validation schedules, diagrams and a requirement and specification document.

Design stage: the design stage is the stage in which the solution to the problem described by the requirement and specification stage is analyzed. The design of a system represents the blueprint of the overall system's organization that describes the architecture [14] of the system, its composing elements, and the way that these elements interact with each other. Several artefacts might be produced during this stage including design patterns, documents and diagrams.

This research refers to such artefacts as high-level artefacts as it is believed that they are usually generated at a high level of abstraction. The artefacts generated during the forthcoming stages are referred to as low-level artefacts.

Implementation stage: the main objective of the implementation stage is to write source code that reflects the design of the system being built, according to specific syntax, so that tools (e.g. the compiler and the linker) can parse the source code and interpret it into executable form (i.e. into binary code) that provides the real working functionality of the system. Several artefacts might be produced during this stage including components, either in the form of source code or binary code, complete systems and documentation.

Test stage: the purpose of the test stage is to ensure that the code artefacts have passed through a number of tests in order to confirm their suitability and correctness in terms of working as expected. Several artefacts might result from this stage including a test suite, unit test document, test plans, test practices and some code review techniques.

Theoretically, many artefacts produced in a software development process could be re-used. Specification documents, for example, could be re-used in a development process that intends to build a similar system with some added functionality. Design patterns [54], for instance, are a type of artefact that can be re-used in software development within the design stage. Design patterns describe solutions to recurrent problems at the design level. So, one might re-use design patterns to help overcome system design problems. The tests generated in the validation of the code of one project can be re-used in similar projects [36]. Re-using tests can increase the maintainability of software systems [3].

Although re-using high-level artefacts might be beneficial, several problems might be encountered by potential re-users. High-level components produced as part of the software development process normally suffer from redundancy, errors, disagreements and ambiguity [108]. This is mainly because they result from meetings, email discussions, or

interviews that are usually recorded in a natural language which is the primary form of communication between the stakeholders or used in legacy documents [149]. An additional problem that re-users of high-level artefacts may encounter is that one cannot guarantee the availability or accuracy of such artefacts at the end of the development process, due to their being lost or not appropriate for the system that is eventually produced. The only artefacts that can be reliably found at the end of a development process are the source-code components.

Re-using source-code components is a particularly important and growing area of interest, especially with the pioneering developments of open-source software [20] that gives access to huge collections of freely available source-code components. One advantage of re-using source-code components is the possibility of re-users examining the source code for “Trojan horses” which form a threat to their systems. Another advantage is that the source code is a precise description of behaviour, unlike the natural language that might be used to describe behaviour in the high-level artefacts. Moreover, source-code components can be utilized by a repository system to be analyzed, checked, and organized automatically. Therefore, this research is concerned with the re-using of source-code components.

Practicing re-use involves examining which re-use approach suits the need of the development process in terms of the ease of obtaining and customizing particular components [121]. Two main approaches are commonly used in the field of re-use, namely, white-box and black-box re-use [98]. White-box re-use involves re-using a component that have its source code available, hence the component can be modified to suite the needs of the developer. Black-box re-use involves re-using components as-is, without modification as the source code is not available. In practice, components that match the exact specifications of re-users are rarely available. Re-users need to modify or customize a component to fit it into their system. In the case of white-box re-use, modification can be done by modifying the source-code, while in black-box re-use the modification is done by customizing components to match the desired specifications. Two types of modification have been identified, contextual modification and domain modification [121]. Contextual modification concerns modifying components to match the environmental and programming language requirements of the system to be built. Domain modification concerns modifying the functionality of a found component to satisfy the required functionality to be incorporated into a system. One potential advantage of white-box over black-box re-use components is that if a component that matches the functional

requirements is found then, in terms of white-box re-use, the source code can be modified to match the new contextual requirements. This, however, is not possible in the case of black-box components. Hence, re-using source-code components (white-box re-use) can be advantageous over binary-code components (black-box re-use).

Despite the many advantages of re-use in general terms, it is not widely practiced due to the difficulty of finding re-usable components [13]. The problem of finding them includes locating a proper source of re-usable components and also the ability to find suitable components that fit among the available ones. Another problem is related to quality assurance issues in the sense that component quality cannot be certified [145]. So, many aspects of software components need to be studied in depth in order to understand the characteristics of software components and consequently to establish a solution that tackles the problems encountered. It is appropriate now to present the different meanings of the term "software component" in the literature before going further in discussing other aspects of background work. Hence, the next section presents a number of definitions of "software component".

2.3 Software Components

A software component is defined variously in the literature, in that there is no single accepted definition of the term yet available [19]. The following descriptions are the most prominent ones within the software industry. Brown and Wallnau [22] described components as a nearly independent and replaceable part of a system that satisfies some functionality in the context of a well-defined architecture. The component can be bound dynamically and accessed through a well-defined interface at run-time. Szyperski *et al.* [139] described a software component as a unit of composition with a specified interface and explicit context dependencies. The component can be deployed independently and subject to composition by a third party. Meyer [107] described a software component as a software element that can be used by other software elements (e.g. clients), possesses an official usage description, and is not tied to any fixed set of clients. Heineman and Councill [65] described a component as a software element that conforms to a component model [87] and can be deployed independently and composed according to composition standards without modifications. Yang and Ward [152] described a component as a coherent and configurable package that is available independent of the application in which it has been used with a well-defined interface that can be used in different contexts to interact and communicate with other components to form a system. Brown and Short [21] characterized

a component as "*…an independently deliverable set of reusable services*". Hopkins [70] described a component as a physical package of executable code that exhibits a well-defined interface.

The different interpretations of a component seem to agree with regard to the description of the main characteristics of a component. All the descriptions agree that a component is a packaged part of a system that conforms to specific characteristics and provides some functionality. Moreover, the descriptions emphasise the necessity to have an interface that can achieve interaction between components and also define explicitly the dependencies of a component. The author's understanding of a component, in a very abstracted view, is that components are just parts that fit into a system. They must exhibit characteristics through their interface to facilitate incorporation into a system and also for identifying them for re-use.

2.3.1 Component Interface

A significant part of a component is its interface. An interface describes the specifications of a component [139]. It separates the abstract specification of a component from the underlying implementation that specifies how a component can provide certain behaviour [21].

Design by contract (DbC) by Meyer [110] is mainly concerned with defining the formal specifications of component's interface in order ensure that the collaboration between the components of a system is correct. The notion of DbC guides the design of the software system by specifying a set of pre-conditions and post-conditions as part of the interface of a component. Pre-conditions are the requirements that must be made available to a component prior to be able to provide its services (e.g. "You need a debit card to withdraw from a cash machine"). Post-conditions define what a component will provide once a condition is satisfied (e.g. "withdrawing money"). Brown and Short [21] described an interface as a way of summarizing the behaviour and the responsibility of the component. They used an interface to capture all the semantics related to the collaboration between components. The set of operations provided by a component is considered as part of the exhibited interface that a client or a system can use to obtain the required functionality of that component. Sametinger [127] described a component's interface as a way to determine how a component can be re-used and composed with other components in a system. An interface defines the set of operations that characterizes the behaviour of a component. Sametinger distinguished between three types of interface namely, data interface, user

interface, and programming interface. Data interface concerns the format and transformation of the data between components. User interface captures the protocol of interaction between a component and a user, for example through a simple command line or a graphical user interface. Programming interface captures the possible interactions between components and how they can be composed in a system. Arbab *et al.* [9] described the interface as a definition of the observable behaviour of components that contains five elements. These are a name, a channel signature, a blocking invariant, pre-condition, and post-condition. The name of an interface is used to uniquely identify an interface from other interfaces. The channel signature captures a set of parameters representing the data input and output of a component. The blocking invariant specifies special cases when a component needs to allow exceptions or perform a special action. The pre-condition refers to the required set of inputs that must be supplied to the component in order for it to operate. The post-condition refers to the set of values that are supplied by the component. They considered the component interface as a way to reason about the correctness of composition of a system from its components. Hondt *et al.* [69] described the notion of a *re-use contract* that concerns capturing the requirements of a component from other components in a system. They considered the interface as a way that not only captures the operations responsible for providing functionality, but also document what a component requires in order to work and what interaction structure is required in order to obtain a correct collaboration between the components of a system. An interface of a component captures the signature of operations without considering any semantics or type of information. The key contribution of the notion of a *re-use contract* is to detect conflicts in component interfaces, in that a conflict indicates that components cannot work together in a system.

The author of this thesis describes the component interface abstractly as a contract of fit. An interface is, in fact, a kind of contract of communication between a component and a system. Both a component and a system must agree upon a defined contract in order to allow for a component to be re-used in a system and also to allow a system to use the component. The characteristics defined by an interface capture the functional and non-functional aspects of software components. Based on the exhibited characteristics of a component's interface, a component can be identified and re-used. The component's interfaces can be represented directly in the code of the component (e.g. Java Interfaces) or by using additional files (e.g. a textual file) that describes the interface of the component.

2.3.2 Component Categorization

In the literature, several works have provided a categorization of software components. An early categorization was provided by Booch [18] that suggested that source codes have categorized components into structure, tool, and subsystem. Structure denotes components that are of an abstract data type (e.g. class). Tools are components that are denoted by an algorithmic abstraction. The subsystems category refers to a logical structure of cooperative components. Booch's categorization seems concerned with identifying the granularity of the software component. So, structure represents individual classes or objects, hence can be considered as a fine-grained view of components. Tools represent a collection of classes or objects that interact to perform certain functionalities, hence can be considered as a medium-grained view of components. A subsystem represents a collection of tools packaged together as a library, hence they can be considered as a coarse-grained view of components. An interpretation different to Booch's taxonomy could be that components might be of a single class, an application, or a library. Kain [80] distinguished between two types of components namely, specification and implementation. The major motivation behind proposing this categorization seems to be that Kain wanted to separate the interfaces of a component from its real implementation. So, specification captures the description of the characteristics (e.g. Java interface) while implementation captures the description of the technical details behind a component and the way that component behaviour is implemented (e.g. Java class). Dusink and Katwijk distinguished between two types of components based on thread of control as active and passive components. Active components are those that originate a thread of control in a system. They could be the core components that instantiate other components in a system (e.g. the framework). Passive components are those that receive a thread of control to accomplish certain task such as libraries or databases. Heineman and Councill [65] categorized components into GUI components, service components, and domain components. Their categorization seems to distinguish between components based on their cost and complexity. Heineman and Councill suggested that re-using GUI components (e.g. buttons, forms) is the simplest ones among the other types of components and their re-use might increase productivity by 40%. Service components (e.g. database access) are more complex than GUI components, but their re-use can increase productivity by 150%. Domain components (e.g. payroll, bill calculation) are the most complex components among the other two. Their re-use can increase productivity by 1000%. However, their development requires extensive work.

The author of this thesis has categorized components based on their interfaces into single components, component structures (i.e. system) and applications. Detailed discussion of the different interfaces is given in Chapter 4. In the meantime, two interfaces are distinguished namely, external and internal interfaces. The single component category specifies two interfaces; an external interface that a component must exhibit to fit into a system, and an internal interface that specifies the dependencies of the components (i.e. sub-components). The system category describes two interfaces - an external interface which describes how a system can be used by a user, and an internal interface that describes the specifications that components must conform to. The application category describes only the external interface that describes how the application can be used by a user.

Components are the cornerstone of the field of component-based software development (CBSD) that addresses aspects relevant to putting the components together in order to form a system. The next section describes the notion of CBSD.

2.4 Component-Based Software Development (CBSD)

The notion of CBSD is not new. It was firstly coined by McIlory [103] who established the need to componentize software (i.e. building software from components) as a way of resolving some issues identified by the software crisis that concerns the case of building large and reliable software in a controlled way [148]. CBSD is concerned with the assembly of software systems from pre-existing software components. One of the main objectives of the CBSD approach is to promote the re-use of previously developed components to allow the building of a new system. The notion of building a system from components can reduce development costs and increase the quality of the final system [139].

Building a software system from re-usable components requires a clear understanding of the aspects related to the characteristics of the overall system, the characteristics of software components, and aspects related to obtaining and integrating components [101]. A common model for CBSD is that a re-user who wants to add functionality to their system might find a component repository to search for re-usable components. The re-user then gathers their ideas about the characteristics of the component they are looking for. After that, the re-user types a search query that formulates their thoughts about the characteristics of the required component, either as free-text or in the form of a specification model [21]. Alternatively, the re-user could browse the available categories in

case they are not fully aware of the representation method [49] used by the repository to organize the component. In this way, browsing can build up their knowledge [93]. In response, the repository may list a number of results that are relevant to what the re-user needs. Consequently, the re-user can examine the characteristics of every component on the list until they find a best match in terms of the required characteristics. Sometimes, the re-user might need to modify the component they have found in order to exactly match the requirements of the system to be built, so they might apply some adaptation techniques [16] to accomplish the modification. Once the component matches the required characteristics, it can be incorporated safely into the system.

The above model identifies a number of aspects with respect to development according to the CBSD approach. The various aspects are discussed in the next sub-sections.

2.4.1 Identification

Identifying components involves recognizing the potential of re-usable ones, based on their exhibited characteristics from a list of components. This activity involves searching and browsing software components. The selection of the appropriate component from a list of components is done by matching the characteristics of the component to the specifications of the system to be built. This requires a precise definition of the components' characteristics in order to facilitate an understanding of them by their users and also to classify them for re-use [94]. The success and soundness of the identification of the component is a major factor for the success of the CBSD approach as components cannot be re-used unless they are found [7, 135]. The key element for the success of the identification activity is the availability of an effective organizing scheme with regard to the software component [101]. A detailed discussion about the available characterization organization schemes (i.e. classification schemes and indexing schemes) is given in Chapter 3.

The software components can be identified in various ways. Some of the common ways of identifying components are based on matching their behaviour [117], their signature [156] and their specifications [66, 77]. Behavioural matching identifies components based on a set of predicates (i.e. pre- post-) that are used to execute components. The resulting values of the execution are then used as representative "*terms*" to identify the corresponding components. Signature matching identifies components based on the signature of the functions within a component and the type of parameters. For example, in ML a function "hd" can be identified by the type of its input and output parameters "a

`list → a`". A whole component that is composed of several functions can be identified by the signature of the functions within the component. Specification matching is derived from the behavioural matching approach. However, it relies on predicates of the entire component's operation. The set of predicates are written using formal specification languages such as Z language [137] or OCL [46].

2.4.2 Validation

Validation is a way of checking the characteristics of the component against a pre-defined specification. Two kinds of validation are relevant to the CBSD paradigm - unit test and integration test [123]. The unit test is done by a component developer to ensure that the provided behaviour of the component is correct [35]. Testing a component's behaviour could either be done as black-box testing by providing a set of inputs and examining the resulting output, or white-box testing by inspecting the source code [124]. The integration test is undertaken by a component re-user to determine whether or not the component can interact with the other components in the system and is not going to raise any structural problems [72]. In addition, integration tests can be done in some cases to measure the quality of a component in order to decide whether or not the component can be trusted for re-use [107].

2.4.3 Integration

The activity of integrating a component can be seen as a mechanical activity involving connecting components by means of matching their syntax and semantics to form a system [27]. Part of the integration activity is related to checking the compatibility of the components to match the characteristics of a system [38]. The main issue to address in this activity is related to solving potential mismatches [52] between components. One reason behind the occurrence of a mismatch in a component's characteristics is due to the fact that the component's producers may be unaware of the potential usages (i.e. context) that their component might be re-used in, hence their assumptions are different from the assumptions considered by the components initial users [29]. Thus, it is extremely important that the software components are produced with a well-defined interface in order to understand the assumptions that components can match [65], and also can be connected with at runtime [114].

In a system that is built locally, integrating heterogeneous component can be achieved using a wrapper or glue code to bridge the differences between the components' interfaces.

So, if a component that takes two parameters as an input is composed with a component that provides three values as an output, then a glue code can be used to map the input and the output of the components. In building a distributed system, the interaction between the components can be addressed using the notion of middleware (e.g. CORBA [134]) that unifies the components' interfaces to enable their interaction across a network.

2.4.4 Evolution

This activity is concerned with replacing components from a system with other components that conform to the same interface, so that we can substitute the replaced component without affecting the other components of the system [10, 21]. The reasons for replacing the components could be to fix bugs in the system or to extend the functionality of the system by incorporating new components that provide the desired behaviour into it. Consequently, this activity is important in the notion of CBSD.

2.4.5 Discussion

With regard to the above aspects, integrating components is a significant issue that needs to be investigated in depth [142]. Addressing component integration is especially important when dealing with heterogeneous components, as they might cause lots of interoperability problems when re-users need to incorporate components into their systems. Components can either be integrated statically or dynamically [25, 109]. Statically integrated components are those that are bound by programming mechanisms (e.g. method invocation) at compile time and usually conform to an architectural style [56] (described in Section 2.6). The dynamically integrated components are those that are bound at run-time and they are identified by the services (i.e. behaviour) they can provide.

Integrating components involves adapting components to resolve potential mismatches in the characteristics of a component and the characteristics of the system to be built. Adaptation refers to modifying the interface of the software components by means of using a wrapper, glue code, or a translator to eliminate the unnecessary behaviour of a software component and also to add additional characteristics to its current interface in order to meet the requirements of a re-user [21]. Specifically, the adaptation of the component is mainly concerned with solving potential interaction problems that are caused due to potential architectural mismatches between components interfaces [57].

Several attempts have been made to try to tackle the problem of integrating components in a system. Eclipse [31] has established a framework by means of plug-ins that

encapsulate components in order to unify their interfaces. So, different components that conform to different assumptions can be incorporated into an Eclipse if they are wrapped with the necessary plug-ins' architectural characteristics. The Vienna Component Framework (VCF) [115] has established a framework similar to that of Eclipse, but its authors claim that it is has an advantage over Eclipse in the sense that it provides uniform access through different component models. VCF has defined general characteristics for software components that, they claim, are common among different types of components. These characteristics are:

- Life-cycle: every component must implement a set of methods that allows a system to control it when it must be initialized and destroyed.
- Persistence: this allows a component to be stored and retrieved from storage.
- Method: this characteristic gives a handle to the real methods provided by a component that are responsible for functionality aspects.
- Property: this characteristic allows for the manipulation of the component's state.
- Event: this characteristic allows components to be registered as listeners to be notified about events.

Integration problems are experienced in various situations where the CBSD approach is used to build a system. One of the prominent examples that demonstrates this problem is the integration of Commercial Off the Shelf products (COTS) [26]. The problems encountered at integration time are primarily due to potential mismatches in the architectural assumptions between the components that are planned to be re-used and for the system to be built [133][56]. One may find, somehow, a component that seems to satisfy their functionality. However, that component does not fit into the system under development due to an incompatibility in the programming language used to write the component and that of the system, such as incompatibility in the operating system or the database schemes [56]. These incompatibilities are additional difficulties that a re-user might need to take into consideration when considering re-use. As a result, software components must define their characteristics through their interfaces in order to make the characteristics explicit for re-users wishing to identify a component from others [30].

The next section introduces the background work in the field of software architecture to give the reader information about how software architecture might affect components' re-use and the role of software architecture in tackling the problem of integrating components.

2.5 Software Architecture

People usually refer to the term ‘architecture’ to indicate the physical construction of a building in terms of external shape, and also how rooms are structured within that building. In software, the word ‘architecture’ is a term that is in general use, with a number of different interpretations. However, as an analogy to its meaning in civil engineering, it inspires the meaning of creating a product (software system in this case) from a number of selected components rather than building a single monolithic one. So, the way components must be incorporated, the order in which they must be placed, and the mechanism of interaction between them, are parts of what system architecture describes.

Bas *et al.* [14] defined software architecture as the structure of a system that comprises software elements, their external visible characteristics, and that defines the relationship between them. IEEE 1471 [74] defines software architecture as “*the fundamental organization of a system embodied in its components, their relationship to each others and the environment, and the principles guiding its design and evolution*”. Jones [78] defined architecture as the structure that is composed of components and rules that establish the basis for the interaction between them. All the definitions have agreed upon the fact that architecture is concerned with the constituting parts of a system and the relationship between them.

In the literature, many of the available sources have explained the significance of considering architecture in software development (especially in the CBSD paradigm). One reason for considering software architecture is to help our understanding of complex software systems [60]. Shaw and Garlan [133] suggested that architecture can be used to define the overall design of a system. Garlan and Perry [59] identified the benefits of considering software architecture in software development as providing support for re-using, evolving, analysing, and managing software. Budged [24] considered software architecture to be a way of describing the constructional aspects of a software system at a high-level of abstraction (e.g. design stage). Allen [5] identified architecture as being the vehicle to communicate between the requirement and the implementation stages. Szyperski [139] suggested that architecture is important for establishing a context for software systems representing standards and platform requirements.

Garlan *et al.* [56] identified a number of architectural characteristics that might cause a mismatch to occur in terms of component interaction within a system. These characteristics are:

- The infrastructure that a component is primarily built on.
- Control issues of whether a component can generate a control signal or not.
- The data type manipulated by a system and the way it is transferred between components.
- The pattern of interaction between components.
- The sequence that components must be instantiated and invoked with.

From the re-users point of view, these characteristics are significant in order to identify whether or not a component can be re-used within their system and is based on an understanding of the different characteristics of the architecture to hand. Consequently, a component that supports a single thread of control will not be suitable for re-use in a system that assumes its components must be thread-safe. Also, a component that communicates through RPC will not be re-usable in a system that uses message passing to transfer data [37], hence a mismatch might occur.

Yakimovitch *et al.* [151] refined the work of Garlan [56] and identified five variables that describe assumptions about components' interactions, namely packaging, control, information flow, synchronization, and binding. Their main motivation was to establish a mapping between certain architectural assumptions and some real problems. They demonstrated that the defined variables can be used to classify different software architectures.

Software architecture seems to consider another view of a system that is not tightly relevant to functionality. This view examines the structure of a system and tries to identify components and define the possible interaction that a component can have in order to avoid the occurrence of fault [90] due to a potential mismatch between components in a system. The development of the AESOP system [56] from large-scale components demonstrated the difficulty of incorporating components, and emphasised that the main reason for the observed difficulty is due to architectural mismatch between the various components. Even though the various components of the AESOP system were providing the required functionality as the developers needed, the integration of the various components to form a complete system was impossible without major modifications. The problems encountered by the AESOP developers was in favour of the assertion by Shaw [130] that considering functionality alone is not enough to successfully re-use components.

The core elements of software architecture are *architectural styles* and *architectural patterns* [25]. These two elements of software architecture are discussed in next sub-sections.

2.5.1 Architectural Styles

The basic element of software architecture is represented by the notion of architectural styles that define a family of structures for software components to guide the design of a system. Garlan and Shaw [60] defined the vocabulary for architectural styles in order to understand the construction of a system. They defined the notion of components (e.g. COTS [26]) to capture functionality, connectors (e.g. shared memory, RPC, Network protocol [133]) to capture interactions among components, and constraints to define how components and connectors can be combined. Later on, Shaw and Clements [131] defined architectural style as set of design rules that identify components and connectors that make up a system, together with constraints that govern its composition. They characterized architectural styles based on:

- The kind of constituting parts (components and connectors)
- How control is transferred among components
- Issues of how data is passed through the system
- How control data interacts
- What type of reasoning is compatible with a selected style

Their characterization is based on the coarse-grained description of properties and motivated by the need to discriminate between the organizational structure of software systems (i.e. styles) in order to help the software designer to identify a suitable style for the system that they intend to build. Moreover, their characterization is aimed at establishing uniform descriptive standards for architectural styles so they can be publicised among software architects.

Mehta *et al.* [105] established a taxonomy for software connectors in order to understand the building blocks of the main constitution of component interaction (i.e. connectors) as this is a significant aspect to consider for accomplishing the successful integration of software components. They identified the major connector types as: procedure calls, data access, linkages, streams, events, arbitrators, adaptors, and distributors. The main motivation for their work was to define precisely the high-level

terms used to refer to connectors in a more meaningful way that can be physically observed. They have identified that one problem with the current approach with regard to software architecture is in representing the interaction of software components. Interactions are always represented at a high-level of abstraction at the design stage. However, interaction is hidden at the source-code level and one may not able to identify what part of a source code is concerned with the interaction with the other parts that are responsible for providing functionality.

DeLine [37] identified two parts of a component; the functionality that can be provided by a component (named as ware) and the packaging that indicates the way that it can interact with other software components (named as packager). The two parts, when combined, form a complete software component. DeLine has established a method known as *Flexible Packaging*, that takes the decisions about the components' interaction out of the provider's hand and places them into the hands of the component's re-users. An assumption is made in their work that in using *Flexible Packaging*, the components' seller needs only to concern themselves about the functionality of the software component; the packaging will be selected by the re-users when they acquire components. The main motivation for their separation is to reduce the difficulty of writing the packaging source code by a component developer. As a result, a developer will focus only on the functionality aspects of the component they are building, while the packaging is addressed based upon the request of the integrator.

When a component is integrated into a system it must match the characteristics defined by the architectural style of the system to be built [151]. The characteristics include the type of the component and the way that the system expects the components to interact. If a component conforms to different architectural styles than the one required by a system, then developers need to use techniques such as wrapping the component or writing a glue code in order to exhibit the characteristics that allow for its integration within a system. The availability of source-code is the prime factor in deciding upon the technique used to integrate components [130].

2.5.2 Architectural Patterns

The notion of an architectural pattern is derived from the notion of design patterns [54] that are concerned with defining the organizational structure of a system. The architectural pattern follows a more precise path to define the structure of a system. It captures the recurrent form of the interaction between components and also the types of the components

selected in every pattern. Buchman *et al.* [129] defined architectural patterns as a way to express the fundamental structural organization of a system by identifying the set of components, specifying their responsibility, and laying down the rules that govern their relationship with one another. The architectural pattern helps software builders to understand how they need to organize their systems. A prominent example of an architectural pattern is the Model-View-Controller (MVC) architectural pattern that separates the presentation of data and its computational aspects. One significant motivation behind developing the notion of an architectural pattern is to support the inter-changeability of the components of a system. The architectural pattern defines the components' characteristics that distinguish it from other components in a system. The definition of the components' characteristics is advantageous for supporting the mix-and-matching of the components of a system in the sense that one component can be identified and extracted from a system and replaced with another that fits into its place without affecting the other components in the system.

However, architectural patterns suffer from the ambiguity of what they really represent [11]. In addition, there is no single way to define an architectural pattern as every designer can define a pattern based on their perspective with regard to the construction of a software system. This results in having a different view or examples of a single pattern. One potential reason that can cause the ambiguity in defining architectural patterns is caused by the fact that they are represented at a high-level of abstraction without considering any concrete representations of such patterns at the implementation level. Thus, various implementations might be available for the same architectural pattern.

2.5.3 Architecture Description Languages (ADLs)

ADLs are specification languages for defining the structure of software systems at a high level of abstraction by identifying elements and the relationship between them [32, 55]. ADLs provide a description of the conceptual architecture of a system [104]. A general characterization of ADLs' capability was given in [61]. ADLs aim to support architecture-based software development by establishing notations that are appropriate for defining system architecture and its constituting elements. They formalize the definition of a system at the architectural level in a graphical way that can be communicated to humans. Moreover, instead of drawing boxes and lines that may not involve rules that govern connections between them (i.e. boxes and lines), ADLs provide a semantic check of

whether two elements can be linked together and what the requirements are that need to be satisfied in order to successfully create the links between these elements.

ADLs are built on the notion of components, connectors and constraints that have been described in the software architecture field. They provide a basis for analyzing and verifying the design of a software system [61]. There are many ADLs available nowadays such as ACME [58], that can be used to represent the architecture of the system to be built (Darwin [34], Rapide [97] and many others). ADLs possess several characteristics that are relevant to the CBSD field as many of them facilitate the automatic generation of glue codes to form a system [76]. Despite the variety of ADLs, they are not widely adopted by the software industry, because they are not general enough as they only support specific architectural styles [17].

2.6 The Development of Open-source Software

Open-source software [147] represents one of the most interesting and influential trends in the software industry over the past decade. The notion of open-source software development was firstly coined by the GNU project at MIT in the early 1980s. Their main intention was to encourage freedom in producing software systems [88], and also to compete with commercial software products [122]. In fact, commercial software organizations could not function without using open-source software as part of their products [20]. Today, many organizations are looking towards open-source software as a way of providing greater savings in their development costs, and many software systems such as Linux, Apache, Mozilla and Openoffice have been developed in this way. The term ‘open-source’ refers to software in which the source code is freely available for others to view, use, execute, amend and adapt [88]. The open-source definition proposed by the *Open Source Initiative* (OSI) can be used to determine whether or not software can be considered as being open source. With regard to the definition of open-source, a software must satisfy a number of criteria including [44]:

- It can be freely re-distributed.
- The source code must either be included with the component or freely obtainable.
- Re-distribution of modifications to the source code must not be restricted to usages or applications [51].

Many open-source software hosting environments (i.e. repositories) exists on the Internet. These include Sourceforge, Freshmeat, Apache, CPAN, RpmForge and many

others. These environments are places were software project developments are launched and contributors from all around the word participate in building software products within these environments. Contributors can communicate with each other by e-mail to exchange ideas, decisions and design documentation [20]. However, these open-source environments are not limited to the contributors who participate in building systems, but are freely open to anyone who might be interested in observing, learning, and re-using some of the available source codes. In this sense, these environments can be considered as repositories of source code products. So, open-source repository systems represent a rich resource that can be utilized by re-users to obtain re-usable source code due to the huge amount of source code available at no cost within these repositories.

Based on the author's practical experience of re-using open-source software, current open-source repository systems facilitate finding software components using free-text searching mechanisms based on matching words or phrases to the descriptions of software components or their corresponding source code. Many search engines such as Koders, GoogleCode, Krugle and many others can be used to facilitate searching through the source codes of open-source software in order to help find re-usable possibilities. Alternatively, a re-user could browse through a number of categories that organize software components until they find a potential candidate for re-use. These are the two methods used to acquire components in almost all the open-source repository systems. Further characteristics of open-source repository systems are given in Chapter 3 when discussing the characteristics of the Sourceforge.net repository system.

In the literature, the major concern with regard to discussing the re-use of open-source software, focuses on the security and licensing aspects [20]. Addressing aspects such as the effective re-use of open-source software to improve software development is not considered in depth. As a result, research in this area is still needed when it comes to utilizing open-source products for software development.

2.7 Setting the Context of the Thesis

The main theme of this research is a desire to utilize the architectural characteristics of source-code components in order to support the finding of re-usable candidates that can fit into the re-users' system that is under development. Based on the work reviewed in this background chapter, the following key aspects are the main forces and motivations that drive this research:

1. The free availability of rich source-code components that resulted from the open-source software movement.
2. The support provided by open source software repositories is limited to string matching which is not very effective when it comes to obtaining re-usable components.
3. The major problems that hinder re-use are related to identifying and integrating components into a system due to the potential mismatches between their characteristics and the characteristics expected by a system.
4. The architectural characteristics are defined abstractly in the design stage and are not reflected precisely in the source code.

So, this section draws on the different work discussed earlier as a context for this thesis.

A problem encountered with component re-use is related to identifying the standards that components have to be conformed to, and also the dependency issue that is related to specific models [115]. Moreover, source-code components obtained from open-source repositories usually suffer from the absence of documentation that a re-user can examine in order to identify whether or not a component can fit into the system under consideration [121]. Furthermore, the developers of open-source software components are not normally available. The only thing that can be examined is pure source code. The current support provided by the open-source repository system is restricted to sting matching that may result in listing a large number of irrelevant results. As a result, an effective characterisation of source-code components is needed in order to refine the re-user's search and to generate a list of more focused results.

The approach adopted in this research is to characterize software components based on their architectural characteristics. The motivation for this selection is based on the author's experience in finding components that have the required functionality but never work in the system to be built. This decision to consider architectural characteristics is motivated by Shaw's argument that stated functionality alone is not enough and that packaging should also be considered [130].

The value of considering architectural characteristics appears at the time of integrating components to the system under development. Integrating components, as described earlier, involves an understanding the architectural characteristics of the software components, in order to identify whether or not a component can be re-used in a system

and is not going to cause a mismatch. However, architectural characteristics are always defined at the early stages of a development process (e.g. the design stage). Not many details of the architectural characteristics are reflected in the actual implementation of a software system. The work by Shaw and Clements [131] identified the notion of components and connectors to classify different architectural styles. However, these terminologies (i.e. components and connectors) are very abstract and can only be used to define a system at the design stage. The work by Mehta *et al.* [105] identified a fine-grained view of component interaction (i.e. connectors) in an attempt to reflect the high-level principle of connectors with a physical meaning that can be observed during implementation. However, at the source-code level, one may not be able to tell whether a method in the source code is responsible for interaction (i.e. for the connector) or whether it provides functionality [37]. For example, if an interaction is defined between two components as a "method invocation", then knowing this will not be of significance to a re-user who wants a precise specification in terms of how the interaction has been accomplished. Thorough characterization of components' interaction is required in order to identify them in the source code [99].

The problem that high-level architectural decisions are not reflected precisely on the source code was addressed partly by DeLine [37] who established a distinction between the functional concern and the architectural concern of software components. However, DeLine's approach focused more on addressing the problems of interaction between the source code responsible for providing functionality and that of the architecture. Moreover, DeLine assumed that the component should be made available to a repository as a source code that purely defines functionality. The source code responsible for capturing the architectural characteristics are left unspecified until a re-user describes the required architectural characteristics. Based on the provided characteristics, the component is then wrapped as necessary to match the characteristics of the architectural style of the re-user. Once a suitable wrapper has been generated, then both the functional and the architectural source codes are combined to form a complete component. Although the work of DeLine is closely related to the work presented in this thesis, it may not be applicable in the case of open-source software components where all the source codes that are relevant to functionality and architecture are mixed together. A provider might provide a component that is composed of functional and architectural source code, hence violating the assumption made by DeLine that a component should only be provided as a "ware". In addition, open-source software components usually lack any form of documentation that a

packaging specialist might use to identify the architectural source code from the functional one. Even though the specialist was able to use their expertise to identify the architectural characteristics and split them from the functional characteristics, it would be very difficult and time consuming.

So, based on the above identified problems, an approach is needed to identify the architectural characteristics of source-code components, based on nothing but the source code itself. Currently, open-source software components are characterized by textual descriptions that reflect the behaviour of the software components. Very little effort has been made with regard to characterizing components based on their architectural characteristics. Even the available architectural characterizations are restricted to programming languages. The major aspects discussed in the literature regarding the re-use of open-source software tends to focus on the security and licensing aspects [20]. Addressing aspects such as effective re-use of open-source software to improve software development has not been considered in depth. Therefore, this research is concerned with identifying the architectural characteristics at the source-code level in order to solve the problem of architectural mismatch, hence encouraging re-use. This research proposes a particular approach, namely *Architectural Interface,* that is dedicated to addressing the problem as will be seen in Chapter 4. However, it is appropriate to mention abstractly that the nature of the architectural characteristics considered in this thesis is derived from the previous work in the field of software architecture that concerns how to avoid architectural mismatch between components. This research aims to represent some of the key high-level architectural characteristics in a way that can be examined in the source code in order to avoid potential mismatch. Considering source-code components to capture some of the architectural characteristics is useful, as the source code is a precise representation of the system's design. Garlan *et al.* [56] suggested that a way to avoid architectural mismatch is by explicitly defining the architectural assumptions of the software components, hence identifying the architectural characteristics at the source code level is in line with their assertion.

With respect to the terminologies used in the field of software architecture, two terms (i.e. architectural styles and architectural patterns) are discussed and linked in this thesis. The author understands architectural style to be a representation of a systems view in the sense that it defines the architectural characteristics that a system requires, while architectural pattern represents components' view, as it defines the architectural characteristics of the components and their relationship. One architectural style can be

composed of a number of architectural patterns but not *vice versa*. As a result, architectural patterns can be considered as instances of an architectural style. Although the definitions of architectural styles and architectural patterns in the literature might seem similar, the author of this thesis observes that the architectural pattern identifies the characteristics that define the type of components at a high-level of abstraction. For example, architectural patterns define how a component can be a “Model” or a “Filter”. This characterization of component characteristics is not within the scope of architectural styles.

This research has adopted a similar term with regard to the two terminologies. The term *architectural interface* is used to represent architectural style and the term *architectural type* is used to refer to architectural pattern. These two terms are identified because the high-level views established in the field of software architecture are not commonly used at the source-code level, hence avoiding possible confusion in terms of terminologies. However, the identified terminologies for this research fall in the same context as those identified by the field of software architecture (i.e. to define the architectural characteristics).

2.8 Summary

This chapter has reviewed the different work in the literature that forms the basis for this research. The chapter has defined the term ‘re-use’ and has identified the various re-usable artefacts. The chapter then listed the different descriptions of software components and provides the description of components adopted in this research. After that, discussion about the notion of CBSD was undertaken and the different key activities that can be conducted in a CBSD were reviewed. Then, there was a discussion with regard to the field of software architecture to indicate the kind of characteristics that are of importance when considering the re-user. Finally, a number of key points were identified from the background work to describe a context for this thesis.

The next chapter discusses related work in the literature with respect to the characteristics of the ideal repository system (as described in Chapter 1) over three dimensions, namely organizing scheme, re-factorization and the overall characteristics of repository systems.

Chapter 3 - Related Work

The previous chapter discussed the background work conducted in this research to set the context for this thesis. It discussed the notion of CBSD and software architecture mapped the terminologies used in the literature to the ones presented in this thesis.

This chapter discusses the related work in the context of the ideal repository system in order to identify the characteristics that are of importance to a repository system to support component re-use. This chapter starts by an overview in Section 3.1 that describes the characteristics of the ideal repository system and its potential users in order to set the framework for the conducted surveys in this chapter. After that, section 3.2 surveys the available organizing schemes of software components in the literature and analyzes them in the scope of the characteristics, described in section 3.1, of the organizing scheme of the ideal repository system. Section 3.3 discusses the available work in the literature regarding components re-factorization. Section 3.4 describes the survey of a number of key repository systems and analyzes their capability to support re-use as compared to the characteristics of the ideal repository system. Finally, section 3.5 draws the chapter to a conclusion and establishes the necessary link to Chapter 4.

3.1 Overview

Pohthong and Budgen [120] identified two strategies for re-use that a re-user might practice; one strategy is that a re-user searches for re-usable artefacts first to derive their decisions for developing a system and then based on what is found the re-user start the building of the system. The second strategy is that a re-user might establish a framework (e.g. Java abstract classes) of the system to be built first and then starts searching for artefacts that fit with the requirements of the system to be built. This research adopted the second strategy that assumes a re-user is going to establish a framework first and then search for *artefacts* in the light of the specifications defined by the framework.

The vision story at the beginning of Chapter 1 described the support provided by the ideal repository system for re-use. As a way to express the various parts that are going to be introduced in this chapter in a precise manner, this section was intended to set the scene for the background work by illustrating the design of an ideal repository system. The envisaged design of the repository system that provides complete support for re-use is illustrated in Figure 3.1.

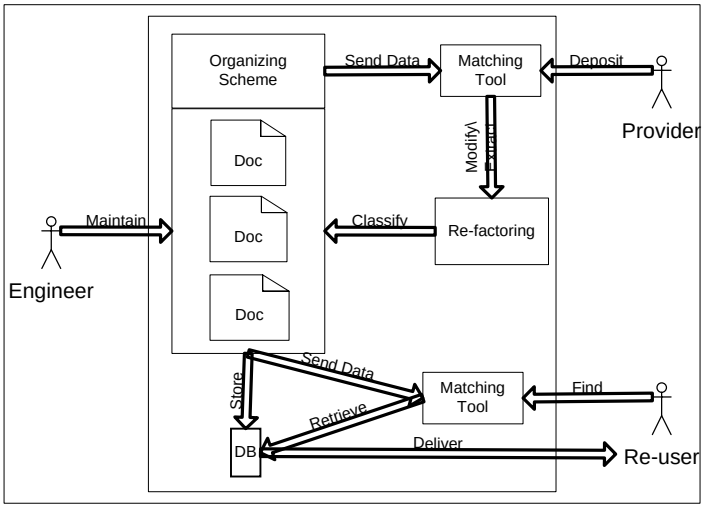


Figure 3.1: Design of Ideal Repository System

The illustrated repository system is composed of several components such as, organizing scheme, re-factoring tools, search engine, analyzing tool, matching tools, and database storage. The core elements of the repository system are the organizing scheme, matching tool, and the re-factoring tool, whereas the other elements are, somewhat, supporting tools. The main purpose of having a repository system is to establish an environment for supporting components re-use. The environment addresses human aspects and technical aspects. The human aspects concern the actors of the repository system, while the technical aspects concern the processes to be done inside the repository system to promote re-use.

Brereton and Budgen [20] have identified two actors as a part for establishing a framework for component-based software development, namely, provider and integrator. The component's provider addresses the aspects to ensure that the provided component is usable by a tool and can be easily understood by a component's integrator. The component integrator concerns with aspects of finding re-usable components and solving potential functional or not-functional dissimilarity to the needed specification. The framework seems to establish a channel of communication between the provider and the integrator in order to facilitate better support for re-use. Vitharana [146] identified three actors for a CBSD environment, namely; component developers, application assembler, and customer. The component developers are those who build components and make them available for re-use. Application assemblers re-use components and integrate them into their system that is being built. Customers are those who only use the application as a whole to serve their purposes. DeLine [38] identified three actors for a CBSD, namely provider, integrator, and packaging specialist. The provider is the developer of a software component. The

integrator is the one who re-uses a component. The packaging specialist is an expert who is aware of a given packaging technology.

Component's integrator in Brereton and Budgen [20] terminology is nothing more than a re-user who is trying to find some functionality to re-use into the system under development. This research adopted partly the two views identified by Brereton and Budgen [20] and Vitharana [146] (i.e. provider and re-user) in a more general manner. In a large development environment the separation between integrator and packaging specialist described by DeLine [38] might make sense, however, from a general point of view both actors (i.e. integrator and packaging specialist) are simply represented as playing the role of a re-user. So, this research identified three actors for the ideal repository system:

- **The component provider**: component providers could be the developer of the component being deposited or could be another repository system, for example linked to one of the open-source repository systems (e.g. Sourceforge.net). Considering component providers in a repository design is essential in order to understand the possible mechanisms by which components are going to be processed after being deposited inside the repository. For example, if components are supplied from another repository system or component providers were persons other than their developers, then there must be an automated mechanism employed in the repository design to extract the required details by the repository system from the deposited component for re-use. Whereas, if the providers were component developers then there might be a need to specify a set of questions that they could answer prior to being able to deposit the components into the repository.

- **The component re-user**: this user is concerned with obtaining components for re-use. The activities that a re-user performs in order to obtain components include searching, selecting, verifying, and re-using. The searching is done by providing the criteria that reflect the specification of the system to be build to a repository system. Selecting components concerns finding parts to fit into a construct of a jigsaw puzzle [66]. The re-user needs to verify that a component that is found in the repository satisfies the specification of the system to be developed in order to be able to re-use it [108]. After a component is verified as matches the desired specification, the re-user can then re-use that component by integrate it into the system under development. From the perspective of a repository system, a re-user's view concerns the process of mapping, through re-factoring, what is inside the

repository to match their requirements and also delivering fully working components to a re-user. Mapping components to the re-user's requirements is important to ensure that components conform to all the requirements of re-users. Delivering fully working components is important to re-users as components that do not compile, for instance, are of no value.

- **The repository engineer**: this user is responsible for preserving the repository system in terms of maintaining the organizing scheme by allowing new classifiers to be added to the organizing scheme in order to ensure its flexibility. Also the repository engineer is responsible for ensuring the extendibility of the repository system by allowing new tools (e.g. re-factoring tool, compilers) to be added into the repository to extend its functionality.

The technical aspects that relate to the internal processes (e.g. identification, modification, classification) within the repository system are discussed in detail in the remaining of this chapter.

The work presented in this chapter concerned surveying the literature that corresponds to the identified characteristics of the ideal repository system to denote the extent to which current work achieved these characteristics. So, part of the surveyed work was considering the different characterization of software components in the literature in order to identify whether there is proper characterization that suits the need of the organizing scheme of the ideal repository system. The characteristics of an organizing scheme will be described in the next section and will form the basis for analyzing the surveyed work.

3.2 Organizing Components

The act of organizing components, in its very natural interpretation, is the process of grouping together components that share similar characteristics. The main purpose of organizing components is to explore and understand the various characteristics of components that re-users can utilize to filter their searching criteria to assist finding them. Organizing components is potentially useful in cases where the number of components is large in order to decrease the difficulty of finding a suitable one among a number of components.

Observing examples of organizing components in the literature suggested categorizing the available approaches into two general types based on the way that classifiers are generated for organizing components: classification schemes and indexing schemes.

Classification schemes define classifiers through number of categories defined by human, usually referred to as facets [72] in the literature. The defined categories are resulted from thorough analysis by experts to an application domain to capture all the relevant characteristics of components. Components can then be organized based on similarity in their exhibited characteristics. Indexing schemes are built on generating classifiers based on recording occurrence of words or phrases in software components or their corresponding documentation and analyzing their semantical meaning to identify possible relationships between terms (e.g. using Visual Thesaurus [1]). Components are organized using indexing schemes based on identifying similarity in the representative terms of components.

In the literature, classifying and indexing schemes are mixed together. Some authors use the term classification schemes to refer to what is described in the above paragraph as indexing schemes and vice versa. According to the ISO/IEC 11179 Metadata Registry (MDR) standard [74], classification schemes are categorized into: (i) keyword based, (ii) thesauri based, (iii) taxonomy based, and (iv) ontology based. Ostertag *et al* [118] categorized the different approaches of organizing components into: (i) free-text based, (ii) facet based, and (iii) semantic-net based. Milli *et al* [113] categorized classification schemes into: (i) keyword and string matching based, (ii) facet based, (iii) signature matching based, and (iv) behavioural matching based. Frakes *et al* [50] categorized organizing methods into: (i) free-text and keyword based, (ii) enumerated, (iii) facet based, and (iv) attribute-value based. Cechich and Piattini [29] have categorizes classification schemes into three types: (i) taxonomies and ontologies, (ii) semantic-net based, and (iii) learning based. Interested readers can refer to the original references (i.e. [74] [118] [113] [50] [29]) to find descriptions of the various types of classification schemes.

The above schemes can be mapped to the categorization proposed in this research as follows:

- Classification schemes: taxonomy, facet based, enumerative, and attribute-value based.
- Indexing schemes: keyword based, thesauri based, semantic-net based, ontology based, signature matching, and behavioural matching.

There are some characteristics associated with both types of organizing schemes that can be used to discriminate between classification and indexing schemes. Sections 3.2.1

and 3.2.2 (coming next) describe the characteristics of both types of organizing schemes and present a number of examples corresponding to every type.

3.2.1 Classification Schemes

The key characteristics of classification schemes include:

- Identifying different interesting dimensions of software components that capture distinct view points about the components. For example, a classification scheme can define categories to capture functional characteristics, architectural characteristics, and usage related characteristics.
- Adding new classifiers to a classification scheme can be simply achieved by considering a new category for the scheme. This allows classification schemes to work well in the case of classifying software as the variety of categories can help to ensure that at least one of the available categories in a scheme may match that of components re-users.
- Classification schemes make generating hierarchies of *types* -a type can be any thing that has properties (e.g. human, car, system)- easier as it precisely describes the characteristics of a *type*. So, a general *type* can be split into more specific *sub-types* by identifying the specific characteristics that are not part of a parent *type* in a hierarchy and also discriminate *sub-types* from each other. The tree of life is a good example expressing the idea classification schemes. So, a main *type* is Mammal that has, for instance, the characteristics: eat and drink. New *sub-types* called Animal and Human can be defined as inherited from the Mammal *type* which indicates that both Human and Animal can eat and drink. However, Human can describe some distinguishing characteristics that discriminate its *sub-type* from Animal; for example, one characteristic of Human being that Human can think while Animal cannot. So, the ‘ability to think’ characteristic distinguishes Human from Animal *sub-types*, and also indicates that Human is a new generation of Mammal that exhibits new characteristics not satisfied by Mammal.

This section presents examples of a number of classification schemes identified in the literature for organizing components.

3.2.1.1 Prieto-Diaz and Freeman Classification Scheme

Prieto-Diaz and Freeman [122] established a classification scheme for classifying software components by means of identifying a set of facets to represent components. They

proposed six facets for the scheme, three facets for characterizing functionality and the remainder for describing the environment where components were originally used.

The functional facets include:

- Function, describing what components do
- Object indicating object manipulations by functions
- Medium indicating the place where action is executed.

Whereas the environmental facets include:

- System type, indicating functional or application-independent area
- Functional area, indicating application dependant activities
- Setting, indicating where actions are performed and application used

Following this classification scheme, software components are classified based on selected vocabularies that represent values of their corresponding facets. The scheme allows re-users to provide a set of parameters to search through the classification scheme and selects and recommends software components that match exactly or closely to the provided parameters. The parameters are part of the interface used for the presented system [140]. For example, a certain component might be classified according to the tuple <compare, descriptors, stack, assembler, programming, software shop> that correspond to the values of the facets <function, object, medium, type, functional area, setting>. A controlled vocabulary is used to avoid the problem of synonyms that might cause different descriptions to be produced for the same component. For example, the descriptors <move, words, file> and <transfer, names, file> might be two different descriptors for the same components in term of functional facets. So if a component was classified by the first set of values of facets then the re-user who provided the second set of values may not able to find that component. Therefore, a thesaurus is used to unify the description of components to reflect the same meaning. So, if re-users type “move” it will reflect “transfer” and “copy” as they are the potential synonyms in the sense that all of the terms describe movement from one location to another. A weighted conceptual graph is used to measure closeness by conceptual distance among terms that relate to a facet to determine similarity between a query made by re-user and the actual descriptors for software components.

3.2.1.2 IBM Classification Scheme

IBM's software development environment is a heterogeneous set of developers spread all over the world. They develop systems ranging from operating systems to IDEs, business and medical systems and each may be written in different programming languages. Building software with re-use has become common practice in today's IBM software development [121]. Due to the diversity in nature of the development environment the facet based classification scheme is selected to classify component for re-use.

IBM adapted the facet approach proposed by Prieto-Diaz and added their own facets to classify re-usable components. Components in IBM are packaged with all the required information (e.g. integration instructions, documentation). The provided information can assist re-users to evaluate and understand components. A partial list of the IBM facets is displayed in Figure 3.2 [121].

A thesaurus is used to help unifying terms into consistent structures, so components are identified by a single value of facet (referred to as key in IBM terminology), and the value is used to identify a number of closely related terms that are marked as potential synonyms.

Facet	Description	Examples
Algorithm	Technique to perform an action.	bubble, merge
Application Domain	Broad area of application.	advertising, aerospace
Certification Level	IBM quality rating.	Certified, as-is
Component Size	Component size in LOC or words.	512 LOC, 8245 words
Data Structure Characteristics	How the data object is implemented.	bounded, directed
Development Standards	Compliance with standards.	ISO 9000, DoD-2167A
Function	What the component does.	sort, add
Object	What the component acts on.	buffer, array
Implementation	Describes various techniques.	sequential, dynamic
Implementation Language	Programming language.	C++, Ada
National Language.	National language	English, French
Proven Hardware	Tested computer platforms.	R/6000, S390
Proven O.S.	Tested Operating Systems.	AIX, OS/2
Support Level	Guaranteed service provided.	Limited, Full

Figure 3.2: Partial listing of IBM facets [121]

3.2.1.3 Sametinger Classification Scheme

Sametinger [129] classified software components based on four facets: (i) programming interface, (ii) user interface, (iii) data interface, (iv) and component platform. The programming interface facet is concerned with the various forms of component compositions, and is considered to be one of the most significant aspects for component re-use. Sametinger identified eight types of component composition:

1. external/internal
2. textual
3. functional
4. modular
5. object-oriented
6. sub-system
7. specific platform, and
8. open platform composition

Each form of composition addresses some aspects that concern re-using software components.

The user interface facet addresses the interactivity between software components and their end-users. Two types of user interface are distinguished: command-line; and graphical user interface. Sametinger claimed that components with command-line user interfaces are easier to re-use than components with graphical user interfaces. A possible justification of that might be that command line user-interfaces do not usually require special implementation of the protocols of interaction with users as opposed to graphical user-interfaces where a complex mechanism of interaction must be established.

The Data Interface facet categorises the format and structure of the data that components take as input and provide as output. Data formats and structure are important as far as re-use is concerned. Components with different assumption about their data formats and structure might raise run-time errors indicating data mismatch.

All the previous characteristics were considered by Sametinger as specific to components. Sametinger emphasized that matching a component's specific characteristics is not enough to re-use components successfully in a system as they need to conform number of characteristics that are required by the system (e.g. different run-time

environment) in which components can work. So, established capturing the characteristics of the system where components can be re-used by the platform facet. The Component platform facet gives an indication of the kind of architecture that components were originally worked in. Five values are identified: hardware, operating system, programming language, frameworks, and programming environment. Identifying these values is essential in a classification schemes to support component re-use.

3.2.1.4 Ali and Du Classification scheme

Ali and Du [5] established a mechanism to classify software design artefacts, namely object-oriented design models, hence facilitating their re-use. Ali and Du considered design models as an abstract representation of software system at the design level – the design models are primarily created to help in understanding system structure and operations from the design point of view. The model captures the design of components in the blueprint of the system to be built, the services they provide, and the relationship between them.

They proposed six facets to represent and classify design models for re-use, they are summarized in Figure 3.3 as appeared in the original reference [5].

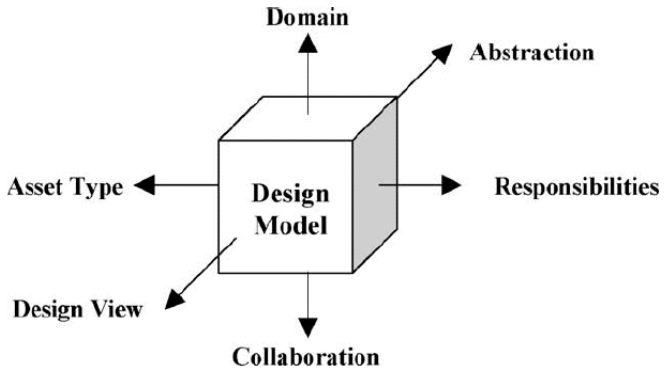


Figure 3.3: Facets for describing a design mode [5]

The description of the six facets is as follows:

- Domain: this facet captures the problem space in general that is addressed by a design model. For example, the possible values for this facet could be “Banking”, “Industry”, “Hospitals” and so on.
- Abstraction: this facet lists potential domain-specific terms that capture the possible key functional aspects of design models within different domains. For example, the terms “account”, “deposit” and “transfer” could be used to represent a design model in a banking domain.

- Responsibilities: responsibilities indicate the functional requirements of the system as whole. For example, the responsibilities of an ATM system can be described by the type of transactions a user can perform (e.g. deposit, withdraw, and check balance). This facet describes terms associated with actions that capture the functional characteristics of a design model.
- Collaboration: every object model, that represents components in a system, must interact with other objects in a system. The interaction represents the contract between two objects. At the design level, collaboration between object models is captured by design patterns. Therefore, this facet captures the design patterns [55] of design models.
- Design view: this facet captures the analysis and design decisions of classes of a system and their collaborations. For example, possible values of the design view facet can be class diagram, object diagram, sequence diagram, activity diagram, collaboration diagram, and stat-chart diagram.
- Asset type: the type of software design artefacts is indicated in terms of their variability and granularity. This facet contains terms that reveal the type of software design models, and lists design models in terms of their size and complexity. Possible values for this facet could be system, framework, and template.

Design models are classified based on terms identified from their design documentations that include the analysis and requirement specifications document. For example, the terms that represent the facet abstraction can be extracted from the class diagram. Also, the terms that represent the responsibilities facet can be obtained from different sources, such as important class methods, scenarios, sequence diagrams and use-cases that represent the functional requirements of a design model. One significant advantage of Ali and Du's classification scheme is its ability to capture the relationship between design models, so during retrieval of one design model from a repository, other design models that have some similar values of facets are retrieved for re-users as well.

3.2.1.5 Ugurel Classification Scheme

Ugurel *et al* [145] developed an approach for automatic classification of software components based on extracting characteristics related to programming languages and application domain facets. Their approach is centred on the Support Vector Machine

(SVM) [16] developed for automatic text classification. The characteristics are selected using the notion of expected entropy loss [3].

The characteristics correspond to programming language facet extracted to tokens in the source code and specific words in the comments. Tokens are defined to be any alphabetical sequence of characters separated by non-alphabetical characters. Numerical values and operators are not considered as tokens. Based on the assumption that every programming language has some reserved words exclusively related to it, the programming language of components is identified by applying keywords matching against the list of keywords corresponding to the programming language facet. For example, the words *struct*, *void*, *sizeof*, *include*, *unsigned* were extracted from C/C++ components, so they were used to programming language identification.

The related characteristics of the application domain facet are extracted from words and lexical phrases from comments, README files in addition to the source code's header file name. For example, Ugurel *et al* identified the occurrence of the word "*calculator*" for mathematics related applications, "*high score*" for games related applications and "*database*" for database related applications.

3.2.1.6 Yacoub, Ammar, and Mili Classification Scheme

Yacoub *et al* [152] classified characteristics of the components according to three main dimensions: (i) human-related characteristics, (ii) external characteristics, and (iii) internal characteristics.

The human related dimension is informal description of components that concerns the following categories (i.e. facets):

1. Age: this facet is to reflect the status of a component, whether it is newly generated or mature. Re-using newly generated components might involve some risk as there is no recorded re-use history, whereas mature components that have been re-used extensively are more desirable.
2. Source: this category indicates the producer who developed a component, for example, IBM or HP. This characteristic is useful to re-users who are keen to re-use components from selected suppliers.
3. Level of re-use: this category specifies the type of component that can be re-used, for example, requirement & specification, design, code, and tests.

4. Context: describes the domain and applications in which components can be used.
5. Intent: this facet describes the main purpose of generating components by stating the set of problems they solve.
6. Related components: this facet lists components that solve similar problems to the one described.

The second dimension that Yacoub *et al* considered in their classification scheme concerns the external dimension of software components that describes their interactions with other components and the underlying platform where components reside. The set of facets are as follows:

1. Interoperability: this facet identifies the mechanism of interaction with other components in a system, and determines how to call/invoke a component (e.g. RMI, or RPC) and the interaction direction (e.g. from client to server).
2. Portability: this facet describes the mechanism by which components can interact with their underlying platform (e.g. hardware, operating system, sub-systems). It is a property of components that specifies what kind of systems components could work in.
3. Role: the role of components specifies their actions within a system. So components could be either active, i.e. affect other components, or passive, i.e. affected by other components in a system.
4. Integration phase: this facet specifies the time of integration to a system as either development-time where components are compiled in a system or at run-time where components can be dynamically loaded-to and unloaded-from a system.
5. Integration framework: components are integrated together to form a complete system, however, sometimes components cannot interact directly with each other, but through an underlying framework that virtually connects components with each other to form a system. This facet identifies the underlying framework that a component uses to interact with other components in a system. Some examples of frameworks are: CORBA [136], EJB [9], or web services [64].
6. Technology: there was not enough information explaining what this facet means, however it is inferred that it relates to some aspects about the architecture of the system where components are used, for example, object-oriented system.

7. Non-functional features: this facet describes aspects other than functional characteristics of software components such as a component's security, performance and reliability issues.

The third dimension that is considered in the Yacoub *et al* classification scheme is components internal which describes the sub-components of components.

1. Nature: this facet describes where components can be used in a development process. For example, specification components, design components, source-code components, or executable-code components. It seems that this facet is similar to the level of re-use facet identified in the human related facets as it discusses the type of components again.

2. Granularity: this facet describes whether components are coarse-grained or fine-grained based on their sizes (e.g. LOC) or based on component type as design artefact is considered as coarse-grained while source-code components are fine-grained. Yacoub *et al* referred to the work done by Digre [40] who classifies components from their business perspectives into enterprise, domain subsystem, domain object, and semantic primitives.

3. Encapsulation: describes the decision hidden inside components such as specification decision, design decision, or implementation decision.

4. Structural aspects: this facet describes the internal participating sub-components of certain software components. For example, a component in object-oriented programming might have the structure of collaborating sub-components represented as the number of classes structured according to the specific order required by the component.

5. Behavioural aspects: Yacoub *et al* specified two values for this facet, stateless behaviour and retrospective behaviour. The stateless behaviour describes a component's responses to specific set of inputs, while retrospective behaviour describes a component's responses to a sequence of actions.

6. Accessibility to source code: this facet makes the assumption that components are binary code that may or may not be attached to a source-code component. So if the source code was available with a binary-code component then this indicates the possibility of modifying the component.

3.2.1.7 Kienle and Muller Classification Scheme

Kienle and Muller [83] established a classification scheme for characterizing software components. They assumed that components can be IDEs, domain-specific tools, application generators, compound documents, frameworks, and libraries. The Kienle and Muller classification scheme is composed of six main facets as follows:

1. Origin: this facet identifies the original supplier of components; the supplier can be either internal or external. 'Internal supplier' indicates that the supplier of components is the same as the one building the complete system. External indicates that components are obtained from a third party.

2. Distribution Form: this facet describes whether components are modifiable or not. The availability of source code was assumed as a factor to enable modification. So the possible values for this facet are black box, white box, or glass box. Black-box components indicate that source code is not available, while white-box components denote that the source code is available and also modifiable. Glass-box components, however, indicate that the source code is available but it is not modifiable.

3. Customization Mechanisms: this facet implies the possible customization that re-users might apply to components. Two possible values are identified for this facet, non-programmatic and programmatic. Some examples of non-programmatic customization are editing parameters in start up and configuration files, or with direct manipulation at the GUI level. Programmatic customization involves mechanisms to extend the behaviour of components, for example, via API in Java. Programmatic customizations vary from black-box and white-box components, as in the former customizations could be done via API or scripting, whereas in white-box components customizations can be simply done via source code modification.

4. Interoperability Mechanisms: this facet describes whether components interact with other components or not. The possible values for this facet are: no interoperability and interface interoperability. No interoperability indicates that components do not interact with any other components, they might only interact with the end-user. Interface interoperability indicates that components offer some interface for other components to establish interaction. Three types of interfaces are distinguished: data, control, and presentation.

5. Packaging: this facet denotes how components can be used. Two possible values are identified, stand alone component and non-stand alone component. ‘Stand alone component’ indicates that a component is an application; further refinement to this value can be whether the stand alone component is interactive or batch. Interactive means it interacts with users while batch indicates that there is no user interface for the stand alone component. The non-stand alone component value of the facet indicates that a component must be integrated, and might need to be customized, in a system to work.

6. Numbers of Components: this facet identifies the number of components in a system. The possible values of this facet are: single or multiple. The value ‘single’ indicates that a system is composed of one important component; single component often referred to as a stand alone application. The value ‘multiple’ indicates that a system is built from several components. Components in a system can be either homogeneous, meaning that all components in a system conform to a single interaction standard, or heterogeneous indicating that components in a system conform to different interaction standards or architectures.

3.2.1.8 Morision and Torchiano Classification Scheme

Morision and Torchiano [114] have established a classification scheme for COTS products. They classify the characteristics of COTS components according to four dimensions: (i) Source, (ii) Customization, (iii) Bundle, and (iv) Role.

The first dimension (i.e. source) concerns defining where the components come from and the implication of obtaining that component from its supplier. This dimension involves the following facets:

1. Origin: this facet identifies how components are developed by their vendors. Possible values for this facet are: in house, existing external, externally developed, special version of commercial, and independent commercial.

2. Cost and property: this facet describes whether a component is obtained subject to a cost or free of charge. Also, the facet defines the form of the obtained component whether it is in source code or in binary code format. The possible values of this facet are: acquisition, licensed, and free.

The second dimension (i.e. customization) specifies how much a component can be modified and how much work is needed to modify the component. This dimension defines the following facets:

1. Required modification: this facet describes whether a component can be modified or not. The possible values for this facet are: extensive reworking, internal code revision, customization, parameterization, and minimal.
2. Possible modification: this facet refers to the possible internal customization to a component. The possible values of this facet are: none or minimal, parameterization, customization, programming, and source code.
3. Interface: the facet the ways that a component's interfaces are defined. Te possible values for this of this facet are: none, documentation, API, Object-oriented interface, and contract with protocol.

The third dimension (i.e. bundle) specifies the form in which component are delivered to re-users. This dimension defines the following facets:

1. Packaging: this facet defines the architecture of the component. the possible values of this facet are: source-code component, static library, dynamic library, binary component, and stand alone application.
2. Delivered: this facet defines whether a component should be delivered or not. For instance, a Java compiler is not a component that needs to be delivered with a Java class. Possible values of this facet are: non-delivered, partly, completely.
3. Size: developers who are concerned about the performance of their system's execution might need to consider this facet. Possible values are: small that is less than 0.5 MB, medium that is between 0.5 MB and 2 MB, large that is of size 2 MB, and huge that is more that 20 MB.

The last dimension (i.e. role) defines the characteristics that a component assumes about the system in which it is going to be re-used. This dimension defines the following facets:

1. Type of functionality: this facet specifies whether the functionality provided by a component is general over multiple application domains (e.g. spell checker, web browser) or it is specific to a single domain (e.g. banking). Possible values of this facet are: vertical and horizontal.

2. Architectural level: this facet identifies type of a component from the architectural perspective. Possible values of this facet are: operating system, middleware, core, UI, and support.

Figure 3.4 illustrates an example of classifying three COTS components using the proposed classification scheme as depicted in the original reference [114].

Attribute	COTS Product		
	MS Windows NT	Samba	MS Chart Control
Origin	Indep. Comm.	Indep. Comm	Indep. Comm.
Cost & Property	License	Free	License
Required Modification	Parameterization	Parameterization	Minimal
Possible Modification	Programming	Source code	Programming
Interface	API	API	Contract
Packaging	Standalone	Standalone	Binary Component
Delivered	Completely	Completely	Completely
Size	Huge	Large	Small
Functionality	Horizontal	Horizontal	Horizontal
Architectural Level	OS	Middleware	UI

Figure 3.4: Example of Classifying COTS Components [114]

3.2.2 Indexing Schemes

The key characteristics of indexing schemes include:

- Easily automatable as it is purely based on extracting terms and phrases and analyzing their semantic meaning.
- Flexible in nature as there is no fixed set of pre-defined categories required to organize components. New indices are generated automatically whenever new ways of organizing components are needed.
- Can describe complex relationships between indices which can be beneficial in finding more potential re-usable components.
- Precise as they work at the source code level.

This section presents examples of number of the indexing schemes identified in the literature for organizing software components.

3.2.2.1 Maarek, Berry and Kaiser Indexing Scheme

Maarek *et al* [101] proposed a mechanism to identify indices automatically by analyzing natural-language documentation, represented in the form of manual pages or comments usually attached with the software components, and using them to build free-text indices to characterize software components. Maarek's approach is not concerned with extracting

indices from the source code itself (e.g. syntax of programming languages, method names) but depends primarily on the attached documentation.

Maarek *et al* have built a tool, GURU, that extracts indices from the documentation associated with the software components to be organized utilizing information retrieval technique (i.e. free-text searching) based on the notion of lexical analysis that is primarily based on Latent Semantic Analysis (LSA) [87] that describes the relationships between words. The frequency of appearance of words or phrases is considered as a measuring criterion to identify significant terms that might be useful for use as representative indices.

Maarek also provided a mechanism to facilitate browsing components by clustering them in a hierarchy based on the identified indices resulting from document analysis. The similarity between components in the hierarchy is identified using a numerical measure called a dissimilarity index [101]. There is no category established by Maarek's approach for browsing as the browsing is assumed to be done in a bottom-up fashion. Re-users are assumed to use the searching facility provided by the GURU tool to search components based on some criteria at first, if they do not find relevant results then they can browse through a component hierarchy starting from a component retrieved from the previous search that they believe has some relation to what they need and moving up towards the top of the hierarchy. For example, suppose that a re-user wants a component that performs the functionality "identify a process in UNIX". Imagine that one of the retrieved results was "kill", then the re-user can exploit that result as a starting point for browsing the hierarchy and explore components that are classified relevant to the "kill" component in order to find an exact or nearest match to the required functionality.

3.2.2.2 Ye and Lo Indexing Scheme

Ye and Lo [155] have developed an approach to index and organize software components automatically. They utilized the free-text indexing mechanism and the self-organizing map (SOM) [85] technology, a mechanism used for organizing documents, for building their indexing scheme.

The proposed approach is centred on extracting significant features (i.e. keywords) that reflect functionality from component descriptions and not from the components themselves (i.e. source code). They require that the descriptions attached to software components must be accurate in order for their approach to be effective.

Documents are indexed using free-text indexing mechanisms, and then vectors describing components are generated for each component. Elements within vectors represent the extracted features from component descriptions. Some features are significantly related to component functionality while others are less significant, and *weighted input vector* mechanism is used to specify the most relevant features that represent components functionality from those that are less relevant based on their significance to components. The significance of features is measured by counting the number of occurrences in the documentation attached to the components. Figure 3.5 illustrates an example of feature weights in different UNIX commands using the *weighted input vector* mechanism is appeared in the original reference [155]. From that figure, it is observed that the most relevant functionality for the command ***sed*** is *edit*, ***cp*** is *copy*, ***whatis*** is *manual*, ***ex*** is *edit*, and ***vi*** is *edit*.

Feature	Feature weight in different commands					
	sed	tee	cp	whatis	ex	vi
Edit	0.5279	0	0	0.3587	0.6021	0.4123
Copy	0.1706	0.1749	0.1699	0	0	0
Output	0.1499	0.3076	0	0.1238	0.3509	0
Manual	0	0	0	0.6109	0	0.1592

Figure 3.5: Commands and their Feature Weights [155]

After all features have been weighted, the generated vectors for various software components are deposited into the SOM as multi-dimensional vectors. SOM is capable of mapping multi-dimensional input vectors into a two-dimensional grid. So, similar features are organized close to each other on a grid after comparing the input of a vector's features against a specific cell on a grid that is associated with n-dimensional reference vectors.

Ye and Lo claimed that their approach supports flexibility, extensibility, and visualizability as follows:

- Flexible as software components are not grouped and organized based on fixed categories, instead components can be grouped in different ways depending on their classifier's perspective.
- Extensible as it can add new classifiers to organize software components whenever new generations of components not belonging to any classifiers from the available ones in the scheme appeared.

- Visualizable as the resulting map and the semantic relationships amongst components can be clearly identified and illustrated.

3.2.2.3 Tangsipairoj and Samadzadeh Indexing Scheme

Tangsipairoj and Samadzadeh [142] have adapted the SOM technology and extended it to include hierarchal representations of data. GHSOM is a further enhancement over SOM as it concerns dynamic SOM modelling that can build a hierarchy of multiple layers of several SOMs. The depth of the hierarchy is determined based on the level of refinement that might be required by features of nodes in the first SOM. So the top level SOM is composed of coarse-grained features (i.e. keywords) while the low level SOM in the hierarchy captures more fine-grained features that are refinements of the top SOM. Figure 3.6 illustrates the architecture of GHSOM as appeared in the original reference [142].

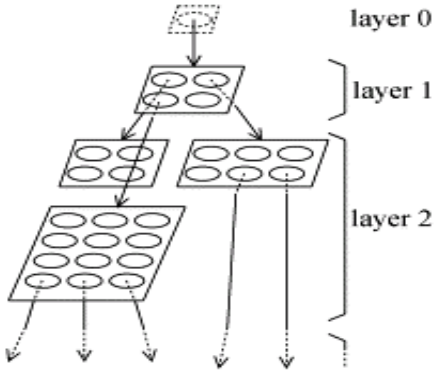


Figure 3.6: GHSOM Architecture [142]

For example, the SOM at layer 0 can be used to organize components based on their subject areas (e.g. domain information). Every index of the SOM at layer 0 might be refined to another SOM at a lower level. So, if an index in the SOM at layer 0 is about banking systems then indices of the refinement SOMs at lower levels can represent information about types of accounts, names of branches, interest rates and so on. Each index then can be refined to a set of SOMs at a lower level as necessary.

The refinement of features can help to capture more focused keywords to represent software components accurately, in addition, the hierarchical nature of GHSOM can help in relating components that belong to same domain.

3.2.2.4 Lin, Amor and Tempero Indexing Scheme

Lin *et al* [96] established an automated indexing mechanism for organizing Java API classes in order to ease finding them for re-use. The main intention behind developing the

indexing mechanism for Java APIs was based on their belief that the numerous numbers of APIs available nowadays might hinder re-using them. As a result, Lin's main contribution was to improve the efficiency of finding classes in the Java API as compared to browsing through the API's documentations manually.

Lin's approach is mainly based on the *Latent Semantic Indexing* (LSI) approach [87] and provided as tool, *Prophecy*, designed to work as a plug-in for Eclipse IDE. *Prophecy* extracts details about APIs from their corresponding JavaDocs, in fact, *Prophecy* indexes API classes based on indexing their respective JavaDocs. The process of generating indices is as follows. A term representing a general concept that is identified as semantically common through different documents, after applying latent semantic analysis, is selected. After that, a number of representative terms are identified from the documents that are believed to be addressing different dimensions of the identified concept, these terms are not necessarily synonyms. Some of the representative terms could be the important terms appeared frequently in various documents. In fact, a key feature of the *Prophecy* indexing mechanism to identify relationships between indices is the consideration of words that appear frequently together in a document as a way to relate documents together. For example, if the words "String", "Append", and "Stringbuffer" appeared together continuously in several documents, then if a searching query containing the words "Append String" is provided then the set of documents containing these two terms, as well as documents that contain just the word "Stringbuffer", will be retrieved to the re-user.

3.2.2.5 Kawaguchi, Garg, Matsushita, and Inoue Indexing Scheme

Kawaguchi *et al* [82] established a mechanism for organizing open-source software components automatically. Kawaguchi's approach is one of originating categories and assigning a number of representative terms to every category. The approach generated is an attempt to generate a classification scheme automatically, however, due to the operational nature of generating categories the approach is nothing more than an indexing scheme. Kawaguchi's approach is based on the following assumptions:

- Categories generation is based on processing only source code.
- Allow components to be categorized by multiple categories.
- Categorizing terms are not pre-defined as they are generated automatically.

Kawaguchi has built a tool, MUDABlue, that performs the generation of categories for organizing components. The tool is based on Latent Semantic Analysis (LSA) [87] to

identify relationships between terms. The MUDABlue tool targets variable names and methods names, referred to as identifiers, assuming that names of variables and methods should reflect the behaviour of source code. MUDABlue works as follows:

1. The tool extracts all the identifiers from the source code.
2. Count the occurrence of the extracted identifiers in all the available software components.
3. Remove the identifiers that recorded the least number of occurrences from any further analysis.
4. Apply LSA to identify the semantic meaning and relationships between identifiers.
5. Generate categories based on the result of the LSA of identifiers, considering identifiers that recorded the maximum numbers of occurrence. Similar identifiers are then grouped under the newly defined categories. For example, the terms “gtk_window”, “gtk_menu” might indicate that both terms can be grouped by GTK as a category.
6. Organize software components based on the identified categories.
7. Generate a titled description for every category derived from the indices that recorded maximum occurrence.

Figure 3.7 illustrates the steps followed by the MUDABlue tool to generate categories for components organization as illustrated in the original reference [82].

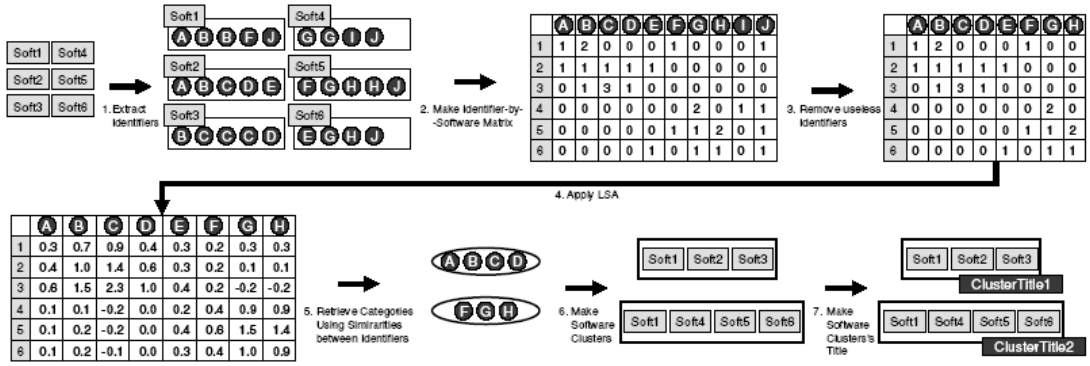


Figure 3.7: Generating Categories [82]

3.2.3 Analysis of Related Work

In the scope of the ideal repository system, the organizing scheme for software components should satisfy to the following characteristics:

- Automatable: the organizing scheme should allow classifiers to be generated automatically. Being automatable is important to make the organizing scheme self-maintainable in the sense that changing (i.e. adding or removing) classifiers can be generated automatically; also automatability allows components to be identified and organized automatically.
- Extensible: new classifiers can be added to the organizing scheme whenever new components that cannot be organized using the available classifiers in a scheme appear, so the scheme can support evolution.
- Flexible: it is essential that the set of classifiers (e.g. attributes in classification schemes and terms in indexing schemes) are not fixed and vary amongst components belonging to different requirements. This makes room for an organizing scheme to organize components that are of different types and levels of abstractions (e.g. high abstraction such as design artefacts and low abstraction such as source-code components).
- Re-usable: new classifiers can be defined by re-using old ones and adding the necessary characteristics to them in order to reduce the effort of defining new classifiers. For example, if one wants to describe new classifiers that capture sets of characteristics that had only been defined partly by a classifier in a given scheme, then being able to add new classifiers to a scheme building on the ones already available is a desirable feature in the ideal repository system.
- Describe relationships: it is a desirable feature that the organizing scheme of the ideal repository system can define the relationship between classifiers. One advantage of this is to retrieve potential components that re-users might also be interested in. For example, if a component has a design artefact that is organized under a design classifier then retrieving the component can also result in retrieving its corresponding design.
- Consider Architecture: in some cases, a component re-user might find a component that seems to be re-usable into their system as the component they found in a repository provides similar functionality as they need. However the re-user discovers later, when integrate the component into their system, that the component does not fit into their system due to architectural mismatch as discussed in Chapter 2. As a result, considering the architectural characteristics of software component is

important aspect to identify in an organizing scheme in order to understand the re-usability of software components.

- Source code availability: there might be cases where a re-user may find a component that satisfy their functional requirement but conform to different architectural requirement that they need. If the component can be modified to satisfy the architectural requirements of the re-user then it will be a perfect fit. The modification cannot be applied unless the source code is available. So, considering the availability of source code in an organizing scheme is useful to inform the re-user if the component can be modified before extracting the component from a repository and try to find out whether it is modifiable or not themselves.

Table 3.1 describes a summary of analyzing the examples of classification and indexing schemes against the characteristics required for organizing components in the ideal repository system.

✓ Satisfied X Unsatisfied ? No enough details		Automation		Extensibility	Flexibility	Re-usability	Relationships	Architecture	Source code
		Generate classifiers	Organize components						
Classification	Prieto-Diaz & Freeman	X	✓	✓	X	X	?	✓	✓
	IMB	X	✓	✓	X	X	?	✓	✓
	Sametinger	X	X	✓	X	X	X	✓	?
	Ali & Lu	X	X	✓	X	X	X	✓	X
	Ugurel	X	X	✓	X	X	X	✓	✓
	Yacoub *et al*	X	X	✓	X	X	X	✓	X
	Kienle and Muller	X	X	✓	X	X	X	✓	X
	Morision and Torchiano	X	X	✓	X	X	X	✓	X
Indexing	Maarek *et al*	✓	✓	✓	✓	X	?	X	✓
	Ye and Lo	✓	✓	✓	✓	X	X	X	✓
	Tangsipairoj & Samadzadeh	✓	✓	✓	✓	X	X	X	✓
	Lin et al	✓	✓	✓	✓	X	✓	X	✓
	Kawaguchi *et al*	✓	✓	✓	✓	X	✓	X	✓

Table 3.1: Analysis of Organization Schemes

Table 3.1 shows that the classification schemes generated by Prieto-Diaze & Freeman, and the one generated by IBM can organize components automatically while the other classification schemes cannot. The reason for that is that both classification schemes are

implementable while the other classification schemes are merely abstract and not implementable. The Prieto-Diaze & Freeman and IBM classification schemes identified categories in which their characteristics can be identified from the source code; the other classification schemes, however, lack precise characterization that can reflect the characteristics of concrete components. In contrast, all indexing schemes are implementable and can organize software components automatically.

With respect to the extensibility characteristic, classification schemes are useful to classify new evolutions of software component as classification schemes are built on selecting independent categories, hence new categories can be added to classify new generation of components. Indexing schemes are extensible as well, as new representative terms can be generated whenever there is a need to index a new generation of components that cannot be indexed using the current indices in a scheme.

The table denotes that classification schemes are not flexible; all schemes must be associated with a fixed set of categories for classifying components. The set of categories might also require a fixed set of values in some cases but it is not necessary. Indexing schemes are, however, flexible. There are no categories to use for indexing components as components are organized on the basis of some keywords that are selected based on monitoring their significance in a certain domain. The selected indices can be changed as necessary.

Regarding the support for re-use, none of the identified classification schemes support this characteristic. Classification schemes define facets that are distinct from each other and never define any relationship between them. Indexing schemes do not support re-usability as they do not define categories in the first place. Some indexing schemes might involve hierarchical structure of terms; however, the relationships between terms are based on lexical meaning of terms which may differ from one lexical analyzer to another.

With respect to the support for capturing relationships between classifiers, it is not clear whether the classification schemes developed by Prieto-Diaze & Freeman and the one developed by IBM support this characteristic. Both classification schemes employ LSA which describes relationships between terms, but it is not obvious whether these relationships are utilized to draw links between categories or not. None of the other described classification schemes support capturing relationships between categories as categories are completely independent of each other. Indexing schemes vary, some indexing schemes do not support this characteristic, such as the work by Ye & Lo, and

Tangsipairoj & Samadzadeh. The work by Maarek neither explains this feature nor provides enough information to indicate whether the approach can support capturing relationships between indices or not. The work by Lin *et al* and also the work by Kawaguchi *et al* describes explicitly that their approach is capable of capturing relationships between classifiers by identifying terms that appear together in different documents.

With respect to considering the architectural characteristics, all the classification schemes consider abstractly some architectural characteristics that relate to the system in which a component was developed to work in. In contrast, none of the available indexing schemes have considered any architectural characteristics. A possible reason for not considering architectural characteristics by any of the indexing schemes seems to be caused as architectural characteristics can not be represented using simple keywords. Keywords are used by the various indexing schemes manly to represent functionality.

With respect to the availability of the source code, the indexing schemes are fully automatable and identify precise set of terms that can be checked in the source code of a component. As a result, all the indexing schemes assume the availability of source code in order to organize components. In the contrary, not all the classification schemes require the availability of source code as they characterize components abstractly that only human can read and understand. However, the classification schemes by Prieto-Diaze & Freeman, IBM, and Ugurel have defined some representative terms that can be checked in a component's source-code to identify and classify components. So, their approaches required the availability of source code in order to organize components.

3.2.4 Observation

Classification schemes are useful to capture different interesting dimensions about software components in order to allow for understanding of components by re-users and also to facilitate their re-use. However, classification schemes lack the tool support to achieve automatic generation of classifiers for software components, which is one the desirable features for the ideal repository system. Most of the work presented in the classification scheme section showed the necessity of involving experts to do extensive domain analysis for the purpose of generating classifiers for component classification. In contrast, indexing schemes are fully automatable; human involvement is not necessary in the process of generating classifiers for software components. The examples presented in the section on indexing schemes expressed the feasibility of automating the process of

generating classifiers for components. However, all the generated classifiers are semantic-oriented which is primarily based on extracting terms and identifying their semantic meaning and relationship with others potential terms. A problem with this approach (i.e. indexing scheme) is that it cannot capture other aspects about components other than what is implied by the semantic meaning of terms. For instance, one example of classification schemes described a category to capture the possible interaction between components; interaction types cannot be identified using regular semantic analysis employed by most of the available indexing schemes. As a result, indexing schemes may not be sufficient for general component organization.

In general, the characteristics of classification schemes imply analytical perception while indexing schemes imply easiness. So, what is needed to achieve an adequate components organization in the ideal repository system is a scheme that captures the capability of both organizing schemes (i.e. classification and indexing schemes).

Another interesting observation with respect to the characteristics considered in the different classification schemes is that the characteristics are not solely functional based but there are also non-functional characteristics (e.g. architecture, performance, human related) that have been considered. Many of the classification schemes have defined some architectural characteristics to classify software components. For example, the classification by Sametinger defined different types of component interfaces that are not relevant to any functional aspects. Also, IBM used operating systems as a distinguishing architectural characteristic to classify software components. Observing the consideration of many architectural characteristics in the available classification schemes indicates the importance of defining the architectural characteristics of software components in order to support obtaining them; hence the observation is a valuable justification for the significance of the approach of this research (to be described in Chapter 4).

3.3 Re-factoring Software Components

A desirable feature that the ideal repository system ought to support is to allow mapping what is deposited (e.g. software components) into the repository system by providers (i.e. provider's view) to what re-users actually need (i.e. re-user's view). The mapping between provider's view and re-user's view concerns two dimensions:

- Modifying the component deposited by a provider to match the requirements needed by a re-user.

- Extracting a component from a standalone application with all its necessary dependencies.

Apparently, the mapping should consider the non-functional characteristics of a software component and not the functional characteristics, because if a component's functionality was not the one required by a re-user then there is no point in applying any modification to the component in order to reduce the gap between the provider's view and that of the re-user. So, if a provider supplied a component that matched the functionality that a re-user needed but was different in some non-functional characteristics, then applying modifications to the component to bridge the gap between the provider's view and the re-user's view, attempting to match the non-functional characteristics, can increase the re-usability of the supplied component. This mapping between the provided view and required view can be achieved by employing an advanced re-factoring mechanism.

Re-factoring is commonly known as the process of changing the internal structure of components without affecting their external behaviour [49]. Re-factoring is applied primarily for the purpose of optimizing components by removing unnecessary code (e.g. duplicated or dead code) [49] or re-structuring the code of software components [44]. It is been claimed that optimizing components can improve their quality (e.g. extensibility, modularity, re-usability, complexity) and also reduces maintenance cost [107]. One example of a kind of re-factoring is changing a variable name into something more meaningful, another example could be to turn the code within an "IF" block into a method or a function or even replacing the whole "IF" block with polymorphism (specific to object-oriented programming) [49].

A survey [107] of component re-factoring has captured various re-factoring mechanisms aimed at optimizing components. Beck and Fowler [49] defined re-factoring as the process of removing "bad smells" (e.g. duplicated source code) from software components. They described bad smells as "*structures in the code that suggest (sometimes scream for) the possibility of re-factoring*". They established 22 re-factoring mechanisms for overcoming bad smells. Balazinska *et al* [13] used a clone analysis tool to identify duplicated code that indicates re-factoring candidates. Ducasse *et al* [41] established an approach to detect duplicated code within software components by employing a language independent and visual approach.

Other work [148] [97] has established re-factoring mechanisms to generalize components in order to make them re-usable by tackling the problem of coupling between

software components. A number of studies [103] [24] [69] [46] have established re-factoring mechanisms to assist in the identification of design patterns from software components as a way of identifying design decisions from components and mapping them to the design stage.

The intended meaning of re-factoring in the ideal repository system has commonalities and slight differences to the more general perspective of re-factoring identified in the literature. The commonality is that both perspectives (i.e. the ideal repository system and the work identified in the literature) propose that component functionality is unaffected by any re-factorization. However, the difference is in the purpose of re-factorization. As stated at the beginning of this section, the ideal repository system employs a re-factoring mechanism to enhance re-usability of a component by bridging the gap between the provider's view and the re-user's view, whereas the purpose of re-factoring in the literature is primarily to optimize the performance and maintainability of a component [49]. The few works (e.g. [148] [97]) that consider increasing component re-usability by re-factoring components are concerned with re-use in general term and not similar to the view adopted by the ideal repository system. The intended meaning of re-factorization is in line with the notion of "packaging specialist" proposed by DeLine [38], but re-factoring is intended to automatically provide the necessary characteristics for fitting components into a system.

Therefore, there is a need to establish a new re-factoring mechanism that satisfies the requirement of the ideal repository system. As a result, establishing a mechanism to support the requirement of re-factorization in the ideal repository system constitutes part of the investigation in this research.

3.4 Software Repository Systems

This section surveys the related work in the literature that concerns establishing repository systems to support re-use, and also compare the characteristics of the developed repositories in the literature against the characteristics of the ideal repository system to evaluate the extent to which they achieve the ideal characteristics.

3.4.1 Overview

The term "repository" is sometimes used to mean a search engine. Although repositories usually have their own search engines, it is a place where data can be deposited, organized, and retrieved, whereas a search engine is nothing more than a mechanism to identify, according to certain criteria, items stored inside a repository.

Many repository systems are available. Some are in the public domain, such as open-source repository systems, while others are private to an organization. A number of key repository systems are discussed in the next sections.

3.4.1.1 +1Reuse Repository System

The +1Reuse repository system was developed by the +1 Software Engineering Corporation [65]. It supports re-using several types of artefacts such as designs, documentation, code, header files, test cases, test scripts, test results, and modelling information. +1Reuse repository stores and organizes projects corresponding to building systems. Every project is considered as a library of re-usable artefacts. +1Reuse repository supports re-using sub-modules (e.g. components from a system) of projects. Re-users can select an artefact from a project within the +1Reuse repository and the repository will download the selected artefact and all its associated files to their directory, which is a useful characteristic for re-users as it saves them the effort of figuring out dependencies manually, and resolves any name duplication problems. The references for this repository provide little details about the mechanisms used to organize projects and how re-users can search or browse projects to find re-usable artefacts.

3.4.1.2 CodeFinder-PEEL Repository System

CodeFinder-PEEL [68] is a repository system for organizing source code components written in the Lisp programming language. The designers of the CodeFinder-PEEL repository system identified three key characteristics that they believe must be supported by a repository system:

1. Utilize a tool to generate initial indices for organizing components starting with a number of randomly selected components.
2. Provide flexible mechanisms to search and browse the repository.
3. Tools to refine and adapt indices as re-users work with the repository.

The repository system organizes components by applying an automatic indexing mechanism, using a tool called PEEL [68] that performs the necessary latent semantic analysis (LSA). The PEEL extracts terms from definitions of functions, variables, constants, and macros of the source code and analyzes the extracted information to generate representative terms (i.e. indices) and identifies the possible relationships between the indices. The resulting indexing scheme is not final as the repository supports the facility to allow re-users to refine indices as they search for re-usable components, and thus

organizes components differently based on the refined terms. So, re-users are able to add new terms and remove terms that they think are not relevant for representing components. The repository builders claimed that a major advantage of employing such an adaptive indexing mechanism [143] is to avoid the cost of building complex organizing mechanism at the beginning of building the repository system, also to allow the repository to support evolution.

The CodeFinder-PEEL repository system assists re-users in selecting terms that are semantically near to what they typed in the search query. This feature is useful to teach re-users about the vocabulary used in the repository system to represent components and also to reformulate their search queries to specify precisely the required component.

3.4.1.3 The WebComposition Repository System

The WebComposition repository [54] system is aimed at supporting re-use of components for building web application systems based on the Components-Based Software Engineering (CBSE) approach [141]. The repository is composed of three main elements:

- Components store.
- Metadata store.
- Searching and browsing tools.

The Components store is a persistent storage place (e.g. database) that is responsible for storing and maintaining components. The Metadata store is responsible for storing data about components that is used by the organizing scheme of the repository system for organizing components. The searching and browsing tools are responsible for querying and inspecting the repository in order to find components of interest.

Components within the repository can be organized in different ways based on the Metadata store used for organizing schemes. One way of organizing components that a Metadata store might describe could be by the kind of information available with components such as their design, history, interactions with other components, semantic information, and documentation. This type of organizing scheme takes the form of a classification scheme in the sense that a number of categories should be defined by experts to capture the different dimensions of components. Another way employed by the repository is an adaptive indexing mechanism [143] that indexes components inside the repository based on learning from the actions of re-users recorded against using the

searching or browsing tools. The repository system is extensible in the sense that it allows new Metadata stores (i.e. classification and indexing schemes) to be added for classifying or indexing components differently. In addition, new tools can be added to the repository to help searching or browsing for components.

3.4.1.4 Sourceforge.net Repository System

Sourceforge.net is one of the largest open-source software development repositories. It provides free hosting to open-source software systems. Some of the organized systems in the repository are in beta versions and others in their final version. Sourceforge.net employs a version control system to manage storing and tracking all changes applied to source code by developers. The repository system established a classification and indexing schemes for organizing software components.

Sourceforge.net classifies systems based on some facets including:

- Topics (e.g. security, games, Database).
- Operating systems.
- Programming languages.
- License.
- Intended audience.
- User interface; translations.
- Database environment.
- Development status.

The values of these facets are provided by system developers at the time of initiating projects or depositing systems into Sourceforge.net.

Re-users can browse directly through the facets of the classification scheme, for example, all systems classified under Java programming language. Alternatively, re-users can use the provided free-text search engine that searches for matching keywords against the free-text descriptions of corresponding systems, and use facets for filtering results. Sourceforge.net has recently established a searching mechanism powered by Krugle Corporation [127] at the source code level. So, once a potential system is found in Sourceforge.net then a re-user can use the source code search engine to search for components within that system. The search engine is basically built on keyword matching;

the search can be filtered by specifying a value of the programming language used by a project in addition to specifying the expected occurrence of the matching keyword from within one of six options listed as, source code, comments, documents, function definition, function call, or class definition. Figure 3.8 illustrates the user interface of Krugle, the source code search engine in Sourceforge.net.

Figure 3.8: Krugle Search Engine

3.4.1.5 CompSrc Repository System

CompSrc [128] is a repository system supporting the re-use of source-code components that were developed in different environments and by different development teams. The repository employs a mechanism to represent components at a high level of abstraction using a meta-language called VDM-SL [48]. One of the key aspects of the repository system is to establish a uniform description of all the components in the repository.

The repository system is composed of four main components:

- Interface extractor tool.
- Components calculator tool.
- Indexing scheme for software components.
- Testing tool.

The interface extractor tool extracts details of the APIs associated with different software components and generates new versions of the extracted APIs represented in VDM-SL called abstract APIs. The generated abstract API is used then by the component calculator tool to identify possible functional composition between components and the order in which they need be composed. Two ways of component composition are identified: (i) composition by aggregation, (ii) composition by wiring. Composition by aggregation concerns combining the abstract APIs of the components to constitute a more complicated component whose API is the aggregation of the APIs of its components. Composition by wiring involves identifying data type similarities between different methods within different components. For example, if the return type of one method in component A is similar to the input parameter to another method in component B then the two components might be wired together to compose a new component AB. The CompSrc repository employs a testing tool, which is part of the component calculator tool, that

performs the compatibility check between the data type of methods within components. The Indexing scheme is purely text-based and is built on the *Latent Semantic Indexing* (LSI) mechanism [87] and it is responsible for organizing components by their corresponding abstract APIs in a database.

3.4.1.6 CRECOR System

CRECOR [90] is a repository system to facilitate re-using Enterprise Java Beans (EJB) [9] components. CRECOR provides the following support to components re-users:

- Browsing.
- Selecting;
- Analysing.
- Adapting.
- Deploying.
- Testing.
- Component Assembly.

The repository system utilizes a graphical user interface to display components and their relationships in the form of tree structures, so re-users can browse and select the desired component graphically. The component analyzer analyzes the deployment descriptors corresponding to Enterprise JavaBeans (EJB) [9] components that describe their conformance to the EJB specification. There are some other details such as methods signature and interface name extracted from components in the analysis process to support identifying components behaviour. The structure of the analyzed components and the signatures of their methods are used for identifying re-usable components and the context in which they might be re-used.

The component adaptation process includes renaming attributes/interfaces, adding new attributes/interfaces/methods, and replacing source code in methods. So, components might be modified to resolve any mismatched interface between components, for example, by modifying return type, parameter types, parameter order, and method name. The repository's adaptation mechanism allows extra attributes and methods to be added to components in order to match new requirements. Figure 3.9 illustrates the GUI of the adaptor utilized in the CRECOR system for the components adaptation process.

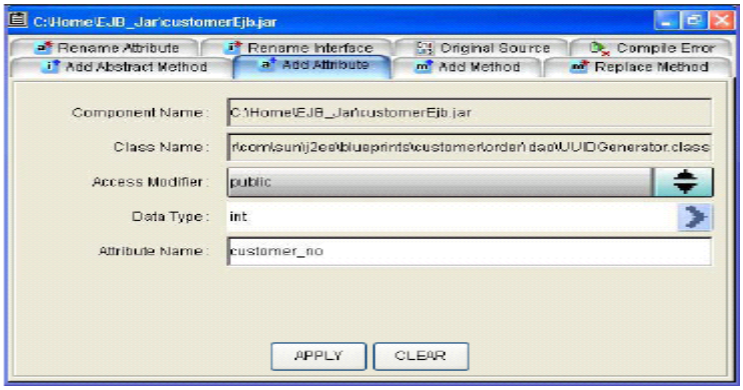


Figure 3.9: CRECOR Components Adaptor

The Component deployment process supports generating components deployment descriptors, packaging of components with their constructing files, and installing components in application servers (e.g. J2EE, JBOSS). The component testing process generates a web-based client program for a selected EJB component that allows re-users to test component functionality by supplying different parameter values and examine their impact on components behaviour. The assembler tool, COBLAT [92], composes components together to form a bigger composite component or a system. Re-users can utilize the graphical representation of components in the repository to drag and drop components in order to assemble a composite component.

3.4.1.7 CodeBroker repository system

Yunwen [157] has established a repository system called *CodeBroker* to support component re-use written in Java. The repository system is considered active, as compared to the other repository systems, in the sense that it works in the background of a development environment and provides advice about the availability of components while developers are busy writing code. The main objective that *CodeBroker* repository system addresses is to inform developers about the availability of components they might be interested in re-using without requiring them to search for components manually using the traditional methods. The repository monitors a developer's activities and provides possible re-use advice. Figure 3.10 illustrates a snapshot from an environment with CodeBroker taken from the original reference [157].

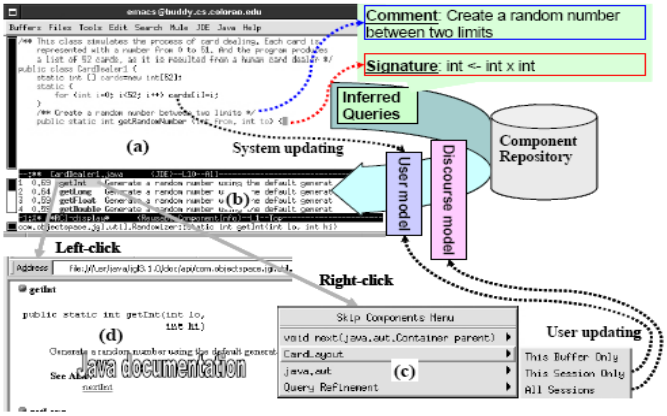


Figure 3.10: CodeBroker System [157]

The *CodeBroker* system is composed of an interface agent and a backend search engine running in the background of the development environment. The interface agent infers and forms searching queries automatically to supply to the *CodeBroker* repository by analyzing partially written code typed in by developers. The inferred queries are then passed over to the search engine which attempts to find components that match the supplied query. *CodeBroker* indexes components based on their JavaDocs. The repository delivers components whenever comments or source code written by developers are identified as matching one in the repository. *CodeBroker* does not deliver all components it finds but it uses a filtering mechanism to deliver the most relevant components. The filtering mechanism used in *CodeBroker* is based on a *discourse model* that allows a developer to specify components that are not of interest in a development session. The *CodeBroker* system learns from a developer's responses in a session and uses the information obtained from a session in future component delivery.

3.4.2 Analysis of Repository Systems

The various repository systems described in the previous section indicated the kind of support they provide to facilitate re-use, hence attempting to resolve some of the obstacles hindering re-use. This section is going to analyze the identified repository systems against the characteristics of the ideal repository system which are:

- Support organizing schemes that have the characteristics identified in section 3.2.
- Support re-factoring mechanisms that satisfy the characteristics identified in section 3.3.
- Support testing components against defined requirements. So, re-users can provide their requirements to the testing tool of the repository and the tool can check

whether a component matches these requirements or not prior to delivering it. Also, the testing tool can be utilized to search for components inside the repository. Re-users can provide the characteristics of the components they need to the testing tool, and the tool can examine the available components in the repository against the provided characteristics.

- Support delivering components with all their associated dependencies.
- Support evolution by allowing new tools to be added to the repository system to enhance its functionality.

Table 3.2 illustrates a taxonomy matrix that analyzes the characteristics of every repository system against the characteristics of the ideal repository system. The table shows tradeoffs between the various repository systems in their capability to support re-use.

✓ Satisfied X Unsatisfied ? No enough details		+1Reuse	WebComposition	Sourceforge.net	CompSrc	CRECOR	CodeFinder-PEEL	CodeBroker
Organizing scheme	Classification Scheme	?	?	✓	X	?	X	X
	Indexing Scheme	?	✓	✓	✓	✓	✓	✓
Re-factoring	Support components modification	X	X	X	✓	✓	X	X
	Support extracting components from a system	✓	X	X	X	X	X	?
Test harness	Ensure components conform to requirements prior to delivering to re-users	X	X	X	✓	✓	X	X
	Support filtering the search for components	?	✓	✓	X	X	✓	X
Delivering	Deliver components with all associated dependencies	✓	✓	✓	✓	✓	?	✓
Evolution	Support adding new tools to extend the functionality of the repository system	X	✓	X	X	?	X	X

Table 3.2: Taxonomy of the characteristics of the Repository Systems

The +1Reuse repository system is described as able to extract modules from the available projects in the repository and deliver them together with all their corresponding files to a re-user's system. Considering modules as an analogous to components, then +1Reuse repository seems to support one of the significant characteristics of the ideal repository system which is about extracting components from a system. Moreover, the

repository is able to deliver complete modules to re-users which is another important feature of the ideal repository system. Due to the lack of enough details that describe the +1Reuse repository system it was not possible to identify the organizing scheme, refactoring, testing, and whether the repository supports evolution or not.

The WebComposition repository is another interesting repository system that is built to address some of the re-use obstacles. One advantage of this repository system is that it is able to organize components by different organizing schemes (e.g. classification and indexing schemes). All the characteristics related to the organizing scheme of the ideal repository system are satisfied by this repository apart from the characteristic concerning the ability to define new classifiers by re-using old ones. The WebComposition repository system provides a search engine that can be filtered by terms representing classifiers; as a result, it is considered as supporting the feature of filtering the search for finding potential re-usable components. The repository is also capable of delivering components with all their necessary dependencies to re-users. The repository system is not closed as new meta-data stores can be added to it indicating its ability to evolve.

Sourceforge.net is built based on a combination of defining a set of facets to categorise software components and an indexing mechanism to support free-text searching for projects within the repository. As a result, the repository can be considered to have captured the required characteristics of the ideal classification scheme. However, the classification scheme lacks the ability to re-use classifiers to generate new ones. Re-users are able to filter their searching for re-usable components using some values of the provided facets, for example by selecting Java as a programming language. Also, Sourceforge.net is able to deliver components with all their required files and dependencies, hence satisfies the delivery characteristic of the ideal repository system.

CompSrc employs an indexing scheme for organizing software components in the repository. A desirable feature satisfied by the CompSrc repository is the ability to compose larger components from smaller ones. This feature seems to fall within the kind of support that the repository provides to the map provider’s view to the re-user’s view by generating new components from the ones already available. This feature is considered to be related to the characteristic of modifying components because, it is believed by the author of this thesis, components must be modified, somehow, to successfully construct a composite component. As a result, the CompSrc is considered as supporting mapping, by modification, the provider’s view to the re-user’s view that the ideal repository system

need to satisfy. The repository system provides a number of test-suites that can be utilized to check the I/O data type of components to ensure that two components can be composed together. The developer of the CompSrc repository reported that components are self-contained entities; this denoted that components are delivered with all their associated dependencies to re-users.

The CRECOR repository system implements an indexing scheme for organizing components in similar manner as the other repository systems identified earlier. The repository supports re-factoring EJB components to match the requirements of re-users. This characteristic indicates that the CRECOR repository system supports the characteristic of mapping the provider's view to the re-user's view. The repository system also provided a number of testing facilities to check the conformance of components to the functional requirements of re-users. Components in the CRECOR repository are EJB that are packaged in JAR files, this indicated that components are delivered in full to re-users without missing their dependencies.

The CodeFinder-PEEL repository system indexes software components for re-use. An interesting characteristic supported by the repository system is the ability to assist re-users to formulate their searching queries to best match the terms used to represent components in the repository. This feature is considered as a kind of filtering search query in the sense that the mechanism used is advising re-users about the possible accurate terms that they can use to find more relevant components. Although this mechanism may not sound like filtering in the traditional interpretation of filtering, it is believed that it could relate to filtering as the advising mechanism used by the repository can limit the number of components found by putting more focus on the searching queries than the ones provided by re-users.

Finally, *CodeBroker* established an indexing mechanism to organise software components in the repository. The provider of the *CodeBroker* repository system claimed that the repository is advantageous as it can find components and bring them to developers without the distraction of switching from their working environment to search for software components as such distraction might hinder re-use attempts. While this feature might seem interesting, it was not obvious whether the ideal repository system really benefits from this feature or not. The view adopted by *CodeBroker* concerns software developers who are building their systems from scratch but may be interested in re-use, while the ideal repository system assumes that developers will build their systems by re-using

components, hence developers are named as re-users. The diversion between the two assumptions (i.e. *CodeBroker's* view and the ideal repository system's view) suggested that the feature provided by *CodeBroker* may not be of interest to the ideal repository system. With respect to delivering full components, *CodeBroker* is reported as supporting this characteristic.

3.4.3 Observations

Apparently, numerous attempts by the different repository systems were established to support re-using software components. However, it seems that the current state of the repository systems is still behind achieving the optimal support as compared to the kind of support that could be provided by the ideal repository system; hence more work is still needed to improve the applicability of repository systems to facilitate re-use.

Specifically, it is observed that the surveyed repository systems are primarily concerned with the functional aspects of software components. This is clearly indicated by the kind of organizing schemes used in the various repository systems. All of the identified repositories employ indexing schemes which attempt to capture component semantics to reveal some of their functional characteristics, even though identifying key functionality using the provided indexing schemes in the described repository systems is questionable as capturing functionality requires a formalized approach to reflect precisely the key functional characteristics [48]. Matching only the functional characteristics of software components to the functional requirements of re-users is not enough to find potential re-usable components [132]. It is essential to consider, in addition to the functional characteristics, matching the non-functional characteristics [93] of software components to ensure that the found component really matches all the re-user's requirements to be re-used successfully in the systems being built. Part of the non-functional characteristics that need to be considered is the architectural characteristics in addition to the functional characteristics to organize and find re-usable software components.

Capturing the architectural characteristics of software components cannot be accomplished by the traditional Latent Semantic Analysis (LSA) mechanisms employed by most of the indexing schemes due to the absence of precise characterization of architectural characteristics. For example, the role of components can be active or passive as described by Yacoub *et al* [152]. So, unless both roles are characterized in detail, no indexing scheme can help to find active or passive components from a repository. Although some of the classification schemes identified earlier in this chapter tried to

establish categories related to some architectural characteristics, the classifications are either coarse-grained in the sense categories cannot be reflected on source-code components or only capture few high level architectural characteristics such as programming language name or platform name without going into any more depth. Coarse-grained classification schemes were not practiced in many of the identified repository systems due to the fact that the categories selected to classify components cannot be reflected on real components. Although there are some repository systems that apply coarse-grained classification schemes (e.g. Sourceforge.net and WebComposition) leveraged by some indexing schemes to achieve mapping categories to concrete components, the representative indices of architectural characteristics are still inappropriate as they are purely lexical-dependant. This difficulty in linking high-level architectural categories to concrete software components to assist finding them for re-use seems a potential problem that can negatively affect finding re-usable components. Therefore, there is a need of further research to investigate the possible benefits of considering the key architectural characteristics of software components as a way to assist in finding re-usable components that can fit with a re-user's requirements and address the problem of mapping high level categorization to concrete components.

3.5 Summary

This chapter introduces the background work that forms the basis for establishing this research. The characteristics of the organizing scheme of the ideal repository system have been identified and a survey of a number of organizing schemes has been presented based on the identified characteristics. Also, this chapter presented some of the key related work in the scope of re-factoring components and examined the extent to which the available mechanisms suit the ideal repository needs. The chapter then ends by identifying the key characteristics of the ideal repository system and surveyed a number of repository systems based on the defined characteristics.

The summary of the outcome of the background work is as follows:

- Organizing components based on their functional characteristics is not enough to re-use components successfully as key architectural characteristics must be considered as well.
- Indexing schemes are not appropriate for organizing components in the ideal repository system as they are based on lexical analysis of keywords even though

indexing schemes satisfy some of the characteristics required by the ideal repository system.

- Current classification schemes use coarse-grained categories that cannot be mapped to concrete components, and even the schemes that use less coarse-grained categories are not appropriate as they are based on lexical analysis of keywords and not on precise characterization of categories.
- Re-factorization is not concerned with mapping the views of component provider and re-user.

Overall, current repository systems still cannot provide the optimal support for re-use. Building on the background work developed in this chapter, the next chapter defines the approach followed by this research to satisfy the characteristics of the ideal repository system.

Chapter 4 - Characterizing Architectural Fit

The previous chapter established the basis for this research by identifying the gap in prior work corresponding to the problem of finding re-usable components in a software repository system. That prior work was concerned primarily with identifying and characterizing the functionality of software components to facilitate finding re-usable components; however, the architectural characteristics of software components were not adequately defined and exploited. As a result, a re-user might find a component that provided the required functionality, but could not be re-used in the system being built due to mismatches between the architectural characteristics of the component and that system.

This chapter describes the notion of *architectural fit* as a new approach that could be utilized by the ideal repository system to facilitate finding re-usable components that match the non-functional characteristics of the system being built. The chapter starts by defining a number of use-cases for the ideal repository system in order to gather the requirements for developing the notion of architectural fit and an approach named *architectural interface* that is a solution to be described in this chapter. After that, the chapter introduces a system model that establishes the basis for generating the vocabulary used in this research. Subsequently, the chapter discusses the characteristics of *architectural interface* that is established in this research to address the gathered requirements of the ideal repository system. The chapter then describes aspects of checking for *architectural fit* and how that can address some of the functional requirements of the ideal repository system.

4.1 Use-cases for the Ideal Repository System

As a way of gathering the requirements of the ideal repository system, a number of use-cases were identified. Figure 4.1 illustrates a coarse-grained view of the ideal repository system depicting its users and part of its architecture. The use-cases are identified from the perspective of the re-user and the depositor who are the two main users of the repository system.

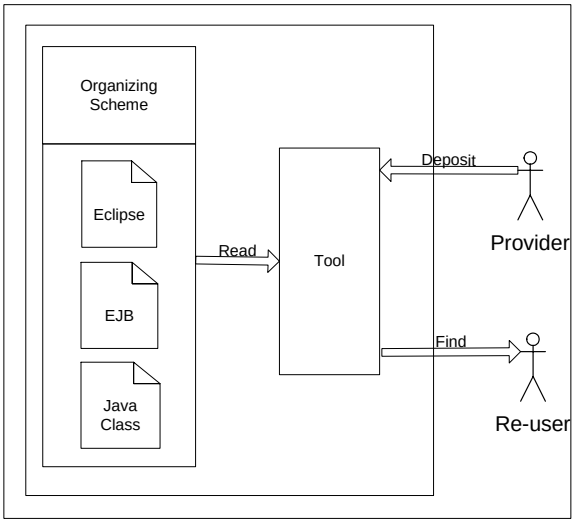


Figure 4.1:Coarse-grained View of the Ideal Repository System

The first two use-cases to be developed correspond to the re-user of the ideal repository system. The third use-case corresponds to the provider of components to the ideal repository system.

Use-case 1: Finding Re-usable Components

A re-user who is building a system might decide to re-use a component that provides new functionality instead of building the component from scratch. The re-user can go to the ideal repository system and search for components. The re-user submits to the repository system a query that contains the definition of the characteristics of the required component. After the query has been submitted, it is then received by the checking tool within the ideal repository system (shown in the above figure) that processes the submitted query and matches the defined characteristics to component meta-data held in documents in the organizing scheme in the repository system or against the source code of the available components in the repository. If a matching component is found then the component is delivered with all its related dependencies to the re-user.

Three main requirements can be drawn from the above scenario:

- Re-users must precisely define the characteristics of the components to be searched for in the repository to facilitate their accurate identification.
- The submitted query must be matched against the meta-data corresponding to components in the repository system. If no match is found, the query must be matched against the available components in the repository.

- Any component matching the defined characteristics must be delivered in full to the re-user. Compile-time or link-time errors must not arise in the system being built due to there being parts missing from the component that was delivered.

Use-case 2: Modifying Components to Suit the Requirements of the Re-user

A re-user submits a query to the repository that defines the requirements of a component required by the system to be built. However, the repository system only finds a component that matches only some of the requirements. There are two reasons why a component does not provide a complete match. One possibility is that the found component provides the required functionality, but conforms to different architectural requirements to those required. The second possibility is that the found component matches the required architectural characteristics, but that the provided functionality is not quite what the re-user needs. The re-user may feel that the effort of modifying the found component is less than that of building a component from scratch, so the re-user decides to modify the component to make it match the full requirements of their system, hence may need the repository to assist the modification.

Three requirements can be drawn from the above scenario:

- The architectural requirements of a software component must be identified precisely.
- The modified component must be checked against the meta-data in the repository.
- The boundaries of the sub-components of a found component must be identified in order to modify the component's functionality.

Use-case 3: Depositing Software Components

A component's provider deposits a component into the ideal repository system. In addition to the component, the provider could supply the "meta-data" that defines the functional and architectural characteristics of that component and any sub-components of that component. In this case, the repository system assists the provider to define the required characteristics of the component before depositing it into the repository. Alternatively, the provider could deposit the component without providing any meta-data. In both cases, after the component is deposited, it is checked and automatically organized in the repository based on the meta-data that the deposited component matches.

Four requirements can be drawn from this scenario:

- The repository must present the characteristics that the component's provider needs in order to define their corresponding values.
- Minimize the effort of defining the values of a component's characteristics.
- Check the deposited component to ensure that it matches the provided definition.
- If the provider did not supply a definition of a component's characteristics, check the deposited component against the available meta-data held in the organizing scheme of the repository system in order to identify and organize that component according to its matched meta-data.

The defined use-cases identify the requirements for developing part of the ideal repository system. Addressing the identified requirements is the subject of the remainder of this chapter and the next. Thus, a starting point for discussing the structure [15] of software systems in a general manner, which can establish a vocabulary for investigating the issues, is to consider defining a system model.

4.2 System Model

A common model for understanding system structure is to consider a system as being composed from a set of components. A component might be atomic [26] in a scene that it can be composed of sub-components, and sub-components might also be composed of sub-sub-components, and so on until a point is reached where a component cannot be decomposed any further (e.g. a binary code component). Each component itself can be considered as a system, with the above description being applied recursively.

Following the perspective of this system model, a software developer can be regarded as building a system. That system might be a complete system (e.g. a stand-alone application) or be a part of a larger system (i.e. a component), but the model permits the general term *system* to be used to cover such eventualities. Since the developer is building a system, *components* are what they may try to find and re-use, and are the dependencies that the system utilizes for providing the necessary functionality.

While this is a simple system model, and clearly does not capture all of the complexity of a software system's structure, it is sufficient to use as the model for identifying the important characteristics necessary for component re-use.

A *re-user* is the person who is trying to re-use components when building a system. Every system has some characteristics that a re-user will need to consider when searching

for re-usable components. Similarly, when a potential component is found, it will have characteristics that the re-user will need to examine to identify whether that component can be re-used in the system under development. If the characteristics required by the system are matched by the characteristics exhibited by the component, that component will be a candidate for re-use in that system.

Following from the system model, the notions of system and component are interchangeable in the sense that a system can be considered as a component if a re-used decided re-use it in another system, while a component can be considered as a system in its own right, for example, if a re-user is interested in examining its composing sub-components. So, all the characteristics (to be defined) that are relevant to a system are applicable to a component and vice-versa. Figure 4.2 illustrates the ontology of the described system model.

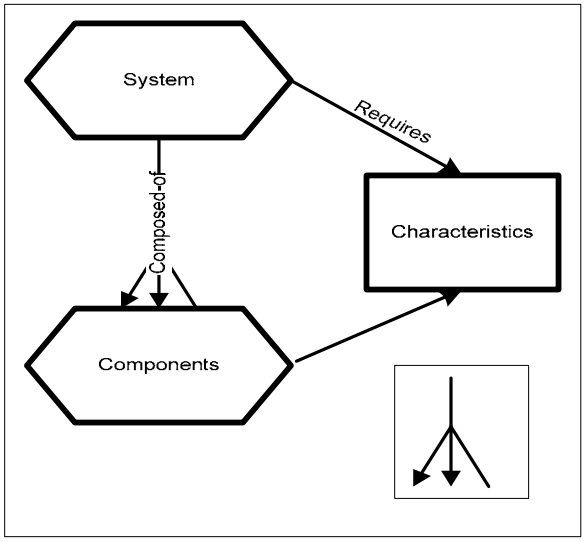


Figure 4.2 Ontology of the System Model

Two types of system characteristics (i.e. exhibited and required) are captured by different interfaces of that system. One interface is intended to identify the characteristics that a component must exhibit in order to be re-used in the system, while the other interface identifies the components of the system and the relationships between them. Both interfaces are crucial to the successful re-use of a component in a system. The next section discusses the two types of interface in depth.

4.3 Types of Interfaces

It is useful to distinguish between two types of interfaces of a system, namely *external* and *internal* interfaces. The external interface of a system captures the characteristics that

must be exhibited by the system, and can be used to identify whether a system is re-usable or not. The internal interface of a system is significant in identifying the characteristics of the composing components, that are the dependencies of the system, and also the characteristics defining how components can interact with each other. An analogy with jigsaw pieces is useful to express the idea of the two interfaces. The things needed are the "hole" in a jigsaw piece, and things provided are the "protruding bobble". So, an external interface of a piece of a jigsaw has holes that it needs, and bobbles that represent what it provides; similarly for the internal interface.

Both interfaces identify characteristics that dictate whether a system can be successfully re-used in another system. For example, if a system requires its components to provide a method called `public void start()` to control when the component starts running, this requirement forms part of that system's internal interface, and the method must be part of the external interface of any potentially re-usable component. Similarly, if a system uses some libraries to implement its functionality, the characteristics of the library must be part of the internal interface of the system. Figure 4.3 depicts the two types of interfaces for a system.

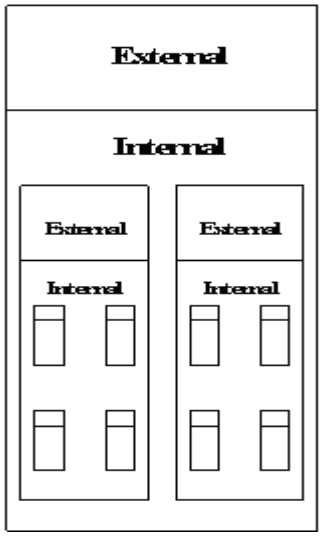


Figure 4.3: The Types of Interfaces

It is worth mentioning that a component's dependencies are in two forms. One form of dependency, namely *external dependencies*, will be satisfied by components that are provided by the system in which a component is re-used. For example, in a Java system, a component (e.g. a Java class) may need to use the "java.io" library which is one of the standard libraries used in most Java-based systems. The other form of dependency, namely *internal dependencies*, will be specific to a component and not related to what the system might provide. Internal dependencies relate to the sub-components of the component. The

external dependencies are captured by the external interface of a component as they form a part of the requirement that must be provided to a component to enable that component to function. The internal dependencies are captured by the internal interface of the component and are represented by the sub-components that must also be provided together with the component.

Consider the Eclipse IDE [32] as an example of a system that a developer wants to add some functionality to by incorporating new "plug-ins". The Eclipse IDE provides an extensible environment that precisely defines how new plug-ins can be added to the system, and also establishes the basis for defining the relationships between plug-ins. Mapping the Eclipse system to the system model introduced in this section, the internal interface of the Eclipse system requires the following methods as part of the characteristics that the external interface of a component (i.e. a plug-in) must match in order to be re-used in the Eclipse system:

- `public void start(BundleContext)`
- `public void stop(BundleContext).`

A plug-in might have interaction with other plug-ins in the Eclipse system or it may need sub-components to accomplish its desired functionality. For example, a file-transfer protocol (FTP) plug-in needs to interact with the "org.eclipse.osgi" plug-in, which is part of the Eclipse system, to facilitate launching the FTP plug-in in the system. So, the "org.eclipse.osgi" plug-in must be defined as a part of the characteristics that the external interface of the FTP plug-in must capture, as it is one of the external dependencies of the FTP plug-in that is required by the Eclipse system. The FTP plug-in uses a Java class called "newSocket" that is not part of what the Eclipse system requires, hence the "newSocket" Java class must be captured by the internal interface of the FTP plug-in as one of its internal dependencies.

Several benefits can be obtained from identifying the two types of interfaces (i.e. external and internal):

- From the perspective of supporting re-use with a repository system, the external interface of software components can be used by the repository system to automatically classify and organise those components, while a set of characteristics that a re-user requires can be specified and used by the repository to identify candidate components. Moreover, the internal interface is useful to help the repository system retrieve a re-usable component together with all of its required

dependencies (i.e. sub-components). Thus, the repository can provide a complete component to a re-user, without requiring the re-user undertake this action manually.

- From the perspective of organizing components inside a repository system, the external and internal interfaces can be used to build an organisational hierarchy; Figure 4.4 illustrates an example. Assume that T is a component that defines the characteristic Y in its internal interface. Sub-components T1 and T2 are both identified as providing the characteristic Y in their external interface. However, sub-component T1 defines the characteristic A in its internal interface while sub-component T2 defines B as a characteristic in its internal interface. As a result of the difference in the characteristics defined by T1 and T2's internal interfaces, the two sub-components can be discriminated from each other. The example indicates that the external interface of a sub-component identifies the potential parent in a hierarchy and the internal interface discriminates a component (or sub-components) from other components.

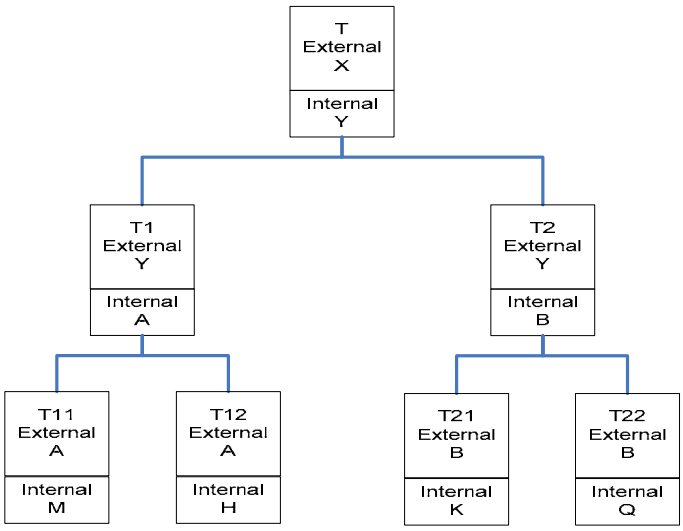


Figure 4.4: Using External/Internal Interfaces to Organize Components

- From the re-user's perspective, understanding the external interface of a component and internal interface of the system under development might influence the modifications that the re-user might wish to make. A re-user could modify the internal interface of the system under development to match the external interface of a potentially re-usable component, or the re-user could modify the external interface of a component to match the internal interface of the system.

The above discussion has identified the potential importance of the external interface of a component in addressing the problem of finding re-usable components. It is therefore necessary to examine in more depth the characteristics that external interfaces should capture, and this is the subject of the next section.

4.4 The Characteristics Defined by the External Interface of Software Components

The external interface of a component involves functional and architectural characteristics. The functional characteristics define the behaviour that a component can provide. Architectural characteristics define the requirements that allow components to fit physically into a system. The term *architectural type* is therefore going to be adopted to represent the architectural characteristics.

If the architectural type that is defined by a component matches the architectural type that is required by a system then the component is considered to be an *architectural fit* for the system. The term architectural fit has been introduced to represent the ability to incorporate components physically into a system, meaning that the component will not cause raise compile-time or run-time errors after integration into a system and satisfy the *architectural fit requirement*. The term architectural fit requirement indicates that the architectural type of a component is the same as the architectural type required by a system. Similarly, if a component matches the functional characteristics required by a system then the component is said to be a *functional fit* for the system, hence satisfy the functional fit requirement. A component that satisfies both the functional fit and the architectural fit requirements of a system is termed a *perfect fit*. The sub-set of characteristics in the external interface of a component that relate to the functional fit can be said to identify the component's *functional interface*, while those that represent the architectural type can be said to identify the *architectural interface*. The two interfaces comprise the complete external interface of a component.

When a re-user is looking for a component, satisfying functional fit is of course a major concern. As indicated in Chapter 3, prior work has tried to establish mechanisms to facilitate finding components that satisfy a re-user's functional fit requirements. However, this thesis argues that satisfying the requirements of functional fit is not sufficient to successfully re-use a component in a system, since it is vital that the architectural fit requirements are also satisfied. For example, a re-user may find a component that has the

exact functionality required, but that will not fit into their Windows environment because the component was designed for a UNIX environment.

Due to the significant impact of architectural fit on a component's re-usability, this thesis addresses the architectural fit requirements identified by the architectural interface of a software component as a step towards achieving better support for re-use.

4.5 Setting the Context of Architectural Interface in the Scope of the Ideal Repository System

In a repository system, software components can either be available by themselves, a standalone application, or as part of a system. In both cases, components must conform to an architectural type that relates to the system that the components were primarily built for. The architectural type that a component exhibits can be exploited to discriminate one component from other components that conform to different architectural types. The identification of the architectural type of a component is of particular interest when a component is deposited into a repository as the identification forms the basis for organizing components in the repository, and hence facilitates their subsequent identification by re-users.

Figure 4.5 illustrates a fine-grained view of part of the design of the ideal repository system that relates to identifying and organizing components based on their architectural type.

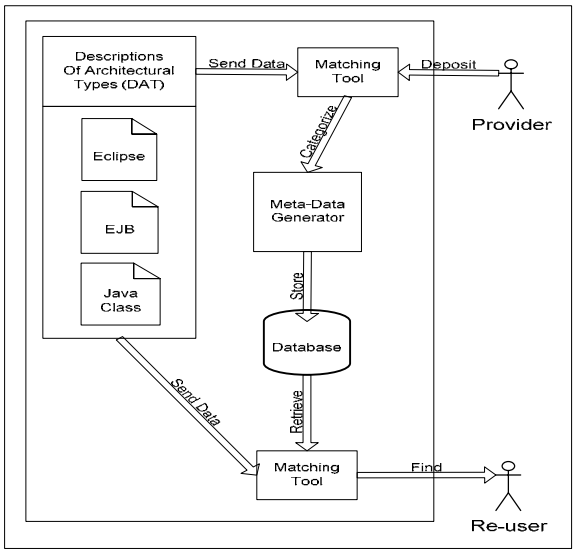


Figure 4.5:Fine-grained View of the Ideal Repository System

The figure shows four components of the ideal repository system:

- Descriptions of Architectural Types (DAT): contains documents describing the characteristics of architectural types and defining the relationships between them.
- Matching Tool: when a component is deposited in the repository, the matching tool can check whether the component conforms to any of the architectural types recorded in the DAT. The matching tool will also be used when a re-user needs to specify their architectural interface requirements for components they are searching for.
- Meta-Data Generator: this tool generates meta-data to be associated with the component, recording any "is-a" relationships identified by the matching tool between the component and the architectural types recorded in the DAT.
- Database: stores components and their meta-data for future retrieval.

The sequence of operations performed by the repository to identify and organize components is as follows:

1. A component is deposited in the repository.
2. The matching tool checks the contents of the component against the characteristics defined in the architectural type descriptions in the DAT.
3. If the component is identified as matching one or more of the architectural types, the component and the matching results are passed to the meta-data generator tool. If a

component is composed of sub-components, the matching tool also matches the sub-components against the available architectural types.

4. The meta-data generator annotates the checked component (and sub-components) as instances of the matched architectural type descriptions, for subsequent storage in the database.

Every architectural type defined in the DAT is represented by a document, and the precise characteristics defined in a document are those associated with architectural interfaces (to be described in section 3.6). Hence, every architectural type within the DAT is an instance of an architectural interface. Accordingly, a deposited component is matched automatically against the instances of architectural interface in the DAT in order to identify its matching architectural type. If a component matches an architectural type, the component is considered as an instance of that architectural type and organized as such.

The definitions of architectural types in the DAT can also be utilized by the repository system to identify and extract the sub-components of a deposited component into the repository system. For example, assume that an Eclipse component (i.e. plug-in) is deposited into the repository system, and it is composed of three sub-components that conform to Model, View, and Controller architectural types already present in the DAT. The DAT can then be used to identify those composing sub-components by matching every sub-component to its corresponding architectural type in DAT. The ability to identify the sub-components of a deposited software application is useful to populate the repository with components, and is especially important to re-users who might be interested in re-using some of the components that are embedded in a deposited application.

This section has set the context of the notion of architectural interface and how it can be used in the repository in which it is located. The detailed characteristics identified by architectural interfaces are discussed in the next section.

4.6 The Characteristics Identified by the Architectural Interface

Figure 4.6 describes the ontology that defines the vocabulary of a fine-grained view of the system model defined in section 4.2.

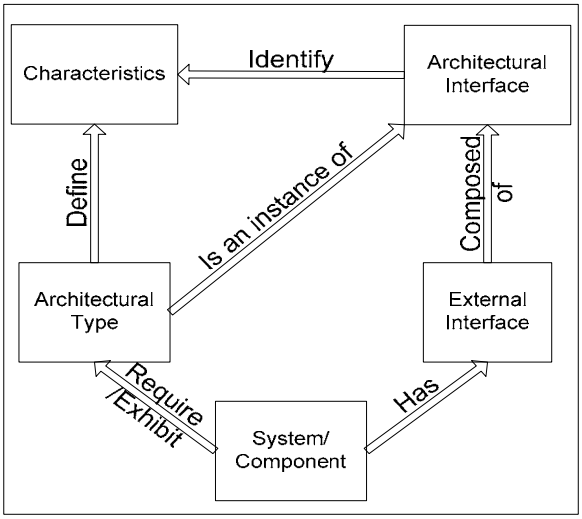


Figure 4.6:Fine-grained Ontology of the System Model

The above figure identifies that a system has an external interface that is, in part, composed of an architectural interface. The architectural interface of a system identifies the characteristics that are going to be defined by the architectural type that is required by the system.

Based on the practical experience and the background work conducted by the author of this thesis, the characteristics identified by architectural interface are as follows:

- Format: this characteristic specifies the language used to write a component. For example, at the source-code level, a programming language will represent the format of a component. So if a system requires a component written in Java then a component written in FORTRAN will not be directly suitable for re-use.

- The way components interact in the system: the interactions between components involve the method of exchanging data and control. Data can be exchanged between components in different ways. For example, one component may exchange data by passing parameters while another component might exchange data through shared memory. Also, data might be exchanged among components following different exchange models. For example, a component might employ the push-model [42] indicating that the data is sent out by the component whenever a change in the component's state occurs. The other model that a component could implement to exchange data is the pull-model [42] where data is requested from the component by other components in a system whenever a change in the component's state is detected by the other components in the system. The way control is exchanged can also be different from one component to another. One component may synchronize

its execution with a system, so the component can return control to the system upon the completion of its execution. Another component might execute asynchronously with the system. Thus, identifying the different ways of exchanging data and control is necessary for finding re-usable components in a repository system.

- The way components can be initialized: some components may provide special methods that must be executed to provide initialization, while others may require the presence of special tools or files for their initialization. For example, a stand alone Java application must have a method called `public static void main()` to be initialized, while an Eclipse plug-in can be initialized by reading a file called "plugin.xml" and the presence of a method called `public void start(BundleContext)`. So, a component must match the initialization mechanism that a system requires in order to be re-used successfully in the system, and considering this characteristic is necessary for finding re-usable components.

- The way components handle failures: if a fault [91] occurred in a component at any stage during its execution then the failure handling mechanism implemented by the component must conform to the one expected by the system. For example, if a system assumes that its composing components must provide a specific recovery action in case of failure (e.g. reset their states), this must be a characteristic of a component to be re-used in the system.

- The non-functional characteristics of software components: characteristics such as performance, size, memory usage, and reliability are important to be identified as far as re-use is concerned. For example, in a real-time system a component's performance may be an important issue to consider. If a system requires components that must respond in 0.05 seconds then components that respond in two seconds are not suitable for re-use.

- Using external dependencies: a software system may require its composing components to use dependencies that it provides for them to fit in the system. For instance, referring to an earlier example, a Java system requires its composing components (i.e. Java classes) to use a library called "java.io" to achieve the basic input and output functionality. Also, an Eclipse system requires its components (i.e. plug-ins) to use a plug-in called "org.eclipse.osgi" to allow the system to control their execution. So, components must use the external dependencies that are

provided by a system in order to be re-used successfully in the system; hence this characteristic should be considered to find an appropriate re-usable component.

- A component's boundary: components in a system must have some boundaries that identify which parts of a system source code form that component. For example, in Java, classes form the boundary of the source code of a component that separates it from other components in a system. Although this characteristic may not be relevant to achieving successful re-use, it is a requirement that must be defined in order to extract a component from a system.
- Internal dependencies: components in a system may depend on sub-components. Hence, extracting one component of a system to re-use in another system requires also extracting all the internal dependencies of that component.
- The sequence in which components need to be invoked: a software system must invoke components in the correct sequence otherwise some of the composing components of the system may not execute correctly. For example, consider a simple *parser* system composed from a *reader* component that reads from a file and stores data in a temporary buffer for processing, and an *analyzer* component that analyzes the data and identifies their semantics. The *parser* system must invoke the *reader* component first and then invoke the *analyzer* component. If the *analyzer* component is invoked prior to invoking the *reader* component then this might cause the *analyzer* component to raise an error, and hence cause the system's execution to fail.
- Support pre-emption: pre-emption is the ability of a system to interrupt the execution of its composing software components in order to switch the thread of control from one component to another one [51]. Re-users must configure the components that they want to re-use to support such a characteristic if the system they are developing requires that.
- Context: the context defines the way components can be registered in a system to subscribe to events that may be raised by the system and also to obtain information from the system. For example, the Applet architectural type defines the method `public AppletContext getAppletContext()` that provides a handler for the applet in a system.

- Persistency: a system might require its components to store their state to an external storage (e.g. file, database) and retrieve their state when necessary. For example, Eclipse plug-in architectural type defines a method `public final IPath getStateLocation() throws IllegalStateException` to handle persistency issues.

Identifying the possible values of the characteristics is necessary to determine whether a component can fit architecturally into a system or not. The values of the identified characteristics are defined by an architectural type. Hence, there is a need to specify these characteristics and their corresponding values in a precise manner that could be identified in the source code of a given component. A prototype of a specification language namely *ArchInt* (to be described in the next chapter) is developed in this research to formalize the characteristics identified by architectural interface.

The specification language has to describe things with respect to syntactic constructions that can be identified in the source code of the component, and semantics concerning what the construct means. For example, the syntactic aspects of identifying what a method is (e.g. a Java Method), must be separated from the semantics of that method (e.g. it corresponds to handling a failure). What is needed is a simple mechanism that returns a "Yes/No" answer with respect to matching a component's source code against the characteristics defined by an architectural type – in other words, performing a syntactic match between a component and an architectural type. Ignoring the semantics of the characteristics defined by an architectural type has the advantage of facilitating a tool to check automatically the availability of the characteristics in the architectural interface of software components without human intervention. For instance, if an architectural type that is required by a system defines one of its characteristics as requiring a UNIX process with standard inputs and outputs then the architectural type of a component must define this requirement in order to match the architectural type required by the system. The semantics of that UNIX process is not important to fit into the system. Therefore, the practical studys conducted in this research are concerned only with syntactic matching of the characteristics defined by an architectural type against the source code of components as a first step to investigate the feasibility of the approach of architectural interface to support component re-use.

The next section introduces some aspects of checking, illustrated by Java programming language mechanisms, that could be utilized by the ideal repository system to check for architectural fit in a language neutral manner.

4.7 Aspects of Checking for Architectural Fit

Recall the earlier discussion in Section 4.4 that described the main concern of architectural fit as facilitating the successful incorporation and integration of components mechanically into a system. This section discusses how architectural fit can be checked in the context of programming languages and draws analogies to the kind of check necessary for the repository system.

Starting with a simple example to express the idea of how architectural fit is checked within the confines of programming languages, the mechanisms in Java can be used to exemplify this idea. The Java compiler performs one form of check at compile time (syntax checking) and another at link time (relationships checking based on methods availability in a class) before generating the executable version of a system. At compile time, the compiler checks whether the source code is syntactically correct. This kind of check is analogous to checking whether the format of a component matches the format defined by an architectural type. The Java compiler can also check the links between components in a system using the "linker" which is a sub-tool of the Java compiler. If a component is missing one or more of its dependencies then the "linker" can identify those omissions and notify the developer. These link-time checks are analogous to checking for a component's dependencies in a repository system prior to delivering the component to a re-user. So, an analogy of the functionality provided by the Java compiler can be utilized in a repository system to identify all the necessary external and internal dependencies of a component, and the internal dependencies can be delivered together with the component to a re-user while the external dependencies are noted for the re-user.

In Java, there is also the notion of abstract classes and interfaces that can be used by the Java compiler to check the availability of methods in a component. For example, if the developer has decided that all the components of the system to be built in Java must implement a method called "`public void run()`" in order to start their execution, the method could be defined in a Java interface (or abstract class) that every component must then implement. This kind of check means checking for the decisions imposed on components by the architecture of the system to be built (the component's architecture for short). So, if one of the components of the system missed the required method by mistake,

the compiler can identify the missing method and notify the developer. The notion of Java's abstract classes and interface mechanisms could be utilized by a repository to check a component's conformance to an architectural type. However, not all programming languages have these features, and hence a solution not linked to a specific programming language is required.

A more general example that is not dependent upon programming language mechanisms to express the idea of architectural fit of a component is the Eclipse system introduced earlier. Currently, there is no tool support to check whether a component will fit architecturally into the Eclipse system or not. An Eclipse plug-in is composed of a compiled Java source code, but the link checks cannot be performed as the plug-in has to be dynamically linked to an already executing Eclipse system. The only way to verify that a component can fit into an Eclipse system is to see whether the system executes successfully with the component; however, unsuccessful execution, indicating an unsuccessful fit, is something that the re-user might have preferred to find out about at an earlier stage, along with the reasons why the component would not fit. Therefore, it is necessary to establish a checking mechanism for software components before incorporating them into a system, to ensure that the component's architectural type characteristics conform to those required by the system. The next section describes the general mechanisms for checking component conformance to the characteristics of an architectural type.

4.8 Checking Architectural Types in the Context of the Ideal Repository System

Recall the design of the ideal repository system illustrated earlier in Figure 3.1 in Chapter 3. There are two occasions where the checking tool described in the design of the ideal repository system can perform the necessary checks to support re-use effectively:

- Checking performed when a component is deposited in the repository; and
- Checking performed when a re-user is searching for a component.

The checks performed at deposit time are necessary for identifying and categorizing the component in the repository. The repository system contains a number of architectural types definitions, held in the DAT component of the repository as illustrated in Figure 4.5. Every deposited component can be matched against the available architectural type definitions in the repository system to identify its matching architectural type. If a match is

identified, the component is categorised appropriately. Matching the characteristics of a component against the characteristics defined by the architectural types in the repository system will be referred to as checking for the “is-a” relationship, that is to check whether the architectural type of the provided component is found to be equivalent to an architectural type defined within the repository. If a match is found between an architectural type in the repository and a component, the following statement will become valid: “*the architectural type of component X is-a Z architectural type*” where Z is the architectural type for which the match was found. The component is considered to be an instance of the Z architectural type and categorized as such.

The checking at search time involves matching the characteristics of the architectural type that a re-user has provided against the characteristics of the available architectural types held in the repository system. If a matching architectural type is found then the components in the repository that are categorized under the defined architectural type will be the set of interest to the re-user. If no matching architectural type is found, the characteristics provided by the re-user can be checked against the source code of the available components in the repository in order to identify any matching components, in a similar manner to the checks performed when a component is deposited in the repository (i.e. checking for the “is-a” relationship).

It is vital for a repository to be able to identify and organize components automatically without requiring human intervention. The repository also should facilitate finding the most appropriate components, otherwise the repository is not going to be very practical to support software development. Hence, checking for the “is-a” relationship can be utilized by a repository system to perform the automatic identification and organization of the components being deposited in the repository without requiring a component’s provider to be involved. Checking for the architectural fit can be utilized by the repository to provide a selected sub-set of all of the components in a repository which more closely match the requirements of re-users. This check can therefore add additional value to the current searching mechanisms (e.g. free text searching) discussed in Chapter 3 in order to facilitate finding components.

The two checking mechanisms (i.e. “is-a” relationship and architectural fit) have to be built on a precise characterization of the architectural types of software components. The improvement in the support provided by a repository system that can be achieved by employing the two checking mechanisms indicates that the notion of defining architectural

fit could be really useful to enhance component re-usability, and consequently improve software development with re-use. So, both checking mechanisms are appropriate as parts of the design of the ideal repository system.

4.9 Summary

This chapter has established the basis for developing an approach that can satisfy some of the key requirements of an ideal repository system. The chapter started by identifying the requirements of two users of the repository system (i.e. re-user and depositor) by describing a number of use-cases for the repository. A simple system model was introduced to provide a useful framework for viewing software systems, their components, and how they relate to each other. After the system model was described, the chapter proceeded by identifying two component interfaces, namely the external and internal interfaces. The chapter then described the different characteristics of an architectural interface and how those characteristics can affect component re-usability. The characteristics of architectural fit and how components can be considered as potential fit candidates in a system was discussed. Finally, the chapter described how architectural types can be utilized by an ideal repository system to perform the checks necessary to categorize and find components.

The important outcomes from this chapter are:

- Re-usable components are those that not only provide the required functionality but also match the architectural type of the system being built, and therefore are perfect fit candidates.
- Component interfaces (i.e. external and internal) can be used for categorizing components and to build an organizing scheme for a repository system.
- The architectural types of software components can be utilized to identify and categorize software components automatically in a repository system.
- The architectural type is useful as a complementary approach to the available searching mechanisms introduced in prior work, and can assist re-users in refining their search criteria to find components that fit architecturally into their systems.

The following chapter describes an approach, namely *ArchInt*, that formalizes the idea of architectural type introduced in this chapter so that the identified capability of architectural interfaces can be examined. The chapter also investigates a concrete prototype of a language to specify architectural interfaces so that experiments can be undertaken to

determine whether the potential advantages of the whole approach described in this research are borne out in practice.

Chapter 5 - The Formalization of Architectural Interface

The previous chapter described the notion of architectural interface and identified several characteristics related to supporting component re-use. The chapter also pointed out that matching the semantics of the various characteristics is not relevant to this stage of the development of the architectural interface; only the syntactical matching is of interest in order to examine the feasibility of architectural interface for addressing the problems related component re-use.

This chapter formalizes some of the concepts of architectural interface described in Chapter 4 by discussing how those concepts can be represented in practice through the development of a prototype specification language for architectural interface to be called *ArchInt*. The chapter also describes experimental work conducted with ArchInt for evaluating architectural interface, and discusses the results of the studies. Chapter 6 will evaluate the overall concept of architectural interface and its support for component re-use.

5.1 ArchInt Specification

It was mentioned in Chapter 4 that a prototype of a specification language is needed for describing architectural interface in a manner that can be constructed by the some human (e.g. an engineer) and subsequently can be machine processable. ArchInt is developed as a prototype of the required specification language to evaluate some aspects of architectural interface. ArchInt represents a document that contains the set of values that comprise a particular architectural type and is used to match characteristics represented in the source code of a software component against the architectural type. An ArchInt document has a specific structure that needs to be processed by machine. Hence, XML seemed a sensible choice for representing the information to help in detecting errors in the document itself. In the context of the ideal repository system, ArchInt documents represent the architectural types defined in the DAT component (discussed earlier in Chapter 4) of the repository system as illustrated in Figure 5.1.

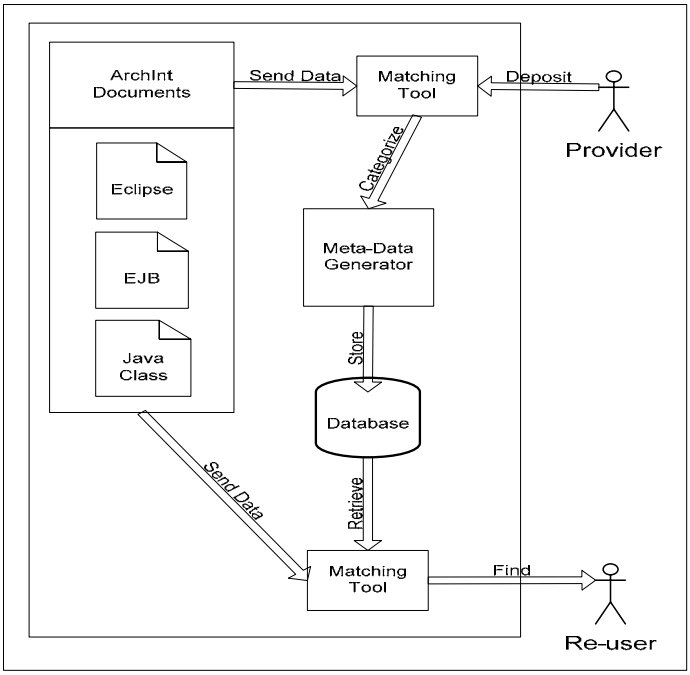


Figure 5.1: Fine-grained View of the Ideal Repository System

Every document must start with a pair of opening and closing tags called `<ArchInt>` to identify the boundaries of a document written in ArchInt and also to indicate that the defined XML document is an ArchInt document. The opening and closing tags must be the first and last tags in any ArchInt document. In a repository system, every architectural type must have a name to distinguish it from other architectural types in the repository, this is captured by ArchInt using a pair of tags called `<name>`. Every ArchInt document must contain only one name and the tag corresponding to the name must be the first tag that appears after the `<ArchInt>` tag. One characteristic that is identified by architectural interface in Chapter 4 was the "Format"; this characteristic specifies the programming language that is used to write a component. ArchInt captures the "Format" characteristic using a pair of tags called `<programming_language>`. This tag is necessary to identify how software components can be processed. A repository system holds a number of tools that can be used to process software components to check their architectural characteristics. The `<programming_language>` tag identifies the appropriate tool to be used by the repository to check the conformance of software components to an architectural type. As will be seen later in this chapter, the compiler associated with a programming language is used as a tool to examine the characteristics of software components.

The three tags described earlier represent the basic features of the ArchInt language and must be present in every ArchInt document to identify the type of the document (i.e.

conforming to ArchInt specification), to identify the name of an architectural type, and to identify the tool that can process a component from the tools available in a repository. The structure of the basic features of ArchInt should be similar to the one used to represent the Java class architectural type as depicted in Figure 5.2.

```
<ArchInt>
  <name>Java Class</name>
  <programming_language>Java</programming_language>
</ArchInt>
```

Figure 5.2: Java Class Architectural Type Represented in ArchInt

In the above example, the tag `<programming_language>` specifies that the necessary tool to check software components is related to the Java language system, hence the Java compiler can be used to perform the necessary checking for the specified programming language.

The characteristics that need to be matched in the source code of software components against an architectural type description are captured by ArchInt using the pair of tags called `<must_have>`. This pair of tags indicates that the content between them is related to the requirement of architectural fit. So, if an architectural type that is required by a system defines a method called "`public static void main(String arg)`", then this method should be described between the `<must_have>` tags pair.

ArchInt captures part of the requirements of architectural fit by the pair of complex tags (i.e. composed of sub-tags) called `<Method>`. The fundamental idea that is captured by the pair of `<Method>` tags is related to identifying the address within a component where data is exchanged, and also the type and sequence of data input and output of the component in that address. The name of this tag is not meant to refer to any particular programming language but might be used to indicate a block or part of a source code that might exchange data. For example, in Java the `<Method>` tag corresponds to the *methods* defined in a Java class, while in Eiffel the tag corresponds to the *features* defined by an Eiffel class, and in FORTRAN the tag corresponds to *sub-routines*. Within the body of the tag `<Method>` the name of a method is captured using the pair of tags `<name>`. The data that can be received by a method is captured as parameters and represented in ArchInt by the pair of tags `<param>`. The data type of the input parameters of a method is captured by ArchInt using the pair of tags `<string>`. A method may have more than one parameter that would be represented by a sequence of `<string>` tags that is important to reflect the sequence of the data input to a method. The data type of the output of a method

is captured by ArchInt using the tag <returnType>. If there is more than one output, the sequence of <returnType> tags written in an ArchInt document reflects the sequence of data output from a method. An exception that might be raised by a method is represented in ArchInt using the tag <exception>.

Software components might include some descriptive files that might satisfy special requirements of a system in addition to the source code of the component. As a result, ArchInt identifies a pair of complex tags called <File> to capture the additional files that might be defined by an architectural type. This tag will be part of the characteristics that should be defined between the <must_have> pair of tags in an ArchInt document. Within the body of the <File> tag the name of a file is captured using the pair of tags <name> to identify a file from any other files that might also be defined by an architectural type. Every file must have a type that represents its format (e.g. XML, Doc, TXT). The type of a file defined by an architectural type is captured by ArchInt using a pair of tags called <type>. This tag identifies what tool can be used from those available in a repository system to check whether a file is well-formed or not. Figure 5.3 illustrates an extract of the Eclipse plug-in architectural type to exemplify the usage of the two complex tags <Method> and <File>.

```
<ArchInt>
  <name>Eclipse</name>
       .
       .
       .
    <must_have>
      <Method>
        <name>start</name>
        <param>
          <string>BundleContext</string>
        </param>
        <returnType>void</returnType>
        <exception>Exception</exception>
      </Method>
      <File>
       <name>plugin</name>
       <type>XML</type>
      </File>
       .
       .
       .

    </must_have>
</ArchInt>
```

Figure 5.3: An Extract of the Eclipse Plug-in Architectural Type

A system might require its composing components to hold temporary data during their lifetime in the system or to define values for some specific attributes of components

required by the system. ArchInt captures this requirement of a system using a pair of complex tags called <Field>. In the source code of components, fields and member variables are the concern of this tag. Every field must have a name that distinguishes it from other fields in a component, hence a pair of sub-tags called <name> that represent the exact name of a field that a system is expecting its component to have. The data held by a field must be of a certain type, hence a pair of sub-tags called <dataType> is defined by ArchInt. Figure 5.4 illustrates an extract of the ArchInt document that describes the "Eclipse XML" architectural type (to be described later).

```
<ArchInt>
  <name>Eclipse XML</name>
      .
      .
  <must_have>
       <Field>
              <name>id</name>
              <dataType>String</dataType>
       </Field>
            .
            .
  </must_have>
</ArchInt>
```

Figure 5.4: An Extract of the Eclipse XML Architectural Type

ArchInt can reduce the effort of writing new ArchInt document of an architectural type that, part of its defined characteristics, is captured by another ArchInt document in a repository. So, old ArchInt documents can be extended instead of replicating the same characteristics in a new ArchInt document. ArchInt captures the feature of extending old ArchInt documents through a pair of tags called <uses_ArchInt>. This tag refers to the ArchInt documents that are going to be extended by their names, hence a pair of tags called <name> is introduced. Figure 5.5 illustrates an extract from the ArchInt document of the Applet architectural type to express the usage of the <uses_ArchInt> tag.

```
<ArchInt>
  <name>Applet</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
  <must__have>
      <Method>
        <name>setStub</name>
        <param>
          <string>AppletStub</string>
        </param>
        <returnType>void</returnType>
      </Method>
      .
      .
    </must__have>
</ArchInt>
```

Figure 5.5: An Extract of the Applet Architectural Type

The defined tags in this section are the ones that the prototype of the ArchInt specification language has defined at the moment. Although this is a small set of tags and only represents some of the requirements identified in Chapter 4, in fact the tags capture some of the essential and key features of architectural interface. Therefore, this set of tags formed the basis for a set of studies designed to evaluate the feasibility of architectural interface to support re-use. The studies have been confined to Java examples since this was sufficient to demonstrate the soundness of the basic idea to start with rather that attempting to generate completely a general solution at this stage of the development of the language. The next section presents the experimental work conducted in this research.

5.2 Experimental Work for Evaluating Architectural Interface

A number of studies were conducted as a *test-bed* [116] to provide evidence about the applicability of architectural interface for helping re-users firstly to identify re-usable components, and secondly to modify an existing system. A re-user is interested to find components in a repository, so identifying re-usable components is important in order to find them for re-use. In addition, a re-user might need to replace components from their system with others from a repository in order to fix a bug, satisfy new non-functional requirement (e.g. need faster response time), or adding more sophisticated functionality to their system. As a result, investigating how architectural interface can help to modify a system need also to be examined. These two types of studies (i.e. identifying and modifying) were selected as they seem to be the major concern of component re-users. So, the hypothesis of the overall experimental work was that:

Architectural interface represented in ArchInt can provide significant support to improve components re-use.

However, a first study before this evaluation was to examine whether ArchInt was sufficient and general enough to capture the characteristics of different architectural types. This was important to ensure that the other following studies are built on solid basis. The architectural type descriptions generated in this first study were then used in subsequent studies for evaluating the notion of architectural interface.

5.2.1 Study 1: Describing Different Architectural Types

The aim of this study was to examine whether the current features of prototype ArchInt language were sufficient to construct different architectural types. Therefore, the hypothesis of this study was that:

ArchInt is appropriate for defining the architectural characteristics of architectural types.

Based on the author's initial examination of a number of architectural types, three architectural types namely Applet, Eclipse, and MIDlet were been used in the study to investigate whether ArchInt would be sufficient to define these particular cases or whether it would need to be modified and improved. The Applet architectural type was selected as it is widely used in web page applications and as it is a simple architectural type to start the evaluation of ArchInt with. Eclipse architectural type was selected as it defines a rigorous framework for components integration and extensibility and it is used hardly in building extensible software systems. Moreover, Eclipse is one practical example of software systems that have their structures generated dynamically at run-time. The selection of MIDlet architectural type was made to examine the soundness of the notion proposed by ArchInt on a different platform other than desktop applications. It was felt that these three architectural types should be adequate to perform the experimental work at this early stage of the development of ArchInt.

1) Applet Architectural Type:

The Applet architectural type enables component to run in the context of another system (e.g. web browser). From studying several components in open-source repository systems that conform to the Applet architectural type and also looking into some of their specifications, the set of characteristics that components must satisfy in order to fit into an Applet system was found to be:

- must be a Java class
- must have the following methods:
  - `public final void setStub(AppletStub stub)`
  - `public void init()`
  - `public void start()`
  - `public void stop()`
  - `public void destroy()`
  - `public AppletContext getAppletContext()`

The above architectural characteristics were straightforward to capture using ArchInt. The ArchInt representation of the Applet architectural type is given in Figure 5.6 and uses the Java Class architectural type introduced earlier in this chapter.

```
<ArchInt>
  <name>Applet</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
  <must__have>
      <Method>
        <name>setStub</name>
        <param>
          <string>AppletStub</string>
        </param>
        <returnType>void</returnType>
      </Method>

      <Method>
        <name>getAppletContext</name>
         <returnType>AppletContext</returnType>
      </Method>

      <Method>
        <name>init</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>start</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>stop</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>destroy</name>
         <returnType>void</returnType>
      </Method>
    </must__have>
</ArchInt>
```

Figure 5.6: Applet Architectural Type Description

This example demonstrated that ArchInt can be used to write a document that captures the characteristics defined by the Applet architectural type that components must match in order to fit into an Applet system.

2) Eclipse Plug-in Architectural Type

Eclipse is an extensible IDE (Integrated Development Environment) that allows new tools to be plugged into the main Eclipse system to provide new functionality. Figure 5.7 illustrates a coarse-grained view of the structure of components (i.e. plug-ins) in an Eclipse system as it appeared in its original reference [32].

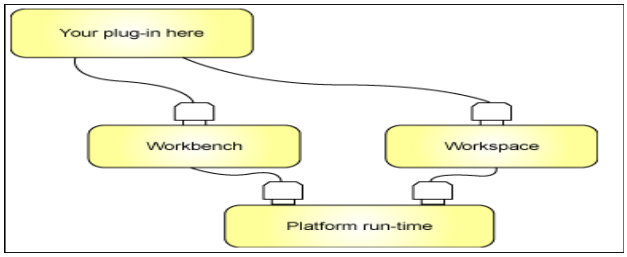


Figure 5.7: Eclipse Architecture [32]

A component cannot fit into an Eclipse system unless it matches the characteristics defined by the Eclipse architectural type. Manual examination of many open-source Eclipse plug-ins and exploration of their specification identified the basic requirements that characterize the Eclipse plug-in architectural type as:

- There must be a file called `plugin.xml` associated with the component. This file introduces a new plug-in to the Eclipse system and captures its details.
- A plug-in must have a number of methods that control the status of the plug-in during its lifetime within a system (e.g. running, active, passive).

The Eclipse plug-in architectural type is more complicated than the Applet architectural type in that its architectural characteristics are not purely based on source code but also on associated additional descriptive files (e.g. `plugin.xml`). The plugin.xml file must, in turn, conform to some characteristics that are required by Eclipse. For example, `plugin.xml` must be a well-formed XML file, and its contents must capture the details that Eclipse is expecting to find (e.g. id, name, extension-points). The ArchInt representation of the Eclipse architectural type is given in Figure 5.8.

```
<ArchInt>
  <name>Eclipse</name>
  <uses_ArchInt>
       <name>Java Class</name>
       <name>Eclipse XML</name>
  </uses_ArchInt>
    <must_have>
      <Method>
        <name>start</name>
        <param>
          <string>BundleContext</string>
        </param>
        <returnType>void</returnType>
        <exception>Exception</exception>
      </Method>

      <Method>
        <name>stop</name>
        <param>
          <string>BundleContext</string>
        </param>
         <returnType>void</returnType>
        <exception>Exception</exception>
      </Method>

      <File>
       <name>plugin</name>
       <type>XML</type>
      </File>

    </must_have>
</ArchInt>
```

Figure 5.8: Eclipse Plug-in Architectural Type

The ArchInt representation of the Eclipse plug-in architectural type uses the Java class architectural type introduced earlier, and defines a number of methods and a File that must be available in a component in order to fit into an Eclipse system. In addition, the ArchInt representation of the Eclipse plug-in uses the "Eclipse XML" architectural type that defines the characteristics that the Eclipse system is expecting to find in the `plugin.xml` file. The ArchInt representation of the "Eclipse XML" architectural type is given in Figure 5.9.

```
<ArchInt>
  <name>Eclipse XML</name>
  <uses_ArchInt>XML</uses_ArchInt>
  <must_have>
        <Field>
                <name>id</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>name</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>version</name>
                <dataType>String</dataType>
        </Field>

        <Field>
                <name>provider-name</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>class</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>extension-point</name>
                <dataType>String</dataType>
        </Field>
  </must_have>
</ArchInt>
```

Figure 5.9: Eclipse XML Architectural Type

As shown in the above figure, the "Eclipse XML" Architectural type defines a number of attributes that an Eclipse plug-in must provide values for, and these values are required by the Eclipse system. This example demonstrated that ArchInt is capable of describing the architectural characteristics of the Eclipse plug-in architectural type.

3) MIDlet Architectural Type:

MIDlet [84] is an architectural type targeting resource-constrained devices such as mobile phones and PDAs. Generally speaking, MIDlet represents the architectural types of games and applications that run on handheld devices. From a manual examination of several MIDlet components and the specification of MIDlet, the following key characteristics were identified:

- Must be a Java class.
- Must implement a number of methods that control the life cycle of the component.
- Must have a file that is of type JAD that describes the attributes of the component (e.g. version, vendor, name).

The ArchInt description of the MIDlet architectural type is represented using ArchInt in a similar manner to the previous two architectural types. The ArchInt representation of the MIDlet architectural type is provided in Appendix A due to its lengthy size and also as it does not demonstrate extra features to those that have already been picked in the previous two examples of architectural types.

Overall, this study has demonstrated that the current prototype of ArchInt can be used to represent the characteristics of different architectural types; hence the hypothesis of this study was contented. The next study is concerned with evaluating whether these architectural types represented using ArchInt can help to identify re-usable components accurately by examining them against some concrete samples (i.e. components) obtained from open-source repository systems.

5.2.2 Study 2: Identifying Re-usable Software Components

Chapter 4 identified that the potential benefit of using architectural types is to facilitate more structured searching of re-usable components. The aim of this study was to investigate that claim. This study concerned obtaining a number of components from existing open-source repositories and checking the components to identify whether they matched the architectural types constructed in the previous study. Therefore, the hypothesis of this study was that:

ArchInt helps to identify re-usable software components from open-source software repositories based on their architectural types.

Sourceforge.net is selected as an open-source repository for conducting this study as it is among the prominent open-source repository systems nowadays. Sourceforge.net supports searching queries written between quotations and also queries without quotations. A query that is written between quotations seems to return more focused results (i.e. exact match) than the one written without quotations. This study considers searching for software components using queries surrounded by quotations as the study was aimed at evaluating whether ArchInt can work to identify components based on the provided architectural type descriptions, hence exact matching results need to be considered at this stage of the development of ArchInt. The selection of components was done randomly using an applet that implement a random number generator using the formula "`1 + (int)(N * Math.random())`" as shown in figure 5.10 below [2]. The random number generator takes two integer values as inputs, the maximum number in a list and the

count of randomly generated numbers, and produce output based on the two provided inputs.

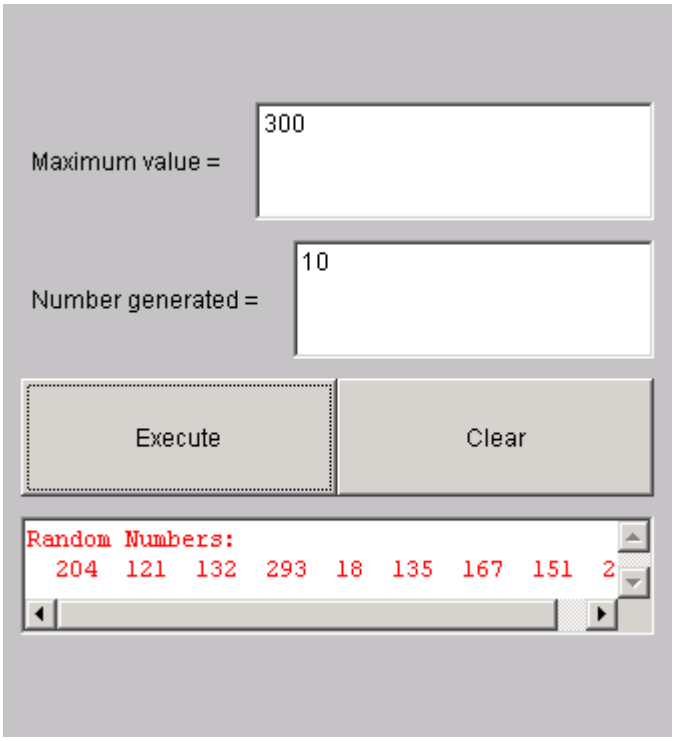


Figure 5.10: Random Number Generator [2]

Results that are listed in Sourceforge.net without their corresponding source code were discarded. Moreover, components returned by Sourceforge.net that are written in different programming languages than Java were not considered at this stage. The development status of components within the repository was not considered as a parameter in this study but it was noted for future reference in case there will be further investigation about the quality of re-using source code that is still under development.

The terms used to search for software components are those that were observed common among various repository systems or those that precisely state the name of an architectural type. However, there might be other expressions of use that were beyond the knowledge of the author; hence the selected terms were not claimed to be extensive. Samples of 30 components were selected randomly for each architectural type from within the sample frame (i.e. Applet, Eclipse, and MIDlet). This number was felt sufficient to build the required confidence about the soundness of the approach as it represents an average percentage of 10% of the number of results listed by Sourceforge.net in response to the searching queries for each architectural type in this study.

A tool, named *ArchIntParse*, was developed for performing the automated matching of the source code of a component to an architectural type. The main functionality of the tool

is to read through an ArchInt document and then parse the source code of a provided component to identify the characteristics that match those defined by one of the three architectural types generated in the first study.

The approach followed by the *ArchIntParse* tool for matching an architectural type document to a provided component is based on utilizing the compiler associated with the programming language identified by the `<programming_language>` tag to check the syntax of a component and also identify whether a component is missing any of its required sub-components (i.e. internal dependencies). The tool works by automatically generating a "TestSuite" Java class from an architectural type document. The TestSuite class contains code to exercise all of the features specified in the architectural type document. For example, the class will contain method calls representing invocations of all of the methods identified in the `<must_have>` tag. Figure 5.11 illustrates an example of the automatically generated TestSuite java class to match the source code of an Eclipse plug-in to the Eclipse plug-in architectural type.

```
package Deserializer;
import tmp.com.nilswinkler.eclipse.accessmodifier.AccessModifierPlugin;
public class TestSuite
{
    AccessModifierPlugin instAccessModifierPlugin = new
AccessModifierPlugin();
    public void test()
    {
       org.osgi.framework.BundleContext a0 = null;
       org.osgi.framework.BundleContext a1 = null;
       try
        {
             instAccessModifierPlugin.start(a0);
        }
        catch(Exception e){}
       try
        {
             instAccessModifierPlugin.stop(a1);
        }
        catch(Exception e){}
    }
}
```

Figure 5.11: TestSuite Class

The tool then compiles and links the generated Java class with the source code of the provided component. Figure 5.12 illustrates an extract of the output generated from executing the *ArchIntParse* tool on a number of Eclipse plug-ins.

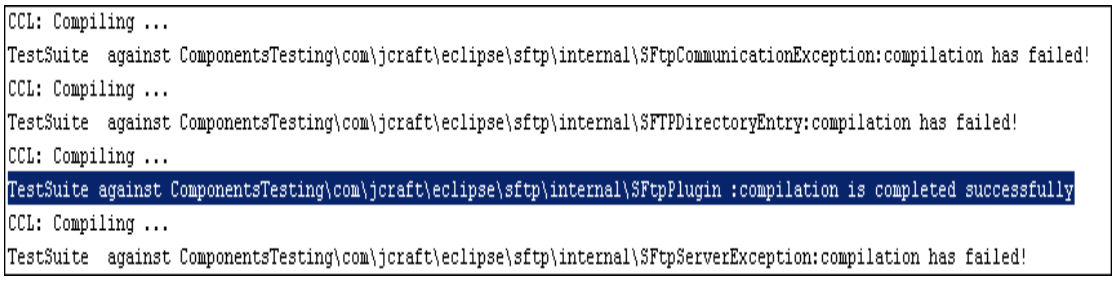


Figure 5.12: ArchIntParse Tool Output

If no compile-time or link-time errors are raised, this indicates that the provided source code matches the architectural type that was used to generate the TestSuite Java class, and the tool returns a positive result. If compile or link errors are raised, this reflects a mismatch and the tool returns a false match result. The design and implementation of the *ArchIntParse* tool is given in Appendix B.

This study was conducted in several iterations. Each iteration evaluated software components against one of the architectural types generated in the first study.

First iteration

This iteration checked the source code of components obtained from Sourceforge.net against the definition of the Eclipse plug-in architectural type. Sourceforge.net was searched for Eclipse plug-in components, using the normal text matching search, for the phrase “Eclipse plugin” provided by the Sourceforge.net repository. The searching phrase returned 279 components as at 12/2008 that only contain the phrase “Eclipse plugin”. The selection of components was made using the random number generator to select randomly a total of 30 samples out of the 279 results to be checked by the *ArchIntParse* tool against the Eclipse plug-in architectural type document. In addition, a random selection of another 30 components that do not include the phrase “Eclipse plugin” in their description was made.

Results

Considering first the results from the 30 Eclipse components that Sourceforge.net provided as Eclipse plug-ins, the tool identified 22 components out of the 30 as conforming to the Eclipse plug-in architectural type, while eight components were identified as not conforming. To check the validity of the generated results, all 30 Eclipse components were tried as plug-ins in an Eclipse system. The 22 components that were identified by the *ArchIntParse* tool as conforming to the Eclipse plug-in architectural type were all recognized and run successfully in the Eclipse system. The remaining eight components did not work in the Eclipse system.

The other 30 components that did not contain the word “Eclipse plug-in” in their description in Sourceforge.net were checked by the *ArchIntParse* tool and also tried in the Eclipse system. The *ArchIntParse* tool indicated an unsuccessful match against the Eclipse plug-in architectural type, and this was confirmed by the components not executing successfully within Eclipse.

Discussion

Visual inspection of the source code confirmed that all eight of these non-conforming components did not implement the methods defined by the Eclipse plug-in architectural type; in addition, three components of them were also missing the `plugin.xml` file. as a result, all the eight components did not work in the Eclipse system. However, five components out of the eight were recognized by the Eclipse system as they had the `plugin.xml` file, but resulted in run-time errors. The remaining three components those were missing the required `plugin.xml` file were not recognized at all by the Eclipse system as expected.

The results obtained demonstrated that ArchInt successfully identified the salient characteristics of requirements of architectural fit into an Eclipse system, and captured that in an architectural type description using the ArchInt prototype language. In addition, the experiment showed that the defined characteristics of the Eclipse plug-in architectural type represented by ArchInt have been successfully matched automatically by a tool. The study also demonstrated the usefulness of the *ArchIntParse* tool. The *ArchIntParse* tool can be utilized to identify the exact reason for the non-matching components and provide the information to a re-user, the provided information could be helpful if the re-user wanted to fix the component.

This iteration also revealed a weakness in Sourceforge.net as it listed components that do not match the characteristics of the Eclipse architectural type, but the repository has considered them mistakenly as matching ones. A possible justification of listing these erroneous results by Sourceforge.net is that the provider of these components seemed to assume that re-users of the components should be responsible for implementing the required architectural characteristics. The providers only focus on producing components that provide certain behaviour without completely concerning about their architectural aspects. As a result, the providers of these components considered them as Eclipse plug-ins, even though they do not practically match the full characteristics of the Eclipse architectural type. This problem could have been avoided if Sourceforge.net used checking

mechanism to validate components' characteristics against the claimed architectural type of components by their providers.

Second iteration

This iteration involved the Applet architectural type. Text matching in SourceForge.net was used again, but this time with the string "Java Applet". A list of 120 results that contained the phrase "Java Applet" was returned by Sourceforge.net as in 12/2008. A number of 30 of these components were selected randomly using the random number generator to be checked by the *ArchIntParse* tool against the ArchInt document for the Applet architectural type. In addition, 30 other components were selected from Sourceforge.net that did not contain the words "Java Applet" in their description to be examined by the *ArchIntParse* tool against the Applet architectural type.

Results

The *ArchIntParse* tool identified that 23 of the 30 components matched the Applet architectural type document. The remaining seven components were flagged as not matching. The other 30 components that Sourceforge.net did not consider them as Applets also did not match the Applet architectural type.

To check the validity of the generated results, all the 30 components that identified by Sourceforge.net as Applets were tried on an Applet system using a normal `appletviewer` utility. It was found that 28 components out of the 30 components ran successfully, including the 23 components that the *ArchIntParse* tool had identified as being instances of the Applet architectural type. Two components did not run successfully, all of which were correctly identified by the tool as not being instances of that architecture type. The five components that apparently were successfully executed as Applets and were not correctly identified as matching by the tool are discussed further below.

Discussion

Inspecting by hand the source code of the five components that returned negative result by the *ArchIntParse* tool showed that all the components matched the characteristics defined by the Applet architectural type. After examining the possible reasons for the conflict in results obtained by the *ArchIntParse* tool and by trying the components on the Applet system, the reason for the conflict was identified. The five components for which negative results were returned by the *ArchIntParse* tool were delivered by the Sourceforge.net repository missing some their internal dependencies. As a result, the compile-and-link process in the *ArchIntParse* tool failed. This was the real reason that

caused the *ArchIntParse* to return negative results and not because the two components were not conforming to the Applet architectural type. So, this result is considered a false negative result as the failure in the compilation was not due to missing any of the characteristics of the Applet architectural type but it was related to missing internal dependencies that allow the components to work in an Applet system. Despite the false negative results, this results obtained in this iteration are promising.

Overall, this iteration demonstrated that the Applet architectural type represented by ArchInt has worked successfully to check and identify automatically the conformance of software components to the Applet architectural type.

Third iteration

This iteration concerned evaluating the ArchInt description generated in the first study for the MIDlet architectural type to identify matching components to that architectural type. As before, text matching of words was used against component descriptions in Sourceforge.net, using the string “J2ME” as that is the common term used to search for MIDlet components. A number of 300 components were listed as a result of the search. A random selection of 30 components that contained the word “J2ME” in their description in Sourceforge.net was made. Moreover, another 30 components that did not include the word “J2ME” in their description were selected. All selected components were checked by the *ArchIntParse* tool against the ArchInt document of the MIDlet architectural type.

Results

The *ArchIntParse* tool identified the 28 components that contained the word “J2ME” in their descriptions as conforming to the MIDlet architectural type. Two components out of the 30 did not succeed in the *ArchIntParse* tool check. The other 30 components that did not include the word “J2ME” in their descriptions were identified by the tool as not conforming to the MIDlet architectural type.

To check the validity of the generated results, all the 30 components that contain the word “J2ME” in their description in Sourceforge.net were tried in a MIDlet system (e.g. J2ME application server). Among the 30 components, 28 components (including the 28 components for which positive matches were obtained by the *ArchIntParse* tool) ran successfully in the MIDlet system indicating that the architectural type description accurately reflected the MIDlet architectural fit requirements. The other 30 components that obtained negative matching results from the *ArchIntParse* tool did not execute in the

MIDlet system to further reinforce the utility of the approach. The two components that did not pass the check by the *ArchIntParse* tool are discussed further below.

Discussion

A visual inspection of the two components that failed to pass the check by the *ArchIntParse* tool was done. It was found that both components were not MIDlet components as they were missing the characteristics of the MIDlet architectural type. The two components that were found as not conforming to the MIDlet architectural type were mistakenly considered by Sourceforge.net as MIDlet components while they are not according to the characterization of the MIDlet architectural type defined by ArchInt. The results obtained in this iteration indicate that the accuracy of matching obtained from using ArchInt was advantageous over the results obtained by using the free-text searching facility in Sourceforge.net. This uncovered an additional problem that can be encountered by re-users when searching for re-usable components in Sourceforge.net, hence ArchInt can be useful to certify that a component is correctly categorized as described.

Overall, this iteration demonstrated that the representation of the MIDlet architectural type in ArchInt has worked successfully to check and identify automatically the conformance of software components to the MIDlet architectural type. Moreover, the iteration revealed that ArchInt could be utilized to certify the correctness of components in terms of their architectural type.

Summary of the Second Study

The results of the study conducted in the three iterations are summarized in Table 5.1. The table shows the total number of components tested by the *ArchIntParse* tool in the study, the percentage of components that returned true positive results, the percentage of components that returned false positive results, the percentage of components that returned true negative results, and the percentage of components that returned false negative results. The true positive column indicates that components were identified by the *ArchIntParse* tool as conforming to an architectural type, whereas the false positive column indicates that components that conformed to one architectural type incorrectly matched a different architectural type (e.g. a component that conforms to Eclipse plug-in architectural type passes the test against the ArchInt of the EJB architectural type). The true negative column indicates the percentage of components that have failed in the compilation process with the generated “TestSuite” Java class due to missing some characteristics required by an architectural type, while false negative column denotes the percentage of components that

failed in the compilation process but due to reasons other than missing some of characteristics of an architectural type (e.g. missing internal dependencies).

Architectural Type	Number of samples	True Positive (%)	False Positive (%)	True Negative (%)	False Negative (%)
Eclipse	30	73	0	27	0
Applet	30	77	0	6	17
MIDlet	30	93	0	7	0
Non-Eclipse	30	0	0	100	0
Non-Applet	30	0	0	100	0
Non-MIDlet	30	0	0	100	0

Table 5.1: Summary of the Results of the Second Study

An interesting observation from the above table of results is that the *ArchIntParse* tool never returned any false positive results. This observation indicates that the representation of the different architectural types in ArchInt was useful and reflected precisely the characteristics of different architectural types without mixing one architectural type's characteristics with another. This study has demonstrated that ArchInt can be used to identify software components, hence satisfied to an extent the hypothesis of the study.

5.2.3 Study 3: Modifying an Existing Software System

Modifying a software system is usually related to modifying its functionality and can be accomplished either by replacing components from the system with others that provide the necessary functionality or by extending the system with new components. Also, system modification can be considered in a case where non-modifiable components are needed to be re-used in a system. So, the architectural type of a system can be adjusted to match the characteristics of the architectural type that the non-modifiable components are matching. However, modifying the characteristics defined by a system to fit components is not part of this study as the main focus concerns modifying a system by replacing or adding components to it. Modifying a component-based software system requires the presence of a precise characterisation of the interfaces of the composing components of the system in order to facilitate unplugging components from a system and plug new ones into it. If a system developer needs to replace one component with another one, then what the

developer ought to understand is the architectural interface of the component that needs to be replaced. The aim of this study was to evaluate ArchInt for providing assistance to the task of modifying an existing software system by replacing some of its composing components that could be obtained from a repository system. The hypothesis of this study was that:

ArchInt can expose component's interfaces and significantly facilitate system modification.

This study was undertaken in two iterations. The first iteration was concerned with replacing a component from a system that requires its composing components to adhere to a rigorous architectural type (i.e. an architectural type that defines precisely a required set of characteristics). The system used in the first iteration was the Eclipse system, which was defined and examined in the previous studies. The second iteration concerned replacing a component from a system that inaccurately specifies what architectural characteristics components must have in order to fit into it. The example system used in the second iteration was the Model-View-Controller (MVC [131]) design pattern system. The MVC system lacks a precise characterization of what defines its architectural characteristics at the source-code level. As a result, one may find two components that are considered by their developers as conforming to the characteristics defined for a Model within an MVC system but the external interface of the two components matches different characteristics than each other. The reason for selecting these two types of systems was to examine the usefulness of ArchInt to model the interfaces of software components in two different extremes with respect to the preciseness of the architectural characteristics. One where architectural characteristics are precisely defined (e.g. Eclipse) and another one where the architectural characteristics are ambiguous (e.g. MVC).

First iteration

In this study the Eclipse wizard was used to build automatically a simple plug-in without worrying about the details of the required architectural characteristics as the wizard can generate them automatically. After generating the plug-in, it was incorporated into the Eclipse system and tested to examine that it fit into the Eclipse system. Assuming that the incorporated plug-in provides functionality that needs to be replaced with other component; the study proceeded by trying to extract the plug-in that has been incorporated earlier and replaces it with another plug-in in order to accomplish the objective of this study. The *ArchIntParse* tool was selected to be the new component that had to replace the

extracted Eclipse plug-in. Since the tool did not satisfy the requirements of the Eclipse plug-in architectural type, it had to be modified to provide the required characteristics:

- Must have the necessary methods required by the Eclipse plug-in architectural type; and
- Must have a necessary `plugin.xml` descriptor file.

So, a file called "`plugin.xml`" was generated by hand for the *ArchIntParse* tool to satisfy one of the requirements of the Eclipse plug-in architectural type as described in the "Eclipse XML" architectural type defined earlier. The *ArchIntParse* tool's source code was also modified to implement the necessary life cycle methods as defined in the Eclipse plug-in architectural type.

The *ArchIntParse* tool was compiled and linked successfully and then incorporated into the Eclipse system to run it. The component was recognized by the Eclipse system and ran successfully. The generated *ArchIntParse* Eclipse plug-in was also checked using the *ArchIntParse* tool itself and it was found matching to the Eclipse plug-in architectural type.

Discussion

ArchInt was useful in this study to understand the salient characteristics of the Eclipse plug-in architectural type that the *ArchIntParse* tool was required to match in order to fit into the Eclipse system. Although the necessary architectural characteristics might be provided automatically by the wizard of the Eclipse IDE, ArchInt could be generalized to identify and check the required architectural characteristics of other architectural types as demonstrated in the previous studies. This study also demonstrated that ArchInt was useful to understand what is required to perform the modification from one architectural type to another.

Second iteration

Figure 5.13 illustrates the architecture of one implementation of an MVC system as found in Java2s repository system [76]. This study involved replacing a component that matched the observed characteristics of the Model architectural type as identified in this system (i.e. "ContactModel") with another modelling component obtained from a repository system.

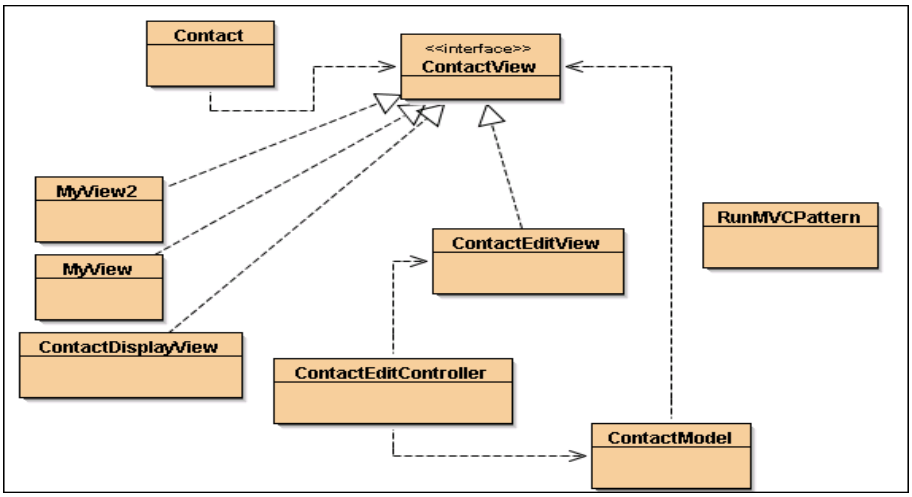


Figure 5.13: Contact MVC System

The Java source code of “ContactModel” is inspected by hand in order to identify the architectural characteristics that would constitute a description for the Model architectural type according to the implementation of the “ContactModel” in this system. The observed characteristics of the Model architectural type of this system are illustrated in Figure 5.14. The *ArchIntParse* tool was used to check the generated ArchInt document for the Model architectural type against all the Java classes in this system and the “ContactModel” Java class was verified as the only one that matched the generated ArchInt document.

```
<ArchInt>
  <name>Model</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
    <must_have>
      <Method>
        <name>refreshContactView</name>
        <param>
            <string>String</string>
            <string>String</string>
            <string>String</string>
            <string>String</string>
        </param>
        <returnType>void</returnType>
      </Method>
      <Method>
        <name>addContactView</name>
        <param>
          <string>ContactView</string>
        </param>
         <returnType>void</returnType>
      </Method>
      <Method>
        <name>removeContactView</name>
        <param>
          <string>ContactView</string>
        </param>
        <returnType>void</returnType>
      </Method>
      <Method>
        <name>updateModel</name>
        <param>
            <string>String</string>
            <string>String</string>
            <string>String</string>
            <string>String</string>
        </param>
        <returnType>void</returnType>
      </Method>
    </must_have>
</ArchInt>
```

Figure 5.14: Model ArchInt

An attempt was made to try to find a component from open-source repository to fit into this system. Although, finding component was not intended to be part of the evaluation in this study, it was done to examine whether the identified characteristics of the Model architectural type in this example are common to all Model architectural types. It was not possible to search open-source repository systems using the characteristics of the Model architectural type identified above as open-source repository systems do not currently support searching for components based on the characteristics defined by an ArchInt document. The only possible way to search was by searching the open-source repositories using the text-matching approach of instances of the Model architectural type available in the repository (“e.g. “servlet”, “JavaBeans”). However, the searches retrieved results that were not re-usable as they were not conforming to the architectural type of the above MVC system in this study.

With respect to modifying the above system, the defined architectural characteristics in Figure 5.14 was used to understand how the “ContactModel” component can be replaced.

So, the component was replaced successfully from the system with another Java class that was generated manually and implemented conforming to the Model architectural type of this system. The new added component fit in the system and ran as expected.

Discussion

A problem encountered when generating the ArchInt document for the Model architectural type of the above MVC system was that the identified architectural characteristics are not fixed for every Model architectural type as it is observed that different characteristics for the Model architectural type were available. For example, one implementation of the Model architectural type might consider implementing data exchange between the instance (i.e. component that conforms to an architectural type) of the Model and the instances of the other architectural types (e.g. Controller and View) using a push-model [42]. The push-model concerns transferring data out of an instance of the Model to other components whenever changes in the state of the instance of the Model occurs, hence requiring the View component to register with the Model component. According to this implementation, a Model component must have a method called "`public void addContactView(ContactView)`" as defined in the Model architectural type in this study. Another implementation of the Model architectural type, however, might be to exchange data by applying a pull-model [42]. An instance of the Controller component would then need to keep checking changes in the state of the Model component and pull data from the Model component as appropriate, thus requiring the View component to register with the Controller. According to this implementation, the component that conforms to the Controller architectural type is the one that must have a method "`public void addContactView(ContactView)`" and not the Model as described earlier. This variation in the implementation of the various components of the MVC system indicates the lack of a precise definition of what the characteristics of the Model, View, and Controller architectural types are.

It seems that the variety in describing the characteristics of the Model, and also View and Controller, of an MVC system is caused as the three architectural types are in fact metaphors normally used at the design stage to identify the high-level architecture of a system. The component that is responsible for storing and manipulating data in a system can be considered abstractly as an instance of a Model. The architectural types of an MVC system are defined abstractly but their definitive characteristics are left for programmers to determine at implementation time, and hence variety in the characteristics of the Model, View, and Controller architectural types resulted.

An advantage of ArchInt is observed in this iteration indicating that the identified characteristics of the Model architectural type depicted in Figure 5.14 can be used to understand what is required to modify a component to fit in the place of the replaced instance of the Model architectural type in that MVC system. If ArchInt was not provided, then a developer would need to identify the interfaces of the components of the system at hand manually, which can be difficult and time consuming.

5.2.4 Observation

The two studies (i.e. identifying components and modifying system) have shown significant contribution to the notion of architectural interface to support component re-use. The study that concerned identifying component from open-source repository system uncovered some interesting aspects about the applicability of the notion of architectural interface to identify and certify component conformance to an architectural type. Components in Sourceforge.net are not checked with respect to their architectural type, hence the notion of architectural interface can be utilized to improve the functionality of that repository. An additional observation on Sourceforge.net is that not all of the components listed in that repository are really open-sourced. Examining various components from Sourceforge.net revealed that some of them are available without their corresponding source code, and some other components are available with only part of their source code. As a result, it is felt that Sourceforge.net seems to violate the definition of open-source software with respect to making source code available, and hence should not be considered as an open-source repository system.

The other study that concerned modifying a system also recorded promising success indicating that architectural interfaces are useful to accomplish the modification as they explicitly describe the architectural characteristics of software component. Moreover, the study revealed a new interesting area where architectural interface could be useful. As a result of the studies, it seems that the overall hypothesis of the experimental work that stated "*Architectural interface represented in ArchInt can provide significant support to improve components re-use*" has been considerably satisfied.

5.3 Summary

This chapter has introduced a prototype of a specification language, namely ArchInt, that formalised some aspects of the notion of architectural interface defined in Chapter 4. The chapter has described the experimental work conducted for evaluating the sufficiency

of ArchInt to represent different architectural types. In addition, the chapter has examined the generated descriptions of the three architectural types represented in ArchInt to support components re-users. The chapter also demonstrated the applicability of ArchInt to assist re-users in modifying an existing software system. The results of the studies described in this chapter are going to be used in the next chapter to form an overall assessment of the architectural interface approach established in this research.

Chapter 6 - Evaluating the Achievements of the Research

The previous chapter presented the experimental work conducted for evaluating architectural interfaces from the perspective of the support that can be provided to facilitate component re-use. A prototype of a specification language called ArchInt was developed to formalize some of the concepts of architectural interfaces for the studies.

From the experience gained from the work described in the previous chapter and from this research overall, this chapter evaluates outcomes of this research based on the objectives identified in Chapter 1:

1. To identify the design of an ideal repository system.
2. To investigate the possibility of characterizing components at the source-code level.
3. To uncover the architectural characteristics and dimensions that correspond to fitting components architecturally into a system, in order to address the use of these characteristics within a repository.
4. To propose an approach, namely ArchInt, that formalizes the architectural interface at a low level of abstraction that reflects the precise characteristics of software components.
5. To investigate the applicability of ArchInt in automating the process of categorizing and modifying software components.

The building blocks of the ideal repository system are centred on the implementation of architectural interface (i.e. ArchInt). Therefore, the evaluation in this chapter will not follow the sequence of the objectives specified above. The first aspect that is going to be evaluated is whether ArchInt can precisely identify the characteristics of software components as that was the major concern of the ideal repository system to support re-use. This evaluation leads into evaluating Objective 4. After that, the notion of architectural interface is going to be evaluated to verify that it has identified the architectural characteristics of software components. This evaluation will lead into evaluating Objective 3. Then, based on the observation obtained from the previous evaluations, Objective 2 will be evaluated. Objective 5 will be evaluated based on the results of studying with ArchInt to satisfy some of the key requirements identified by the use-cases of the ideal repository system. Finally, the evaluation of the first objective will be drawn from evaluating the suitability of ArchInt to satisfy the overall design of the ideal repository system.

6.1 Evaluate ArchInt for Representing Precisely the Characteristics of Source-Code Components

Generally, the source code is the precise representation of a software system, and it captures implicitly the design decisions imposed by developers on the system that is going to be built. Considering re-use at the source code level requires the ability to represent source-code components in a meaningful way that can be utilized by re-users in order to allow them to find re-usable components. The representation of source-code is in fact the meta-data that describes aspects of source-code components. The organisation of the components in a repository system can be based on the meta-data that defines their characteristics.

It was discussed in Chapter 5 that ArchInt was a prototype of a specification language for capturing some of the key characteristics that concern fitting components into a re-user's system, and hence represents the meta-data for source-code components. The studies conducted in this research demonstrated that ArchInt satisfied the following features:

- Extensible: one ArchInt document can re-use another ArchInt document to define new characteristics instead of replicating the definitions of characteristics. This feature was examined in the first study where the architectural type of a Java class was re-used to define the Eclipse, Applet, and MIDlet architectural types.
- Flexible: the tags defined by ArchInt are general enough to represent the characteristics of different architectural types. This generality was obvious in the first study that used ArchInt to represent the architectural characteristics of a number of different architectural types. The same set of tags were been used to define the architectural characteristics of the Applet, Eclipse plug-in, and MIDlet architectural types without the need to add new tags or modify the available ones.
- Precise: ArchInt can accurately define the exact characteristics that software components must have in order to fit into a system. This precision in the defined characteristics was examined in Study 2 when a number of components were matched against ArchInt documents to identify their architectural types.

The result of the studies discussed in Chapter 5 was broadly in line with expectations indicating that ArchInt did capture key characteristics of source-code components.

6.2 Uncovering Architectural Characteristics with the Notion of Architectural Interface

In the early stages of a software development process (e.g. the design stage) architectural characteristics are normally identified abstractly. Many architectural description languages (ADLs [134]) capture some of the key architectural characteristics of components at a very high level of abstraction. For example, in ACME [59], the notion of *port* is developed to indicate an entry or exit point of data and control into a component and to establish interaction with other components in a system. However, at the source-code level, one may not be able to tell just by inspecting the source code which part of the source code is related to defining a *port* of a component and which part is related to providing functionality or implementing security requirements. Like many aspects of a system's design, they are not carried through directly into something recognisable in the source code or even traceable from the source code back to parts of the design.

Identifying functional and architectural characteristics in the source code is necessary to re-use components. Moreover, automatic identification of a component's characteristics is desirable to further enhance re-use. However, identifying the characteristics of components might be extremely hard in the case of using a tool to identify them automatically, unless there is something in the source code of a component that can be used by a tool to identify the characteristics required.

A system model, illustrated in Figure 6.1, was generated to help understanding the characteristics of software components. From that model, the notion of architectural interface has originated.

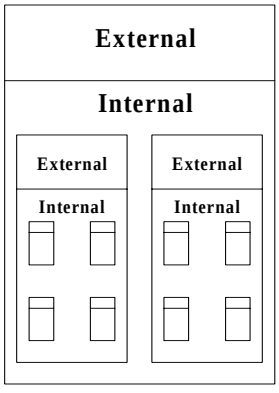


Figure 6.1: System Model

This system model was meant to be simple, so it covers a wide range of software systems. The model identified two basic requirements that software components should satisfy in order to be re-used successfully in a system. One requirement concerned how

components can fit into a system, and the other requirement addressed the issue of how components can work in a system. The model identified that the characteristics required to fit components into a system should be defined through the external interface of the components, while the characteristics that components require to work in a system should be defined in the internal interface of the system.

A number of key architectural characteristics were identified in Chapter 4. These characteristics were defining different aspects that can affect fitting software components architecturally into a system. A number of the identified architectural characteristics were examined in Chapter 5 to verify that they were the characteristics that must be considered to satisfy the requirement of architectural fit. The results of the studies were entirely positive indicating that the identified characteristics were really addressing aspects of architectural fit, hence were related to the architectural characteristics of software components. The outcome of the studies indicated that the notion of architectural interface developed in this research was useful to identify the architectural characteristics of software components.

6.3 Characterizing Source-Code Components

The absence of high-level artefacts (e.g. design documents) is the major cause of the difficulty in understanding software components and of identifying their distinguishing characteristics that can be used to characterize components for re-use. This is the case in most of the available source-code components in open-source repository systems.

Prior work discussed in Chapter 3 based a characterization of source-code components on some representation of their functionality. Nevertheless, representing the functionality of source-code components has not gained widespread success due to the difficulty of defining the semantics at the source-code level. It was identified in Chapter 4 that a re-usable component is the one that provides the required functionality and also fits into a system, hence is a “perfect fit” candidate. While characterizing source code based on functional characteristics is not adequate as discussed in Chapter 4, this research has examined the possibility of characterizing source-code components based on identifying their architectural characteristics.

As discussed in Section 6.2, the notion of architectural interface was introduced to identify the key architectural characteristics of source-code components, and instances of architectural interfaces were formalized using ArchInt. The study of using the generated ArchInt documents for finding components from open-source repositories examined

whether the identified characteristics were useful to characterize source-code components and help find them. The results of the study denoted that 81% of the examined source-code components matched the characteristics defined by the corresponding ArchInt documents. This percentage is really promising, indicating that the identified characteristics accurately characterize source code with respect to an architectural type.

6.4 Evaluating the Usefulness of ArchInt to Support the Basic Functionality of a Repository System

The basic functionality that a repository system must provide in order to facilitate re-use is the support to find, modify, automatically characterize, and deliver fully working software components. Matching component characteristics against the meta-data held in a repository is a prerequisite for finding and automatically characterizing components; hence matching against meta-data will be discussed first in sub-section 6.4.1. After that, the automatic identification of component characteristics will be discussed in sub-section 6.4.2. Component modification will be covered in sub-section 6.4.3. Finally, the issue of delivering fully working components will be discussed in sub-section 6.4.4.

6.4.1 Matching Component Characteristics to Meta-data

The notion of matching against meta-data was introduced by Zaremski and Wing [158] to match the signatures of functions in software libraries. The *function signature matcher* approach they developed is similar to the matching performed by the *ArchIntParse* tool developed in this research, in the sense that both approaches are trying to identify significant characteristics of software components from their source code. However, the difference is in the capability of the meta-data used to perform the matching. ArchInt establishes an ontology, which relates to component fit, for defining criteria to help in searching for re-usable components. In contrast, the meta-data used by *function signature matcher* assumes a re-user's knowledge of the exact signature of functions in a library, and this relates to issue of understanding semantics that is inherently difficult to describe (as discussed in Chapter 4). Moreover, ArchInt is more general than the meta-data used by their *function signature matcher* in the sense that ArchInt can define characteristics other than signatures of software components (e.g. required fields and files required).

The repository systems reviewed in Chapter 3 lacked the necessary checking capability to ensure the conformance of the deposited software components to the characteristics claimed by their providers. For example, in Sourceforge.net, a component's provider can

provide a textual description and also select the most appropriate characteristics for the component from those built into the repository (e.g. environment, operating system). The component is then deposited into the repository system and indexed as appropriate without any verification of whether the deposited components match the characteristics defined by the component's provider or not. Any errors in the characteristics recorded, or by any inaccuracies in the textual description, will then affect re-users whose searches identify components that were not what was expected.

Experimenting with ArchInt demonstrated how ArchInt can be utilized to check the characteristics of software components and ensure that components are as advertised, hence will meet the expectation of re-users. If a component is claimed to be an Applet component, the corresponding ArchInt definition of the Applet architectural type can be used to verify that the component really is an Applet as claimed. The results of the studies were promising with a matching rate of 81% of all of the components that were extracted from Sourceforge.net to the defined ArchInt documents, and also the matched components were demonstrated to actually fit into the corresponding systems. So, ArchInt was useful to do the necessary check of the architectural type of software components.

6.4.2 Automatic Identification of Software Components

A component might be deposited into a repository system without the component's provider defining its architectural type. One way to identify the architectural type of the deposited component is to match it against the architectural type descriptions held in the repository. If a match is found, the component is an instance of that architectural type.

The studies reported in Chapter 5 demonstrated how successful the matching of architectural types was even with the prototype ArchInt specification language. Every component obtained from Sourceforge.net was checked against the generated definitions of architectural types. The components that passed the compilation process were considered to be instances of the architectural type used to check the components. The results of the studies were entirely positive as ArchInt documents were processed automatically to do the necessary check of the architectural type of software components. The results obtained from the study indicated that ArchInt accurately represented the meta-data necessary to support automatic identification of software components, and thus could be utilized by a repository system to identify the architectural type of any deposited components.

The second iteration of the third study that involved the MVC system, using ArchInt raised some interesting cases regarding the potential usefulness of ArchInt to standardize

the interfaces of software components. Despite the MVC system being a common design pattern, the (limited) study demonstrated just how little help this common pattern was in supporting re-use. ArchInt did not work well for this study. The characteristics of the Model architectural type in that system were used to examine a number of source-code components obtained from open-source repositories, but no components were found matching the characteristics of that Model architectural type. The lack of the usefulness of ArchInt in this study was because the characteristics of the Model component had not been standardized by those who have generated the notion of Model in an MVC system. However, there is still the possibility of utilizing ArchInt to affect the re-usability of components positively in systems that suffer from problems similar to those of the MVC system. If the architectural types for Model, View and Controller had been defined and used by developers when developing their MVC systems, re-use might then have been possible as it would be possible to search for components (e.g. Model) based on their standard characteristics. Moreover, if ArchInt was generated for every component in that MVC system, a re-user could understand what should be done to modify a component in order for it to fit into that system. An ArchInt document was generated for the Model architectural type in the study. The generated ArchInt document has helped to understand the precise characteristics that a component must match in order to fit as a Model into the MVC system of that study. Although ArchInt was not useful to identify components automatically in this study, it has been demonstrated that ArchInt could be useful to standardize the interfaces of software components.

6.4.3 Support for Component Modification

Modifying source code components can be achieved in three ways. One way is by changing the source code of the component that is going to be re-used to match the requirements of a system. Another way for modification is to change the source code of the system to match the interface of the component that is going to be re-used. A third way for modification is to wrap [130] the component with the necessary changes to make it re-usable in a system without affecting its original interface or changing the interface required by a system. Any of the three ways of modification could be employed, however, the most effective way is the third one as the modification will not involve any changes to either the component or the system, and this is the form of modification considered in the studies of this research.

The study that involved modifying the *ArchIntParse* tool from the Java application architectural type to the Eclipse plug-in architectural type demonstrated the significance of architectural interfaces in helping to understand what is required to modify a component to fit into a system. In fact, the description of an architectural type using ArchInt provides useful documentation to help a developer understand the characteristics of software components. The modification had not affected the functionality of the *ArchIntParse* tool, which indicated that architectural interface can be used as a means for separating architectural characteristics from the other characteristics (e.g. functionality). This separation could be advantageous to automate the modification of software components from one architectural type to another. A tool could identify the characteristics required to fit a component into a system by parsing the ArchInt document that represents the system's architectural requirements and generating the necessary modifications automatically.

Although the notion of architectural interface was found useful for component modification, the studies revealed that it does not currently specify the source code necessary to map between the new interface of the component after wrapping and the component's interface before wrapping. For example, if a component originally matched the MIDlet architectural type but a re-user has wrapped the component to fit into an Eclipse system, then a mapping source code is required after the component has been wrapped. A re-user must write the source code that establishes the mapping between the method required by the Eclipse system and the method already defined in the component. The notion of architectural interface is not concerned with capturing the relevant characteristics to address this mapping between the old interface of a component and the new one. Although this additional source code is not part of the requirements of architectural fit, it is still important to be defined to help support the automatic modification of components from one architectural type to another.

6.4.4 Delivering Fully Working Components

A re-usable component might be found in a repository system, but the component may not work in a re-user's system because it was delivered without some of its internal dependencies (i.e. sub-components). Generally, architectural interfaces identify characteristics that relate to the internal dependencies of software components. However, these characteristics were not part of the investigation of the studies, because it was felt that internal dependencies are not a new feature and are already demonstrated in other tools

(e.g. the *make* tool). As a result, the investigation of delivering fully working components has not been covered.

6.5 The Design of the Ideal Repository System

The results of the evaluation discussed so far are the key determinant to derive the evaluation of Objective 1 from the list of objectives. The fundamental issue of characterizing source-code components was achieved successfully in this research and the evaluation has reflected that success. A number of architectural characteristics were identified and the evaluation showed the successful uncovering of the key architectural characteristics of software components. The evaluation of the developed prototype of the ArchInt specification language has also shown success in representing the architectural characteristics of source-code components. Moreover, evaluating the usefulness of ArchInt to support finding and modifying components has produced positive results. Therefore, the evaluation in this section is whether ArchInt will be suitable to form part of the design of the ideal repository system.

The design of the ideal repository system was identified as consisting of three key elements namely an organizing scheme, a re-factoring tool, and a matching tool. Figure 6.2 illustrates the high-level design of the ideal repository system that was identified in Chapter 3.

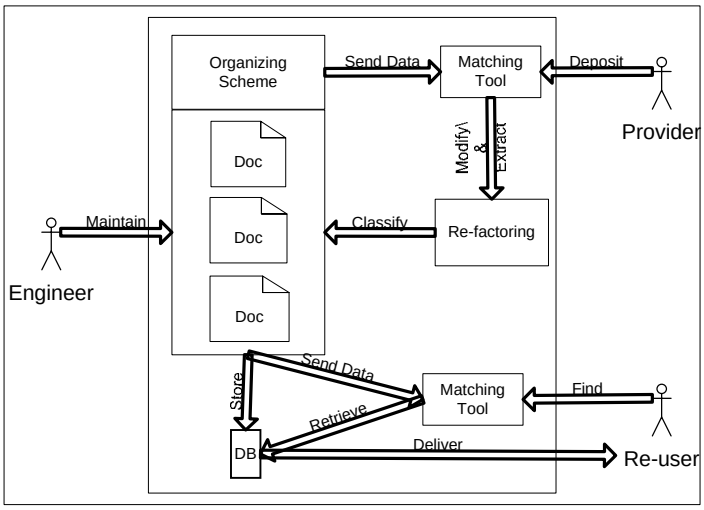


Figure 6.2: Design of Ideal Repository System

The organizing scheme element was part of the design of the ideal repository system to satisfy the requirement of categorizing software components for re-use. The re-factoring tool element was required to support the modification of software components to satisfy the requirement of mapping a provider's view to a re-user's view, as discussed above in

Section 6.4.3. The matching tool was selected to facilitate automatic identification of the deposited software components, and also to support finding components for re-use.

The suitability of ArchInt to form part of the design of the ideal repository system will be evaluated from four dimensions as follows:

1. Organizing scheme: the organizing scheme of the ideal repository system should satisfy the following characteristics:

    a. Extensible: the studies have demonstrated that ArchInt is extensible as it allows for defining architectural types based on existing ones using the pair of tags `<uses_ArchInt>` as demonstrated in the first study of the experimental work.

    b. Flexible: ArchInt has successfully represented a number of different architectural types such as Applet, Eclipse, and “Eclipse XML” architectural types, and the first study in Chapter 5 has demonstrated the applicability of ArchInt to define these selected architectural types.

    c. Automatable: the second study has demonstrated that ArchInt can be used by the *ArchIntParse* tool to automatically identify the architectural type of software components. Hence the study has presented valuable evidence that ArchInt can support automation.

    d. Defines relationships between classifiers: ArchInt defines a “uses” relationship that represent the ontology for linking one ArchInt document with another. This feature is satisfied by ArchInt through the usage of the pair of tags `<uses_ArchInt>`. So, one ArchInt document can be related to another document by the “uses” relationship.

    As ArchInt satisfies the above identified characteristics, it is believed that ArchInt documents can be used as classifiers to organize components in the ideal repository system.

2. Re-factoring components: the study that concerned modifying software components from one architectural type to another has demonstrated the usefulness of using ArchInt to understand what is necessary to perform the modification. Hence, the characteristics defined by ArchInt documents are precise, it is believed that ArchInt could be used to establish the bases for building a re-factoring tool for the ideal repository system.

3. Delivering fully working components: as discussed earlier, this feature was not supported by the prototype ArchInt so has not been investigated further.

4. Support evolution: this feature has not been examined in the studys of evaluating ArchInt. However, it is felt that ArchInt could support evolution as it utilizes tools (e.g. compiler) to perform the necessary checking of software components without applying restrictions to what those tools should be. The `<programming_language>` tag, for instance, identifies the tool to be used to perform the necessary check; whether the tool is a Java compiler or an XML parser depends on the components that are going to be checked. So, if someone wants to extend the functionality of a repository to make it able to check components written in FORTRAN, then ArchInt can be still useful as the compiler can be incorporated into the repository and ArchInt can be used to point to it.

Based on the above evaluation, it seems to be that many of the key features that should be satisfied by the ideal repository system can be derived from ArchInt. Therefore, it is believed that ArchInt could be utilized to form part of the fundamental design of the ideal repository system.

6.6 Limitations of ArchInt

ArchInt was a prototype language needed to demonstrate the feasibility of some of the key aspects of architectural interfaces, and hence was a first step towards a complete specification language for architectural interfaces. This section discusses some of the observed limitations of the ArchInt specification language.

One limitation in ArchInt is that ArchInt documents cannot be generated automatically but requires human involvement in order to identify the characteristics of importance that need to be considered in a document. Architectural types were generated by inspecting the source code of components from the open-source repository that were annotated with names reflecting their architectural types (e.g. “Eclipse”, “Java Applet”), and by examining software systems and trying to identify their internal architectural interfaces. The involvement of people for generating ArchInt descriptions conflicts with the claimed benefits of ArchInt to be a fully automatable approach. At the moment there is no feasible way of identifying architectural characteristics automatically from source code.

Another limitation of the current prototype ArchInt is that the set of tags is not general enough to capture the characteristics of all programming languages. Some of the current tags assume the use of Java, which was the language used for all of the components that

were experimented with. As a result, further generalization of ArchInt would be required to cover a more general set of programming languages. Moreover, the current prototype of ArchInt does not address the issue of how data is exchanged. ArchInt currently utilizes the capability of the compiler of a programming language to check that the sequence of parameters is correct. However, a useful generalization to ArchInt could be to consider precise definitions of how data can be transferred between components and a system to capture other data exchanging mechanisms (e.g. shared memory, streams).

A third limitation in ArchInt is that it does not address the issue of external dependencies that a system should provide to its composing components. For example, if a system requires its composing components to use library X but one of the components to be re-used in the system uses a library Y, then the component may not fit into that system. It is felt that the external dependencies should be part of ArchInt as they identify an additional requirement that a system obliges components to use in order to fit into it.

6.7 Lessons Learned about Architectural Interfaces

Experimenting with architectural interfaces has uncovered some interesting aspects on the overall approach of this research:

- Cohesive software development environment: consider an IDE with the notion of architectural interface integrated into it. A developer can be given help by automatically generating the source-code that represents the architectural framework for the system based on the design at hand. This will help developers to focus only on writing the source code that will provide the functionality for the components of the system to be developed. Moreover, the IDE can advise the developer about the potential components that match the architectural interfaces of their system, so the developer can re-use components without worrying about any architectural mismatches as that could be dealt with automatically by the IDE. Equally, if a software developer needs to apply some modifications to the architecture of the system, then the IDE can reflect the changes on the high-level artefacts (e.g. design, requirement) of the system and presents them to the developer. This kind of support that is provided by the IDE would not have been possible without the support provided by the notion of architectural interface. The usefulness of architectural interface is to maintain the links between the high-level artefacts and the low level implementation.

- Identity for components: a re-user might indicate “I want an Applet component that counts the number of visitors to a webpage”; that would be a more accurate description of the search requirement than “I want a component that counts the number of visitors to a webpage”. Instead of describing only behaviour to search for components it would be useful to know what components are in the first place. The architectural characteristics defined by architectural types can represent identity for components as the characteristics can be used to discriminate one architectural type from another. In the above example, the identity of the component that the re-user was looking for was Applet. A lesson learned about architectural interface is that it can be useful to define identity for components.

- Source-code documentation: most of the source code available in open-source repository lacks documentation that explains the meaning of the written source code and also how to use it. The lack of documentation is an obstacle that could hinder re-using source code. Architectural interfaces represented in ArchInt provide a means of documenting source-code components. A fully implemented ArchInt specification language will generate all the necessary information that re-users need to know in order to re-use components (e.g. how a component can to be registered with a system).

- Formalizing high-level artefacts: the design of a software system is usually an abstract specification of the components of a system and their interaction. System developers are required to map these abstractions into a concrete implementation, and the flexibility they have for doing this is precisely the reason for the difficulty of finding matching re-usable components. Architectural interfaces have been shown to address this issue. If the designer of a software system has provided the description of the architectural types of the system to be built, this will reduce the effort on the implementation stage as developers can use the generated architectural type description to find re-usable components or build their own that conform to the provided architectural type description.

6.8 Summary

This chapter has presented an assessment of the overall achievement of this research. This assessment has uncovered strengths and limitations of architectural interfaces. Overall, the approach generated in this research has addressed issues that have never been considered before, and this evaluation chapter has confirmed that the notion of

architectural interface represents a significant step forward in addressing the problems of re-using software components.

In the final chapter, the work presented in this thesis is drawn together; summarizing the research undertaken and discussing the impact of the results achieved and also provides suggestions for future work.

Chapter 7 - Conclusion

The aim of this research was to address some of the problems that hinder component re-use, and investigate potential solutions to optimise the support that can be provided to components re-users. In the context of this aim, this chapter summarises the important points arising from the earlier chapters, including the evaluation of the research, and suggests areas of future work.

7.1 Overview

Chapter 1 set the scene by establishing the need to have a sophisticated repository system to support re-use. The chapter then discussed the need to characterize software components to enable their classification within a repository system for re-use, and established that component characterization can be achieved through precise descriptions of their interfaces. A major obstacle to re-use was identified in this chapter: re-users who find components that provide the functionality they need could still encounter problems when re-using such components in their system, due to a mismatch between the architectural type of the components and the system.

Chapter 2 described the background work to set the context for this thesis. The chapter described re-use in general terms, software components, CBSD, and software architecture. The chapter identified that a major problem that can hinder component re-use is discovered at integration stage and caused due to architectural mismatch between the system under development and the found component from a repository. It was mentioned also in the chapter that current work in software architecture is not appropriate to tackle the problem of re-use as the architectural characteristics are defined abstractly. The chapter then summed up by describing relation of the work presented in Chapter 2 and the approach of this research.

Chapter 3 described the related work from the perspective of an ideal repository system. A key point was raised in the discussion of different approaches to supporting components re-use, and that was a re-user's searches were imprecise and led to huge number of potential components. The reason for the imprecision was the lack of useful categorization of software components, which in turn was caused by the lack of a precise way of characterising them. The characteristics of the ideal repository system were identified in Chapter 3 to form a basis for analysing the related work. The chapter surveyed the available classification and indexing schemes, re-factoring mechanisms, and repository

systems, and revealed a number of perceived deficiencies including: the lack of precise source code characterization; the lack of support to categorize software components automatically for re-use; and the lack of support to map what is deposited into a repository to what a re-user actually needs (i.e. modifying components). As a result, the development of an approach to capture architectural characteristics from the source code of software components was proposed.

Chapter 4 detailed the important role of component interfaces to help achieve a precise categorization of software components. The chapter established the discrimination between the functional and architectural interfaces of software components as a way to achieve better understanding of component characteristics. Based on the author's initial evaluation, and based on the identified re-use problem, it was decided that the architectural interface of software components had the potential to address some of the significant re-use problems. Re-users could utilize architectural interfaces to focus their search criteria in order to help them find components that not only provide the required functionality, but also fit architecturally into their system. Different characteristics of fit were identified in Chapter 4, and it was noted that the semantics of the identified characteristics were not of importance at this stage of the development of architectural interfaces. Rather, the first concern was to check characteristics in a component's external interface in order to examine the feasibility of the overall idea proposed in this research. The chapter paved the way for the ArchInt specification language that was developed in Chapter 5.

7.2 Results

Chapter 5 presented the formalization of the architectural interface approach by introducing a prototype of a specification language that was called ArchInt. The language was described in this chapter, and then used in studies to evaluate the concepts of architectural interface and to examine its feasibility in addressing the re-use problems.

The first study addressed the sufficiency of the ArchInt specification language for the experimental work that was to evaluate architectural interfaces. This study assessed the applicability of ArchInt to capture the characteristics of different architectural types. A number of architectural types were introduced such as the Applet, Eclipse and MIDlet architectural types. The study identified the different characteristics that components must conform to in order to fit into systems that require any of the three architectural types, and represented them successfully using the ArchInt language. This study demonstrated that ArchInt was sufficient to characterize different architectural types.

The second study examined the applicability of the generated ArchInt descriptions from the first study for finding software components that would fit into a corresponding system. A prototype tool (*ArchIntParse*) was built to setup the experimental work required for evaluating architectural interfaces. The *ArchIntParse* tool performed the matching between software components and the ArchInt descriptions of the three architectural types (i.e. Eclipse, Applet, and MIDlet). The study successfully demonstrated that the architectural type descriptions could be used to identify matching components. The conclusion drawn from experimenting with ArchInt was that architectural interface is a significant approach to support identifying and finding components.

The third study examined the assistance ArchInt could provide when a software system has to be modified by replacing some of its composing components with re-usable components (i.e. components that provide the required functionality and also fit into the system). This study was performed in two iterations. The first iteration considered a system that has its required architectural characteristics rigorously identified, which was the Eclipse system. An Eclipse plug-in was substituted by the *ArchIntParse* tool after applying the necessary modifications. The results successfully indicated that the formalization of architectural types in the prototype of the ArchInt specification language was helpful to understand how to modify a component in order to fit into a system. The second iteration considered a system that had ill-defined architectural characteristics, which was the MVC system. This iteration uncovered the architectural characteristics of the MVC system, even though it was found that the system's architectural characteristics are not fixed as there was nothing to force developers of MVC systems to adopt common interfaces even when working from a common pattern. However, it was felt that ArchInt would be useful if it was defined for every component of an MVC system as it will eliminate the coupling between them by precisely describing the interfaces of software components. So, this study was considered valuable as it identified another area where ArchInt can be useful.

Overall, the studies have demonstrated that the notion of architectural interface is sound and established the necessary ground to derive the building of the ideal repository system. As a result, further research towards building the ideal repository system based on the principle of architectural interface should be undertaken.

7.3 Future work

The optimal goal to support re-use fully is to have the ideal repository system described in Chapter 1. The level of support provided by the ideal repository system would be

centered on addressing the issue of perfect fit of software components. Part of the achievement in this research was to implement a prototype of a tool, named *ArchIntParse*, to perform the necessary check of software components against architectural type descriptions. So an obvious starting point for future work would be to complete the implementation of the tool to fully automate the operation of searching open-source repositories for software components. Moreover, future work could address the implementation of a re-factoring tool that can perform the modification from one architectural type to another using the architectural interface descriptions as that would satisfy one further requirement of the ideal repository system.

Another direction for future research would be to investigate the generation and use of the descriptions of architectural types represented in the ArchInt language as an integral part of a software development process. This integration would help to add some more engineering into the software development process by constraining some implementation details associated with fit. Moreover, it would really help standardize a component's characteristics, and hence encourage potential re-use. The study that involved the MVC system had shed some light on this area, and some preliminary suggestions were made advising that the generation of ArchInt documents should be integrated into a software development process. However, a thorough investigation is still necessary in this area to decide whether integrating architectural type's generation into a software development process will be feasible.

From the perspective of specification language to represent architectural type descriptions, future work should investigate the other dimensions of architectural fit described in Chapter 4. Although some of the characteristics have not been considered explicitly, such as component boundaries, it is felt that these characteristics need to be addressed in more depth as they represent the key aspect addressing whether components can be automatically identified and extracted from a system. The studies carried out in this research assumed that a component's boundaries were Java classes in the case of Java architectural type and JAR files in the case of Eclipse plug-in architectural type. The representation of a component's boundary was not considered in the generated prototype of ArchInt as this characteristic was not germane to the aspects were been evaluated in this research. However, in other programming languages such as the C programming language, the boundaries of a software component may not be obvious. Thus, future work with respect to generating a better specification language that is programming language independent would be useful.

Further work on the architectural interface specification language could also address how data can be exchanged between components and a system, as addressing data-exchange relevant characteristics seems significant to ensure that a component can fit architecturally into a system. Although aspects of data exchange might be handled by the programming language mechanisms, it is felt that identifying data-exchanging mechanisms explicitly would be especially useful in the case of modifying a component from one architectural type to another. So, the data exchanging mechanism required by a system could be mapped to the data exchanging mechanism of a component. Possible future work to generalize a specification language could be to consider a feature for identifying the part of a component that is responsible for exchanging data as being a *block* of source code. For example, in the Java language, Java methods are the blocks in the source code that is responsible of data exchange. Every block might have a name that identifies its address in the source code of a component. Moreover, a block might need to be written in a specific syntax that conforms to the syntax of a certain programming language. The name and syntax of a block are felt to be the two main attributes of a block of source code. A block might exchange data in three ways namely parameters, streams, and shared memory. If a block exchanges data through parameters, then identifying the sequence of parameters seems to be significant as a mismatch in the sequence required by a system to that of a component can cause the wrong *datatype* to be passed to the component. If a block exchanges data through data streams, then identifying the protocol (e.g. HTTP, RPC) of the data transferred could be significant. If a block exchanges data through shared memory, then identifying the name of the shared memory and the *datatype* that can be stored in that memory is significant in avoiding data loss by passing data to a different shared memory than the one a component uses. A block might perform a special action in case of the occurrence of failure. One possible action that could be performed by a block might be to invoke a special component provided by a system, hence the name of the component to be invoked is significant. Another possible action could be to return a special value to a system to indicate that a failure has occurred or the output data of a component is wrong, so a pre-defined set of values could be significant to identify.

An additional feature, that is needed to develop a precise general specification language for representing architectural types, could be the definition of the external dependencies of software components. As discussed in Chapter 4, the external dependencies are the dependencies that must be used by the components in a system, but these dependencies are provided by the system itself for its composing components. So, possible future work

could be to identify the characteristics of the external dependencies of a software component and investigate how the component can use them in order to fit into a system.

Of course, the above proposed features (i.e. data-exchanging mechanisms, external dependencies) for developing a general specification language presented here to give a direction for possible future work, but none of the identified features have been examined thoroughly. Despite that, it is felt that the envisaged new features could be necessary to represent architectural type descriptions in a programming language independent manner.

Another possible piece of work could be to identify the characteristics that may affect the re-usability of software components after a component is found in a repository. It is believed that a re-use process might involve three stages, namely finding, retrieving, and utilizing components, and every stage involves a number of characteristics that are significant from the perspective of re-use. The finding stage involves searching for, identifying, and categorizing components. The retrieving stage involves extracting and delivering components. The utilizing stage involves configuring and using components. The finding stage has been addressed adequately in this research. However, the other two stages have not been addressed in depth. Some discussion was given in this thesis about the characteristics that relate to the stage of retrieving components, and the characteristics of internal dependencies was found to be a characteristic relevant to this stage. However additional work is still needed to capture all possible characteristics in this stage. The third stage (i.e. utilizing stage) has not been addressed in this research, so it could form a starting point for future work. The utilizing stage could identify the characteristics that allow a re-user to understand, for instance, what methods must be invoked in a component to acquire its functionality, or what methods can be invoked to initialize a component and the like.

7.4 Achievement Against the Specified Aim of the Research

As stated in Chapter 1 the aim of this research was to address some of the problems that hinder component re-use, and investigate potential solutions to optimise the support that can be provided to components re-users. Part of the problem that re-users face is the difficulty of finding components that will fit into their system. This research has established the notion of architectural interface as a way to capture precisely the requirements that are necessary for components to fit into a system. Architectural interfaces uncover the architectural characteristics of software components and make those characteristics explicit to re-users (or a repository system), and hence avoid possible

architectural type mismatches between components and the system to be built. The experimental work described in Chapter 5 successfully demonstrated the usefulness of architectural interfaces in overcoming some of the key obstacles hindering component re-use in today's software repository systems. As a result, it is believed that this research has made a significant achievement towards addressing the main aim of this research.

7.5 Closing Remarks

Although software re-use is beneficial to software development, its advantages tend to be hidden by the lack of support offered to re-users. Current repository systems are still far behind achieving the full support that would encourage re-use. Re-users are faced with the problem of finding components that can really fit into their systems. The problems encountered by re-users originate from the lack of precise characterization for software components. Currently, components are loosely characterized based on their functionality, which is neither sufficient nor accurate at the moment.

In the context of this thesis, despite some limitations, the notion of architectural interface has made a positive contribution to the field of software re-use. Certainly architectural interfaces do not solve all the difficulties of software re-use, but represent a step forward in research into the ideal repository system, particularly as architectural interfaces address the characteristics needed by repository system in order to enable automatic identification, organization, and modification of software components. The thesis has established the necessary basis for building a comprehensive repository system that is able to provide the maximum support to encourage software re-use.

References

1. *Visual Thesaurus*. Accessed 02 - 2008, Available from: http://www.visualthesaurus.com.
2. *Random Number Generator*. Accessed 12 - 2008, Available from: http://academic.hws.edu/bio/oldsite/Pages/Random.html.
3. Abramson, N., *Information theory and coding*, McGraw-Hill, 1963.
4. Al-Dallal, J., Sorenson, P. G., Reusing class-based test cases for testing object-oriented framework interface classes. *Journal of Software Maintenance*, 2005. 17(3): p. 169-196.
5. Ali, F.M., Du, W., Towards re-use of object-oriented software design models. *Information & Software Technology*, 2004. 46(8): p. 499-517.
6. Allen, R.,*A Formal Approach to Software Architecture*, PhD Dissertation, Carnegie Mellon University, 1997.
7. Almeida, E., Alvaro, A., Garcia, V., Mascena, J., Burégio, V., Nascimento, L., Lucrédio, D. and Meira, S., *C.R.U.I.S.E - Component Reuse in Software Engineering*. 2007, C.E.S.A.R e-book.
8. Alves, C., Finkelstein, A., Investigating Conflicts in COTS Decision-Making. *Software Engineering and Knowledge Engineering*, 2003. 13(5): p. 473-493.
9. Anderson, G., Anderson, P., *Enterprise JavaBeans Component Architecture: Designing and Coding Enterprise Applications (Java 2 Platform, Enterprise Edition Series)* Prentice-Hall PTR 2002.
10. Arbab, F., Boer, F. and Bonsangue, M.,A Logical Interface Description Language for Components.In *Proceedings of the 4th International Conference on Coordination Languages and Models*.2000.Springer-Verlag.
11. Assman, U., *Invasive Software Composition*, Springer Verlag, 2003.
12. Avgeriou, P., Zdun, U.,Architectural patterns revisited—A pattern language.In *10th European Conference on Pattern Languages of Programs (EuroPlop 2005)*.2005.Universitaetsverlag Konstanz.
13. Balazinska, M., Merlo, E., Dagenais, M., Laguë, B. and Kontogiannis, K., Advanced Clone Analysis to Support Object-Oriented System Refactoring.In *7th Working Conference on Reverse Engineering (WCRE'2000)*.2000.IEEE Computer Society Press,98-107.
14. Bass, L., Buhman, C., Comella-Dorda, S., Long, F., Robert, J., Seacord, R. and Wallnau, K. ,*Market Assessment of Component Based Software Engineering*,2000, Technical Report:2001-TN-007,Carnegie Mellon University/Software Engineering Institute.
15. Bass, L., Clements, P. and Kazman, R., *Software Architecture in Practice*, Addison-Wesley, 2003.
16. Bennett, K.P., Campbell, C., Support Vector Machines: Hype or Hallelujah? *SIGKDD Explorations*, 2000. 2(2): p. 1 - 13.
17. Bergner, K., Rausch, A., Sihling, M. and Vilbig, A.,Adaptation Strategies in Componentware.In *Proceedings of the 2000 Australian Software Engineering Conference*.2000.IEEE Computer Society.
18. Besnard, D., Gacek, C. and Jones, C., *Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective*, Springer, 2005.
19. Booch, G., *Software Components with Ada: Structures, Tools, and Subsystems*, Benjamin/Cummings Publishing Company, 1987.
20. Brereton, P., Budgen, D., Component-based systems: a classification of issues. *IEEE Computer*, 2000. 33(11): p. 54 - 62.

21. Brown, A., Booch, G.,Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors.In *Proceedings of the 7th International Conference on Software Reuse: Methods, Techniques, and Tools*.2002 Springer-Verlag.
22. Brown, A., Short, K.,On Components and Objects: The Foundations of Component-Based Development.In *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*.1997.IEEE Computer Society.
23. Brown, A., Wallnau, K., The Current State of CBSE. *IEEE Software*, 1998. 15(5): p. 37-46.
24. Brown, K.,*Design reverse-engineering and automated design-pattern detection in Smalltalk*,1996, Technical Report:TR-96-07,Department of Computer Science-North Carolina State University.
25. Budgen, D., *Software Design, second edition*, Pearson Addison-Wesley, 2003.
26. Budgen, D., Brereton, P. and Turner, M. and Codifying a Service Architectural Style.In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04) - Volume 01* 2004.IEEE Computer Society.
27. Carney, D., Oberndorf, P., The Commandments of COTS: Still in Search of the Promised Land. *Crosstalk, The Journal of Defense Software Engineering*, 1997. 10(5): p. 25-35.
28. Cechich, A., Piattini, M.,Quantifying COTS component functional adaptation.In *8th International Conference on Software Reuse: Methods,Techniques and Tools (ICSR 2004)*.2004.LNCS(3107).
29. Cechich, A., Piattini, M., Early detection of COTS component functional suitability. *Information and Software Technology*, 2007. 49(2): p. 108-121.
30. Cechich, A., Piattini, M. and Vallecillo, A.,Assessing Component-Based Systems.In *Proceedings of Component-Based Software Quality'2003*.2003.
31. Cechich, A., Requile-Romanczuk, A., Aguirre, J. and Luzuriaga, J.,Trends on COTS Component Identification.In *Proceedings of the Fifth International Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*.2006.IEEE Computer Society.
32. ClayBerg, E., Rubel, D., *Eclipse: Building Commercial-Quality Plug-ins*. 3rd edition, Addison-Wesley, 2004.
33. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J., *Documenting Software Architectures: Views and Beyond*, Addison-Wesley Professional, 2003.
34. Coulange, B., *Software Reuse*, Springer-Verlag, 1997.
35. Crane, S. *Darwin: an Architectural Description Language*. 1997, Accessed 11 - 2008, Available from: http://www.doc.ic.ac.uk/~jsc/research/darwin.html.
36. Crnkovic, I., Larsson, L., *Building Reliable Component-Based Software Systems*, Artech House Publishers, 2002.
37. David, C., Rine,,Success factors for software reuse that are applicable across domains and businesses.In *Proceedings of the 1997 ACM symposium on Applied computing* 1997.ACM,182-186
38. DeLine, R., Avoiding packaging mismatch with flexible packaging. *IEEE Transactions on Software Engineering*, 2001. 27(2): p. 124-143.
39. Depke, R., Engels, G., Thöne, S., Langham, M. and Lütkemeier, B.,Process-Oriented, Consistent Integration of Software Components.In *Proceedings of the*

26th International Computer Software and Applications Conference on Prolonging Software Life: Development and Redevelopment.2002.IEEE Computer Society.
40. Digre, T., Business Object Component Architecture. *IEEE Computer*, 1998. 15(5): p. 60-69.
41. Ducasse, S., Rieger, M. and Golomingi, G.,Tool Support for Refactoring Duplicated OO Code.In *Proceedings of the Workshop on Object-Oriented Technology (ECOOP Workshop Reader)*.1999.Springer-Verlag,177-178.
42. Duller, M., Tamosevicius, R., Alonso, G. and Kossmann, D., XTream: Personal Data Streams.In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.2007.ACM,1088 - 1090.
43. Ezran, M., Morisio, M. and Tully, C., *Practical Software Reuse*, Springer-Verlag, 2002.
44. Fanta, R., Rajlich, V.,Reengineering object-oriented code.In *Proceedings of the International Conference on Software Maintenance*.1998.IEEE Computer Society,238–246.
45. Feller, J., Fitzgerald, B., *Understanding Open Source Software Development*, Addison-Wesley Professional, 2002.
46. Ferenc, R., Siket, I., and Gyimóthy, T.,Extracting Facts from Open Source Software.In *Proceeding of the 20th International Conference of Software Maintenance (ICSM 2004)*.2004.IEEE Computer Society,60-69.
47. Fischer, B., Specification-based browsing of software component libraries. *Automated Software Engineering*, 2000. 7(2): p. 179 – 200.
48. Fitzgerald, J., Larsen, P. G., *Modelling Systems: Practical Tools and Techniques for Software Development*, Cambridge University Press, 1998
49. Fowler, M., *Refactoring: Improving the Design of Existing Programs*, Addison-Wesley, 1999.
50. Frakes, W.B., Pole, T.P., An Empirical Study of Representation Methods for Reusable Software Components. *IEEE Transactions on Software Engineering*, 1994. 20(8): p. 617–630.
51. Gacek, C.,*Detecting Architectural Mismatches During Systems Composition*, PhD thesis, University of Southern California, 1998.
52. Gacek, C., Arief, B., The Many Meanings of Open Source. *IEEE Software*, 2004. 21(1): p. 34-40.
53. Gacek, C., Gamble, C., Mismatch Avoidance in Web Services Software Architectures. *Journal of Universal Computer Science*, 2008. 14(8): p. 1285-1313.
54. Gaedke, M., Rehse, J. and Graef, G.,Supporting Compositional Reuse in Component-Based Web Engineering.In *Proceedings of the 2000 ACM Symposium on Applied Computing*.2000.ACM,927-933.
55. Gamma, E., Helm, R., Johnson,R. and Vlissides, J., *Design Patterns: Elements of Re-usable Object-oriented Software*, Addison-Wesley Professional, 1995.
56. Garlan, D., First international workshop on architectures for software systems workshop summary. *ACM SIGSOFT Software Engineering Notes* 1995. 20(3): p. 84-89.
57. Garlan, D., Allen, A. and Ockerbloom, J., Architectural Mismatch: Why Reuse Is So Hard. *IEEE Software*, 1995. 12(6): p. 17-26.
58. Garlan, D., Allen, R. and Ockerbloom, J.,Architectural mismatch or why it's hard to build systems out of existing parts.In *Proceedings of the 17th international conference on Software engineering*.1995.ACM.

59. Garlan, D., Monroe, R.T. and Wile, T.,ACME: An Architecture Description Interchange Language.In *Proceedings of the 1997 conference of the Centre for Advanced Studies on Collaborative research*.1997.IBM Press,169-183.
60. Garlan, D., Perry, D., Introduction to the Special Issue on Software Architecture. *IEEE Transactions on Software Engineering*, 1995. 21(4): p. 269-274.
61. Garlan, D., Shaw, M.,*Characteristics of Higher-Level Languages for Software Archiecture*,1994, Technical Report:94-TR-23,CMU/SEI.
62. Garlan, D., Shaw, M.,*An Introduction to Software Architecture*,1994, Technical Report:CS-94-166,Carnegie Mellon University.
63. Griss, M., Software reuse: from library to factory. *IBM Systems Journal* 1993. 32(4): p. 548-566.
64. Group, W.C.W. *Web Services Architecture*. 2004, Accessed - 2005, Available from: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/.
65. Guo, J., Luqi.,A survey of software reuse repositories.In *Proceedings of the Seventh IEEE International Conference and Workshopon on the Engineering of Computer Based Systems*.2000.IEEE Computer Society,92-100.
66. Heineman, G., Councill, W., *Component-Based Software Engineering: Putting the Pieces Together*, Addison Wesley, 2001.
67. Hemer, D., Lindsay, P.,Specification-based retrieval strategies for module reuse.In *Proceedings 2001 Australian Software Engineering Conference*.2001.IEEE Computer Society.
68. Henninger, H., An Evolutionary Approach to Constructing Effective Software Reuse Repositories. *ACM Transactions on Software Engineering and Methodology*, 1997. 6(2): p. 111-140.
69. Heuzeroth, D., Holl, T., Högström, G., Löwe, W.,Automatic Design Pattern Detection.In *Proceedings of the 11th International Workshop on Program Comprehension co-located with 25th International Conference on Software Engineering*.2003.IEEE,94.
70. Hondt, K., Lucas, C. and Steyaert, P.,Reuse Contracts as Component Interface Descriptions.In *Proceedings of the Workshops on Object-Oriented Technology*.1997.Springer-Verlag.
71. Hopkins, J., Component Primer. *Communications of the ACM* 2000. 43(10): p. 27 - 30.
72. Hunter, E., *Classification Made Simple*, Ashgate, 2002.
73. IEEE, *IEEE Standard Glossary of Software Engineering Terminology: ANSI/IEEE Standard 610-12-1990*. 1990, IEEE Press.
74. ISO/IEC. *Information Technology - Metadata Registries (MDR), parts 1-6, 2nd edition. ISO/IEC 11179*. 2005, Accessed 03 - 2008, Available from: http://metadata-standards.org/11179/.
75. ISO/IEC. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. IEEE Std 1471*. 2000, Accessed 11 - 2008, Available from: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=00875998.
76. Java2s. *A Java Repository for Developers*. Accessed 11 - 2007, Available from: http://www.java2s.com/.
77. Jefferson, N.,*Dependable Compositions: A Formal Approach*, PhD Thesis, Newcastle University, 2006.
78. Jeng, J., Cheng, B., Specification Matching for Software Reuse: A Foundation. *ACM SIGSOFT Software Engineering Notes*, 1995. 20(SI): p. 97 - 105.
79. Jones, A.,The Maturing of Software Architecture.In *Software Engineering Symposium*.1993.Software Engineering Institute.

80. Joos, R., Software Reuse at Motorola. *IEEE Computer Society Press*, 1994. 11(5): p. 42-47.
81. Kain, J., Components: The Basics: Enabling an Application or System to be the Sum of its Parts ,*Object Magazin*, April 1996,
82. Kawaguchi, S., Garg, P. K., Matsushita, M. and Inoue, M.,MUDABlue: An Automatic Categorization System for Open Source Repositories.In *Proceedings of the 11th Asia-Pacific Software Engineering Conference(APSEC)*.2004.IEEE Computer Society,184-193.
83. Kienle, H.M., Müller, H.A.,A lightweight taxonomy to characterize component-based systems.In *Proceedings — ICCBSS 2007: Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems*.2007.IEEE,193–201.
84. Knudsen, J., Nourie, D. *Wireless Development Tutorial Part I*. 2006 Accessed 12 - 2007, Available from: http://developers.sun.com/mobility/midp/articles/wtoolkit/.
85. Kohonen, T., *The Self-organizing Map*. 3 edition. Information Sciences. Heidelberg, Springer-Verlag, 2000.
86. Krueger, C., Software Reuse. *ACM Computing Surveys*, 1992. 24(2): p. 131-183.
87. Landauer, T., Foltz, P. W. and Laham, D., Introduction to Latent Semantic Analysis. *Discourse Processes*, 1998. 25(2-3): p. 259–284.
88. Lau, K., Wang, Z., Software Component Models. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, 2007. 33(10): p. 709-724.
89. Lawrie, T., Arief, B., and Gacek, C., *Interdisciplinary Insights on Open Source*, in *Proceedings of the Open Source Software Development Workshop*. 2002, Newcastle upon Tyne.
90. Lee, J.H., Kim, J. and Shin, G. S.,Facilitating Reuse of Software Components using Repository Technology.In *Proceedings of the 10th Asia-Pacific Software Engineering Conference*.2003.IEEE Computer Society,136.
91. Lee, P.A., Anderson, T., *Fault Tolerance: Principles and Practice (Second Revised Edition)*, Springer-Verlag, 1990.
92. Lee, S.Y., Gwon, O. C., Sin, G. S., COBALT Assembler : A Case Tool for Supporting EJB Component Assembly based on Architecture. *Journal of Software Engineering*, 2002 5(2): p. 32-38.
93. Leite, J., Yu, Y., Liu, L., Yu, E. and Mylopoulos, J., Quality-Based Software Reuse. *Lecture Notes in Computer Science*, 2005. 3520: p. 535-550.
94. Li, G., Zhang, L., Li, Y., Xie, B. and Shao, W., Shortening retrieval sequences in browsing-based component retrieval using information entropy. *Journal of Systems and Software*, 2006. 79(2): p. 216 - 230.
95. Li, Q., Bjørnson, F., Conradi, R. and Kampenes, V., An empirical study of variations in COTS-based software development processes in the Norwegian IT industry. *Empirical Software Engineering*, 2006. 11(3): p. 433 - 461.
96. Lin, M.Y.J., Amor, R. and Tempero, E.,A Java Reuse Repository for Eclipse using LSI.In *Proceedings of the Australian Software Engineering Conference (ASWEC'06)* 2006.IEEE Computer Society 351-360.
97. Lowy, J. *C# 2.0 Code Re-factoring*. 2004, Accessed 08 - 2006, Available from: http://www.devx.com/codemag/Article/20143.
98. Luckham, D.,*Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Ordering of Events*,1996, Technical Report:CSL-TR-96-705,Stanford University.

99. Lucredio, D., Prado, A. and Almeida, E.,A Survey on Software Components Search and Retrieval.In *Proceedings of the 30th EUROMICRO Conference*.2004.IEEE Computer Society.
100. Lycett, M., Paul, R.,Component-Based Development: Dealing with Non-Functional Aspects of Architecture.In *ECOOP '98 Workshop on Component-Oriented Programming* 1998.Springer-Verlag.
101. Maarek, Y.S., Berry, D.M. and Kaiser, G.E., An Information Retrieval Approach for Automatically Constructing Software Libraries. *IEEE Transactions on Software Engineering*, 1991. 17(8): p. 800-813.
102. Mahmood, S., Lai, R. and Kim, Y, Survey of component-based software development. *IET Software*, 2007. 1(2): p. 57-66.
103. McCSmith, J., Stotts, D.,*Case Studies in Automated Design Pattern Detection in C++ Code using SPQR*,2005, Technical Report:TR05-013,The University of North Carolina.
104. McIlroy, M.,Mass Produced Software Components.In *Software Engineering: Report on a Conference by the NATO Science Committee*.1968.NATO Science Affairs Division,138-150.
105. Medvidovic, N., Taylor, R., A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 2000. 26(1): p. 70-93.
106. Mehta, N., Medvidovic, N. and Phadke, S.,Towards a taxonomy of software connectors.In *Proceedings of the 22nd international conference on Software engineering*.2000.ACM.
107. Mens, T., Tourwé, T., A Survey of Software Refactoring. *IEEE Transactions on Software Engineering*, 2004. 30(2): p. 126-139.
108. Meyer, B., *The Grand Challenge of Trusted Components*, in *25th International Conference on Software Engineering (ICSE'03)*. 2003, IEEE Computer Society.
109. Meyer, B., On To Components. *Computer*, 1999. 32(1): p. 139-140.
110. Meyer, B., On Formalism in Specifications. *IEEE Software*, 1985. 2(1): p. 6-26.
111. Meyer, B., Applying "Design by Contract". *Computer*, 1992. 25(10): p. 40-51.
112. Mili, H., Mili, F., and Mili, A., Reusing Software: Issues and Research Directions. *IEEE Transactions on Software Engineering*, 1995. 21(6): p. 528-562.
113. Mili, H., Valtchev, P., Di-Sciullo,A.M. and Gabrini, P.,Automating the Indexing and Retrieval of Reusable Software Components.In *Proceedings of Applications of Natural Language to Information Systems* 2001.GI,75-86.
114. Morisio, M., Torchiano, M.,Definition and classification of COTS: a proposal.In *Proceedings of the First International Conference on COTS-Based Software Systems*.2002.Springer-Verlag.
115. Ning, J.,Component-Based Software Engineering (CBSE).In *Proceedings of the 5th International Symposium on Assessment of Software Tools (SAST '97)*.1997.IEEE Computer Society.
116. Oates, B., *Researching Information Systems and Computing*, Sage Publications Ltd 2005.
117. Oberleitner, J., Gschwind, T. and Jazayeri, M.,The Vienna Component Framework enabling composition across component models.In *Proceedings of the 25th International Conference on Software Engineering*.2003.IEEE Computer Society.
118. Ostertag, E., Hendler,J., Prieto Díaz,R. and Braun, C., Computing Similarity in a Reuse Library System: An AI-Based Approach. *ACM Transactions on Software Engineering and Methodology*, 1992. 1(3): p. 205- 228.

119. Podgurski, A., Pierce, L., Retrieving reusable software by sampling behavior. *ACM Transactions of Software Engineering and Methodology*, 1993. 2(3): p. 286 - 303.
120. Pohthong, A., Budgen, D., Reuse strategies in software development: An empirical study. *Information & Software Technology*, 2001. 43(9): p. 561-575.
121. Poulin, J.S., Yglesias, K.P., Experiences with a Faceted Classification Scheme in a Large Reusable Software Library (RSL).In *Seventeenth Annual International Computer Software and Applications Conference*.1993.IEEE,90-99.
122. Prieto-Diaz, R., Freeman, P., Classifying Software for Reusability. *IEEE Software*, 1987. 4(1): p. 6-16.
123. Ravichandran, T., Rothenberger, M., Software Reuse Strategies and Component Markets. *Communications of the ACM*, 2003. 46(8): p. 109-114.
124. Raymond, E., *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* O'Reilly Media, 1999.
125. Rehman, M., Jabeen, F., Bertolino, A. and Polini, A., Testing software components for integration: a survey of issues and techniques. *Software Testing, Verification & Reliability* 2007. 17(2): p. 95 - 133.
126. Reid, S.,BS 7925-2: The Software Component Testing Standard.In *Proceedings of the The First Asia-Pacific Conference on Quality Software (APAQS'00)*.2000.IEEE Computer Society.
127. Richard, B., Krugle Code Search Stats or How to Write a Press Release, *Linux Magazine*, 2008, http://www.linux-mag.com/id/5916
128. Rodrigues, N., Barbosa, L. *On the Specification of a Component Repository*. 2003, Accessed 05 - 2006, Available from: http://www.iist.unu.edu/newrh/III/1/docs/techreports/report284/paper2.pdf.
129. Sametinger, J., *Software Engineering with Reusable Components*, Springer-Verlag, 1997.
130. Schmidt, D., Stal, M., Rohnert, H., Buschmann, F., *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. 1st edition. Software Design Patterns. Vol. 2, Wiley, 2000.
131. Buschmann, F., Meunier, R., Rohnert, H., Sommerland, P. and Stal, M., *Pattern-Oriented Software Architecture: A System of Patterns*, Wiley, 1996.
132. Shaw, M.,Architectural issues in software reuse: it's not just the functionality, it's the packaging.In *Proceedings of the 1995 Symposium on Software reusability*.1995.ACM.
133. Shaw, M., Clements, P.,A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems.In *Proceedings of the 21st International Computer Software and Applications Conference*.1997.IEEE Computer Society.
134. Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M. and Zelesnik, G., Abstractions for Software Architecture and Tools to Support Them. *IEEE Transactions on Software Engineering*, 1995. 21(4): p. 314 - 335.
135. Shaw, M., Garlan, D., *Software Architecture: Perspectives on an Emerging Discipline*, Prentice Hall, 1996.
136. Siegel, J., *CORBA 3 - Fundamentals and Programming*, John Wiley & Sons, 2000.
137. Sommervill, I., Integrated Requirements Engineering: a tutorial. *IEEE Software*, 2005. 22(1): p. 16-23.
138. Sommerville, I., *Software Engineering 8th edition*, Addison-Wesley, 2006.
139. Spivey, J., *The Z Notation: A Reference Manual*, Prentice Hall, 1991.
140. Swanson, J.E., Samadzadeh, M. H.,Re-usable software catalog interface.In *Proceeding 92 ACM SIGAPP Symposium Application Computing SAC*.1992.ACM,1076-1082.

141. Szyperski, C., Gruntz, D. and Murer, M., *Component Software - Beyond Object-Oriented Programming*. 2nd edition, Addison-Wesley(ACM Press), 2002
142. Tangsripairoj, S., Samadzadeh, M. H.,Organizing and Visualizing Software Repositories Using the Growing Hierarchical Self-Organizing Map.In *Proceedings of the 2005 ACM Symposium on Applied Computing*.2005.ACM SAC Special Track on Software Engineering,1539 – 1545.
143. Tao, Y., Papadias, D.,Adaptive Index Structures.In *Proceedings of the 28th International Conference on Very Large Data Bases Conference (VLDB)*.2002.Morgan Kaufmann,418-429.
144. Traas, V., Hillegersberg, J., The software component market on the internet current status and conditions for growth. *ACM SIGSOFT Software Engineering Notes*, 2000. 25(1): p. 114.
145. Ugurel, S., Krovetz, R., Giles, C. L., Pennock, D. and Glover, E.,What's the Code? Automatic Classification of Source Code Archives.In *Proceedings of the ACM SIGKDD Conference on Knowledge Discovery and Data Mining* 2002.ACM,632 – 638.
146. Vitharana, P., Risks and challenges of component-based software development. *Communications of the ACM* 2003. 46(8): p. 67-72.
147. Wallnau, C.,*A technology for predictable assembly from certifiable components*,2003, Technical Report:2003-TR-009,Carnegie Mellon University/Software Engineering Institute.
148. Washizaki, H., Fukazawa, Y.,Automated Extract Component Refactoring.In *Proceeding of the 4th International Conference on eXtreme Programming and Agile Processes in Software Engineering (XP2003)*, .2003.Springer-Verlag,328-330.
149. Weber, S., *The success of open source*, Harvard University Press 2004.
150. Wirth, N., IEEE Annals of the History of Computing. *IEEE Annals of the History of Computing*, 2008. 30(3): p. 32-39.
151. Wood, D., Christel, M. and Stevens, S.,A Multimedia Approach to Requirements Capture and Modeling.In *Proceedings of the First International Conference on Requirements Engineering*.1994.IEEE Computer Society Press, 53-56.
152. Yacoub, S., Ammar, H. and Mili, A. *Characterizing a Software Component*. 1999, Accessed 07 - 2004, Available from: www.sei.cmu.edu
153. Yakimovitch, D., Bieman, J. and Basili, V.,Software architecture classification for estimating the cost of COTS integration.In *Proceedings of the 21st international conference on Software engineering* 1999.IEEE Computer Society Press.
154. Yang, H., Ward, M., *Successful Evolution of Software Systems*, Artech House Publishers, 2003.
155. Ye, H., Lo, B.W.N., Towards a Self-structuring Software Library. *IEE Proceedings – Software*, 2001. 148(2): p. 45-55.
156. Yglesias, P.,IBM's reuse programs: knowledge management and software reuse.In *Proceedings. Fifth International Conference on Software Reuse*.1998.IEEE Computer Society,156-165.
157. Yunwen, Y.,*Supporting Component-Based Software Development with Active Component Repository Systems*, PhD thesis,Boulder, University of Colorado, 2001.
158. Zaremski, A.M., Wing, J.M., Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 1995. 4(2): p. 146–170.

Appendix A- ArchInt representations of Architectural Types

This appendix provides the complete representation of the three architectural types discussed earlier in ArchInt prototype specification language.

1. The full ArchInt representation of the Applet Architectural Type:

```
<ArchInt>
  <name>Applet</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
  <must__have>
      <Method>
        <name>setStub</name>
        <param>
          <string>AppletStub</string>
        </param>
        <returnType>void</returnType>
      </Method>

      <Method>
        <name>getAppletContext</name>
         <returnType>AppletContext</returnType>
      </Method>

      <Method>
        <name>init</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>start</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>stop</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>destroy</name>
         <returnType>void</returnType>
      </Method>
    </must__have>
</ArchInt>
```

2. The full ArchInt representation of the Eclipse Architectural Type:

```
<ArchInt>
  <name>Eclipse</name>
  <uses_ArchInt>
            <name>Java Class</name>
            <name>Eclipse XML</name>
  </uses_ArchInt>
  <must__have>
      <Field>
            <name>PLUGIN_PREFERENCE_SCOPE</name>
            <dataType>String</dataType>
      </Field>

      <Field>
            <name>PREFERENCES_DEFAULT_OVERRIDE_BASE_NAME</name>
            <dataType>String</dataType>
      </Field>

      <Field>
            <name>PREFERENCES_DEFAULT_OVERRIDE_FILE_NAME</name>
            <dataType>String</dataType>
       </Field>

      <Method>
            <name>start</name>
            <param>
                  <string>org.osgi.framework.BundleContext</string>
            </param>
            <exception>Exception</exception>
      </Method>

      <Method>
            <name>getBundle</name>
            <returnType>org.osgi.framework.Bundle</returnType>
      </Method>

      <Method>
            <name>stop</name>
            <param>
                  <string>org.osgi.framework.BundleContext</string>
            </param>
            <exception>Exception</exception>
      </Method>

      <Method>
            <name>openStream</name>
            <param>
                  <string>org.eclipse.core.runtime.IPath</string>
                  <string>boolean</string>
            </param>
            <returnType>java.io.InputStream</returnType>
            <exception>java.io.IOException</exception>
      </Method>

      <Method>
            <name>openStream</name>
            <param>
                  <string>org.eclipse.core.runtime.IPath</string>
            </param>
            <returnType>java.io.InputStream</returnType>
            <exception>java.io.IOException</exception>
      </Method>

      <Method>
            <name>getPluginPreferences</name>
            <returnType>org.eclipse.core.runtime.Preferences
            </returnType>
      </Method>

      <Method>
            <name>getStateLocation</name>
            <returnType>org.eclipse.core.runtime.IPath</returnType>
            <exception>java.lang.IllegalStateException</exception>
      </Method>
```

Cont. Eclipse Architectural Type

```
<Method>
            <name>getLog</name>
            <returnType>org.eclipse.core.runtime.ILog</returnType>
      </Method>

      <File>
            <name>plugin</name>
            <type>XML</type>
      </File>

    </must__have>
</ArchInt>
```

3. The full ArchInt representation of the MIDlet Architectural Type:

```
<ArchInt>
  <name>MidLet</name>
  <uses_ArchInt>
      <name>Java Class</name>
  </uses_ArchInt>
    <must__have>
      <Method>
        <name>startApp</name>
      <returnType>void</returnType>
      </Method>

      <Method>
        <name>pauseApp</name>
      <returnType>void</returnType>
      </Method>

      <Method>
        <name>destroyApp</name>
      <returnType>void</returnType>
      </Method>

      <Method>
        <name>init</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>start</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>stop</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>destroy</name>
      <param>
          <string>boolean</string>
        </param>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>notifyDestroyed</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>notifyPaused</name>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>resumeRequest</name>
         <returnType>void</returnType>
      </Method>
```

Cont. MIDlet Architectural Type

```
<Method>
         <name>getAppProperty</name>
        <param>
           <string>String</string>
         </param>
         <returnType>String</returnType>
       </Method>

       <Method>
         <name>checkPermission</name>
        <param>
           <string>String</string>
         </param>
         <returnType>int</returnType>
       </Method>

       <Method>
         <name>platformRequest</name>
        <param>
           <string>String</string>
         </param>
       <returnType>boolean</returnType>
       <exception>ConnectionNotFoundException</exception>
       </Method>

       <File>
             <type>JAD</type>
       <File>
     </must__have>
</ArchInt>
```

4. The full ArchInt representation of the "Eclipse XML" architectural type:

```
<ArchInt>
  <name>Eclipse XML</name>
  <uses_ArchInt>
            <name>Eclipse XML</name>
  </uses_ArchInt>
  <must_have>
        <Field>
                <name>id</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>name</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>version</name>
                <dataType>String</dataType>
        </Field>

        <Field>
                <name>provider-name</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>class</name>
                <dataType>String</dataType>
        </Field>
        <Field>
                <name>extension-point</name>
                <dataType>String</dataType>
        </Field>
  </must_have>
</ArchInt>
```

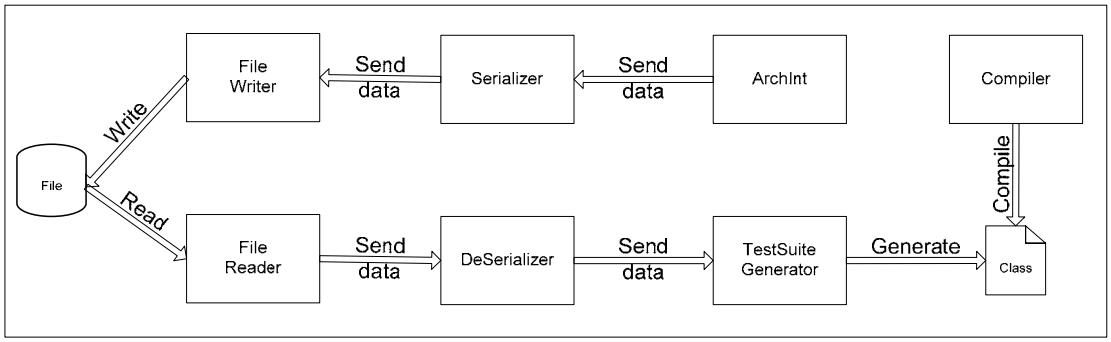
5. The full ArchInt representation of the Serializer architectural type from the *ArchIntParse* tool:

```
<ArchInt>
  <name>Serializer</name>
  <uses_ArchInt>
            <name>Java Class</name>
  </uses_ArchInt>

    <must__have>
      <Method>
        <name>ArchIntProcessor</name>
      </Method>

      <Method>
        <name>setInput</name>
        <param>
          <string>DeSerializer</string>
        </param>
         <returnType>void</returnType>
      </Method>

      <Method>
        <name>getOutput</name>
        <returnType>ArchInt</returnType>
      </Method>


      <Method>
        <name>getMethodList</name>
        <returnType>ArrayList</returnType>
      </Method>

      <Method>
        <name>getFieldList</name>
        <returnType>ArrayList</returnType>
      </Method>

      <Method>
        <name>getContextList</name>
        <returnType>ArrayList</returnType>
      </Method>

    </must__have>
</ArchInt>
```

Appendix B - Source code of the ArchIntParse Tool

This appendix list the design and the source code of the *ArchIntParse* tool

1. **The architecture of the *ArchIntParse* tool:**



2. **the source code of the Serializer class:**

```
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;
import java.util.*;

public class Serializer
{

        MyFileWriter myFileWriter;
        public void serialize(IArchInt archint)
        {

                XStream xStream = new XStream(new DomDriver());
                String xml = xStream.toXML(archint);
                if(myFileWriter != null)
                {
                    myFileWriter.writeToFile(xml);
                }
                else
                {
                    System.out.println("Please specify a file Writer
Object!");
                }
        }

        public void setFileWriter(MyFileWriter myFileWriter)
        {
         this.myFileWriter = myFileWriter;

        }

}
```

3. **the source code of the Deserializer class:**

```
import com.thoughtworks.xstream.XStream;
import com.thoughtworks.xstream.io.xml.DomDriver;

public class DeSerializer
{

    XStream xStream;
    SimpleClassLoader sc = new SimpleClassLoader();
   // String xml = "";
    ArchInt myArchInt;
    public DeSerializer()
```

```
    {
        xStream = new XStream(new DomDriver());
        //xml = xmlEx.getOutput();
       xStream.setClassLoader(sc.getClass().getClassLoader());
    }

    public void setInput(XMLExtractorFromFile xmlEx)
    {
       String xml = xmlEx.getOutput();
       myArchInt = (ArchInt)xStream.fromXML(xml);
    // Object o = xStream.fromXML(xml);
    }

    public ArchInt getOutput()
    {
        return myArchInt;
    }

}
```

4. the source code of the ArchIntProcessor class:

```
import java.util.*;

public class ArchIntProcessor
{

     ArchInt myArchInt;
     ArrayList methodList = new ArrayList();
     ArrayList fieldList = new ArrayList();

     public void setInput(DeSerializer dsz)
     {
        myArchInt = dsz.getOutput();
        processor();
     }

    public ArchInt getOutput()
    {
        return myArchInt;
    }

      public void processor()
        {

                ArrayList list =
myArchInt.getExternal().getMustHave(); // external;
                ArrayList list2 =
myArchInt.getInternal().getRequired(); // internal;


                // traverse the External elements;
                for(int i = 0; i<list.size(); i++)
                {

                    String temp = getMarker((Marker)list.get(i));

                    if(temp.equals("Field"))
                    {
                         Field f = (Field)list.get(i);
                         fieldList.add(f);
                    }


                    if(temp.equals("Method"))
                    {
                         Method m = (Method)list.get(i);
                         methodList.add(m);
```

```
                    }
                }

                // traverse the Internal elements;
                for(int i = 0; i<list2.size(); i++)
                {
                    String temp = getMarker((Marker)list2.get(i));

                    if(temp.equals("Dependancy"))
                    {
                         Dependancy d = (Dependancy)list2.get(i);
                    }
                }


        }


        // extract the type of objects from the arraylist based on
the value returned by the getMarker() method;
        private String getMarker(Marker marker)
        {

            String temp = new String();
            temp = marker.returnMarker();
            return temp;

        }

         public ArrayList getMethodList()
        {
            return methodList;
        }

        public ArrayList getFieldList()
        {
            return fieldList;

        }
}
```

5. the source code of the TestGenerator class:

```
import java.util.*;

public class TestGenerator
{
    String str = "";
    String[] parama = new String[]{"a","b","c","d","e"};
    String className = "";
    String pkg = "";
    int parmaCounter = 0;
    int parmaDecCounter = 0;
    int rtnTypeCounter = 0;
    int fieldCounter = 0;

    ArrayList methodsArrayList = new ArrayList();
    ArrayList parametersDeclarationsArray = new ArrayList();
    ArrayList fieldsArrayList = new ArrayList();

    public void generateTest(String className, String pkg) // split
into methods;
    {
        this.className = className;
        this.pkg = pkg;

        String inst = "inst"+className;
```

```
        if(pkg != null) str = "package " + pkg + ";\n";
        str = str + "public class TestSuite\n" + "{\n";
        str = str +  "    "+ className + " " + inst+ " = new " +
className +"();\n";
        str = str +  "    public void test()\n" + "    {\n";

         printMethod(methodsArrayList, inst);
         printField( fieldsArrayList , inst);

          str = str +"    }\n" + "}";

            System.out.println(str);
            MyFileWriter2 fwt = new MyFileWriter2();
            fwt.writeToFile(str);
    }

    public String getStr()
    {
     return str;
    }

    private void printMethod(ArrayList method, String inst)
    {
        extractParams(method);

        for (int j = 0; j<method.size(); j++)
        {

           if(((Method)method.get(j)).getException() == null)
           {
               /////////
               if(((Method)method.get(j)).getReturnType() != null &&
!((Method)method.get(j)).getReturnType().equals("void")) // if there
is a return type;
               {
                    str = str + "       " +
((Method)method.get(j)).getReturnType() + " rtn" + rtnTypeCounter++
+" = " ;

                    if(((Method)method.get(j)).getScope() != null &&
((Method)method.get(j)).getScope().equals("static"))
                    {
                        str = str  +
className+"."+((Method)method.get(j)).getName()+"(";
                    }
                    else
                    {
                        str = str  +
inst+"."+((Method)method.get(j)).getName()+"(";
                    }
               }

               else
               {
                    str = str  + "
"+inst+"."+((Method)method.get(j)).getName()+"("; // for the spacing
if return type is void;
               }

               setParameters(((Method)method.get(j)).getParama());
               str = str +")"+";\n";
           }
           else
           {
               exceptions((Method)method.get(j), inst);
           }
        }
    }
```

```
    private void exceptions(Method method , String inst)
    {
        str = str + "       try\n" + "        {\n";

         if(method.getReturnType() != null &&
!method.getReturnType().equals("void"))
               {
                 str = str + "          " + method.getReturnType() +
" rtn" + rtnTypeCounter++ + " = " ;
                  str = str  + inst+"."+method.getName()+"(";
                }
         else
         {
            str = str  + "
"+inst+"."+method.getName()+"(";
        }

             setParameters(method.getParama());
             str = str +")"+";\n";

        str = str + "        }\n" + "
catch("+method.getException()+" e){}\n";

    }

    private void setParameters(String[] parameters)
    {

        if(parameters != null)
        {
            for(int i = 0 ; i < parameters.length ; i++)
                {

                str = str + "a"+ parmaCounter++;//parama[i];
                if(i< parameters.length-1) str = str + ","; //e.g.
(a,b,c)

                }
        }
    }

    private void parmaDeclaration(ArrayList parameters)
    {

      if(parameters.size() != 0)
      {
        for(int i = 0 ; i < parameters.size() ; i++)
            {

               if(parameters.get(i).equals("int"))
               {
                   str = str  + "       "+  parameters.get(i) + " " +
"a"+ parmaDecCounter++ + " = 0;\n";
                }

                else if(parameters.get(i).equals("Double"))
               {
                   str = str  + "       "+  parameters.get(i) + " " +
"a"+ parmaDecCounter++ + " = 0.0;\n";
                }

                 else if(parameters.get(i).equals("char"))
               {
                   str = str  + "       "+  parameters.get(i) + " " +
"a"+ parmaDecCounter++ + " = '';\n";
                }
                 else if(parameters.get(i).equals("boolean"))
               {
```

```
                    str = str  + "       "+  parameters.get(i) + " " +
"a"+ parmaDecCounter++ + " = false;\n";
                }
                else
               {
                    str = str  + "       "+  parameters.get(i) + " " +
"a"+ parmaDecCounter++ + " = null;\n";
                }

            }

        }

    }

    public void addMethod(Method method)
    {
        methodsArrayList.add(method);

    }

    public void setMethodList(ArrayList mList)
    {
        methodsArrayList = mList;
    }

    public ArrayList getMethodList()
    {
        return methodsArrayList;
    }



   private void extractParams(ArrayList method)
   {
       for(int i = 0 ; i < method.size(); i++)
       {
           String[] parmas = ((Method)method.get(i)).getParama();

          if(parmas != null)
          {
                for(int k = 0 ; k < parmas.length ; k++)
                {
                    parametersDeclarationsArray.add(parmas[k]);
                }
            }

        }

        parmaDeclaration(parametersDeclarationsArray);

    }


    public void setFieldList(ArrayList fList)
    {
        fieldsArrayList = fList;
    }

    public ArrayList getFieldList()
    {
        return fieldsArrayList;
    }

     public void printField(ArrayList fieldsArrayList , String inst)
     {
         for(int i = 0 ; i < fieldsArrayList.size() ; i++)
         {
```

```
            str = str + "       "+
((Field)fieldsArrayList.get(i)).getType() + " field" + fieldCounter++
+ " = ";
            if(((Field)fieldsArrayList.get(i)).getScope() != null)
             {

if(((Field)fieldsArrayList.get(i)).getScope().equals("static"))
                {
                    str = str + className + "." +
((Field)fieldsArrayList.get(i)).getName() + ";\n ";
                }
             }
             else
                {
                    str = str + inst + "." +
((Field)fieldsArrayList.get(i)).getName() + ";\n ";
                }
        }

     }

}
```