ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

THESIS PRESENTED TO
ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR
THE DEGREE OF DOCTOR OF PHILOSOPHY
Ph.D

BY
Hamdan MSHEIK

SOFTWARE CONSTRUCTION BY COMPOSITION OF COMPONENTS

MONTREAL, MAY 19 2010

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Alain Abran, Thesis Supervisor
Département de génie logiciel à l'École de technologie supérieure

M. Sylvie Nadeau, President of the Board of Examiners
Département de génie mécanique à l'École de technologie supérieure

M. Christopher Fuhrman, member of the Board of Examiners
Département de génie logiciel à l'École de technologie supérieure

M. Witold Suryn, member of the Board of Examiners
Département de génie logiciel à l'École de technologie supérieure

M. Chadi El-Zammar, external examiner
Averna Technologies

THESIS WAS PRESENTED AND DEFENDED

BEFORE A BOARD OF EXAMINERS AND A PUBLIC

MARCH 30, 2010

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

REMERCIEMENTS

Avant tout, je remercie mon directeur de recherche Mr. Alain Abran pour ses suggestions, ses disponibilités, son assistance financière et surtout pour l'inestimable qualité de son encadrement accompagné de son extraordinaire attitude positive. Également, je remercie mes professeurs pour les discussions que j'ai entamées avec eux et pour les suggestions et commentaires qu'ils ont émis à propos de mon thème de recherche. Aussi, je remercie l'École de technologie supérieure représentée par ses professeurs, administrateurs et techniciens. Grâce à leur travail d'équipe ce travail a été mené à terme.

Mes remerciements s'adressent aussi à mes parents, ma femme, mon fils, mon frère, mes sœurs, mes oncles, mes tantes et mes amis pour tout l'encouragement et le support moral qu'ils m'ont offert tout au long du cheminement de ma vie d'étudiant-chercheur. En reconnaissance de leur support et encouragement je leur dédie ce travail.

CONSTRUCTION DE LOGICILELS PAR COMPOSITION DE COMPOSANTS

Hamdan MSHEIK

RÉSUMÉ

Dans une ère où les technologies de l'information subissent une évolution continuelle, la complexité du logiciel ainsi que l'évolution des exigences augmentent à un rythme accéléré. Le développement de logiciels suivant des approches traditionnelles mène à la construction d'applications caractérisées par une taille monolithique, une structure difficile à réutiliser et un coût de construction de plus en plus élevé. Pour remédier à ces problèmes, le développement logiciel à base de composantes a été proposé et a apporté des bénéfices notamment grâce aux avantages que la technologie de composantes de logiciel offre au développement. Cependant, il reste un nombre de problèmes non résolus qui affectent la technologie de composantes de logiciel et ainsi le développement d'applications à base de composantes.

Le but de ce projet de recherche est d'améliorer la construction de logiciel par composition de composantes. Les composantes traditionnelles utilisées dans la construction de logiciel souffrent du syndrome de gonflement attribué aux membres non-utilisés qui varient dépendamment du contexte et du domaine d'affaires. De plus, les composantes traditionnelles peuvent aussi souffrir du problème d'amalgamation chaotique d'éléments de code. Ces deux problèmes peuvent être attribués en partie au manque de composantes de logiciel modulaire et qui typiquement mènent les constructeurs de logiciels soit à la réécriture complète de certaines composantes de logiciel, ou soit à la modification et adaptation du code existant appartenant à ces composantes. En plus des deux problèmes mentionnés précédemment, les composantes de logiciel peuvent aussi souffrir du problème de versions non conformes causé par l'usage de diverses versions de la même composante.

Pour faire face à ces problèmes, une nouvelle approche est proposée dans le cadre de ce projet de recherche. Cette approche se veut à la fois simple et fluide. Cette approche est basée sur l'utilisation de composantes ayant une taille atomique ou ayant une modularité améliorée. Cette approche va contribuer à l'amélioration de la construction de logiciels à bas e de composantes de logiciel en remédiant à certains des problèmes dont cette approche souffre actuellement. La portée de ce projet de recherche se limite à la phase de construction du cycle de vie de développement logiciel.

Le but générique de ce projet de recherche est :

- D'améliorer la construction de logiciels par composition de composantes à travers une approche qui transporte la construction de logiciels d'une approche traditionnelle basée principalement sur l'écriture et l'amalgamation de code vers une approche basée de plus en plus sur la composition de composantes de logiciel.

Les objectifs spécifiques de ce projet de recherche sont :

- De proposer et de spécifier un modèle de composantes de logiciel remédiant aux problèmes de composantes traditionnelles tels que présentés dans ce projet de recherche.
- De fournir une implémentation servant comme référence pour le modèle proposé de composantes.
- De concevoir une méthode de mesure pour quantifier le nombre de membres non-désirés de composantes de logiciel.
- De fournir un prototype d'outil automatique qui sert à mesurer le nombre non-désirés de composantes de logiciel.
- De proposer un mécanisme pour la détection de versions non-compatibles de composantes de logiciel.
- De fournir un prototype d'un outil automatique qui sert à détecter les versions non-compatibles de composantes de logiciel.

L'approche présentée dans cette thèse est partiellement inspirée de l'observation de la façon de développer des produits dans les domaines de génie autre que le génie logiciel. Cette approche représente un pas en avant dans l'évolution de la construction de logiciels à base de composantes grâce à sa façon de composer des composantes de logiciel qui est à la fois simple et fluide. L'usage et l'utilisation de cette approche reduit le besoin de réajuster ou d'adapter des composantes existantes, comme tel est le cas traditionnellement. Le constructeur de logiciels se sert de plus en plus de la composition selective de composantes à modularité améliorée et qui convient aux besoins d'affaires particuliers d'une application. De plus, cette approche fournira un nombre d'avantages secondaires qui peuvent être exploités pour faire de tests de granularité plus fine et pour l'amélioration de processus de mesure de composantes logicielles qui ultimement améliorent le processus de développement de logiciel en sa globalité.

SOFTWARE CONSTUCTION BY COMPOSITION OF COMPONENTS

Hamdan MSHEIK

ABSTRACT

In the continuously evolving era of software technological advances, software complexity and requirements change grow at increasing paces. Developing software using traditional approaches to meet the growing demand for functionalities and computation, in particular for large scale software, produces software applications which are characterized as being monolithic, difficult to reuse and costly to develop. To address those issues, component-based software development has emerged among other approaches and has already produced a noteworthy positive impact. Nevertheless, component-based software development still suffers from a number of drawbacks and limitations.

The aim of this research project is to improve software construction by composition of components. Traditionally, reusable components may exhibit a bloating syndrome caused by bundled but unused set of members which vary according to application contexts and business domains. Furthermore, reusable components may suffer from chaotic amalgamation of code elements. Component members' bloating and a chaotic amalgamation of code elements limitations can be partially attributed to the lack of modular components. Typically, software constructors rewrite from scratch newer software components (even though many code parts exist), retrofit or customize existing components to satisfy applications' requirements. In addition to these two limitations, software components suffer also from version mismatches due to the use of different versions of the same component.

To address the aforementioned limitations a new approach is proposed in this research work. Our approach is based on composition of atomic or enhanced modularity components. This new approach will contribute to the improvement of component-based software construction by alleviating some of the limitations facing it. It The focus of our research targets particularly the construction phase of the software development lifecycle.

The main generic goal of this research project is:

- To improve software construction based on components composition by providing an approach which shifts and promotes software construction from a traditional construction approach based heavily on code writing and amalgamation to an approach relying increasingly on components composition.

The specific objectives of this research are:

- To propose and specify a software component model which provides remedies for some of the limitations facing software construction by components composition.
- To provide a reference implementation for this component model.
- To design a measurement method to measure components' unwanted members.
- To provide a prototype tool to measure components' unwanted members.

- To propose a component versioning mechanism.
- To provide a prototype tool to detect component versions mismatches.

The approach presented in this thesis is partly derived from the observation of product development processes implemented in traditional engineering disciplines. It can be argued that this approach represents a step forward in the evolution of software construction based on components composition for it provides a simple and fluid components composition approach. The application and use of this composition approach reduces the need to retrofit and customize existing components as has been done traditionally. The software constructor relies more and more on selective composition of enhanced modularity components which suit particular application requirements. Moreover, this approach leads to additional secondary benefits which can be exploited in the use of fine-grained testing and in conducting various component measurements that ultimately benefit the software construction process as a whole.

TABLE OF CONTENT

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

LIST OF ABREVIATIONS AND ACRONYMES

.NET	Microsoft component framework
AC	Atomic Component
ANTLR	Another Tool for Language Recognition
AOC	Atomic and Optimal Composition
AOP	Aspect Oriented Programming
API	Application Programming Interface
AST	Abstract Syntax Tree
BCEL	Byte Code Engineering Lanugage
BML	Bean Markup Language
CBSD	Component Based Software Development
CBSE	Component Based Software Engineering
CCM	CORBA Component Model
CIDL	Component Implementation Definition Language
COBOL	COmmon Business-Oriented Language
COM	Component Object Model
COP	Component Oriented Programming
CORBA	Common Object Request Broker Architecture
COTS	Commercial off-the-Shelf
CSCM	Compositional Structured Component Model
CUMM	Components Unused Member Measurement
CoML	Component Markup Language
CoMet	Component Measurement
CoPL	Component Plan Language
DCOM	COM Distribute Object Model
DEC	Digital Equipment Corporation
DLL	Dynamic Link Library
DSOM	Distributed System Object Model
EJB	Enterprise Java Bean
EXE	Executable file under Microsoft operating system families

GIOP	General Inter-Orb Protocol
GUI	Graphical User Interface
GUID	Globally Unique Identifiers
IBM	International Business Machines
IC	Integrated Circuit
IDE	Integrated Development Environment
IDL	Interface Definition Language
IEEE	Institute of Electrical and Electronics Engineers
IIOP	Internet Inter-Orb Protocol
IP	Internet Protocol
J2EE	Java 2 Enterprise Edition
JDBC	Java Database Connectivity
JNDI	Java Naming and Directory Interface
JavaBeans	Component complying with Sun Java beans component model
JavaCC	Java Compiler Compiler
JVM	Java Virtual Machine
MIDL	Microsoft Interface Definition Language
MS RPC	Microsoft Remote Procedure Call
NATO	North Atlantic Treaty Organization
NT	Microsoft windows operating system for networked computations
NetOLE	Network OLE
OLE	Object Linking and Embedding
OMA	Object Management Architecture
OMG	Object Management Group
OO	Object Oriented
OOP	Object Oriented Programming
ORB	Object Request Broker
OSGi	Open Services Gateway Initiative
Piccola	PI Calculus based Compositional Language
RAD	Rapid Application Development

RMI	Remote Method Invocation
RPC	Remote Procedure Call
SOM	System Object Model
SWEBOK	Software Engineering Body of Knowledge
TCP/IP	Transmission Control Protocol/Internet Protocol
UNIX	Operation System developed at Bell Labs
XML	eXtensible Markup language

LIST OF SYMBOLS AND MESUREMEANT UNITS

Ac	attribute per component
Fc	function per component

INTRODUCTION

Currently, software engineering development methodologies and practices use complex processes to satisfy the growing demands on software. In particular, the software reuse potential has not yet adequately met the expectations (Humphrey, 2001; Mili *et al.*, 1999a; Mili *et al.*, 1999b; Mili, Mili and Mili, 1995; Shiva and Shala, 2007). Therefore, a considerable amount of effort is still required for the adaptation and customization of existing software code elements to fit particular application contexts and requirements. The challenges of growing and ever changing software requirements, increased software complexity are fuelling the expectations for reuse especially in the context of heterogeneous computation platforms (Brooks, 1987; Fraser and Mancl, 2008; Hughes, 1990; Mcdirmid, Flatt and Wilson, 2001). Component technology and component-based software construction provide a promising remedy for those challenges. In this respect, components are perceived to be modular, loosely-coupled and compositional (Coker and Hayes, 1997 Monterey, California). Reusable and modular software components are expected to play a key role in improving software construction processes and in reducing software construction time to market.

Component technology has been promoted as an innovative means to tackle the issues of software reuse and modularity. Interestingly, components bear inherently the characteristics of reuse and modularity. Ignoring or inappropriately addressing software reuse and modularity in the software construction process often leads to monolithic software applications which are less flexible, increasingly complex, difficult to reuse and costly to develop, evolve and maintain (Li, 1999; Vanhelsuwe, 1997).

Software development has been evolving in the past 30 to 40 years over several software development paradigms: the procedural, the functional and the object-oriented paradigms. Lately, the component-based software engineering paradigm has gained much attention. Software components represent a major step in the evolution of computing technology. In this respect, the component-based software engineering paradigm has been considered by

Peter Maurer as a computing revolution on a par with those of stored programs and programming languages (Maurer, 2003). It is worth noting that the idea behind software components is not new; it first appeared in a NATO (North Atlantic Treaty Organization) conference on software engineering in the late 1960's (McIlroy, 1968).

Software components have been defined in many different ways (Eclipse Foundation, 2008; Heinemann and Council, 2001; Jacobson, 1998; Maurer, 2003; Szyperski, 1998). The common characteristics among these definitions are:

1. Components have interfaces and interface implementations used in interconnecting with other components;
2. Component behaviors are almost independent;
3. Components can be packaged in a binary form so that they can be treated as black boxes.

A definition given by Heineman (Heinemann and Council, 2001) goes a little further and requires a software component to comply with a component model where the component model defines the interaction between components. Also, the component model has to define a composition standard.

In an effort to reduce the complexity of software applications and increase software reuse, several component models have emerged. At their inception, many of those models put the emphasis on particular aspects of the software application. Several industrial component models have been developed to address the issues of software application complexity, software reuse enhancement, software component distribution (Ongg, 1997), application interoperability (Exton, Watkins and Thompson, 1997) and rapid GUI (Graphical User Interface) construction. While the existing component models represent significant technological improvements, they still suffer from several limitations which are listed below and discussed in details in Chapter 3:

1. Unwanted components' functionalities and members in the context of a particular application;

2. Chaotic and amalgamated composition of code chunks in the process of software construction;
3. Non-optimal component reuse and non-optimal component modularity often attributed to the impact of the software constructor's subjectivity on the software construction process;
4. Component versions mismatch occurring as a result of the evolution of software components.

The research work presented in this thesis aims to tackle the limitations listed above. The first three limitations have close relationships and thus will be tackled collectively by the same proposed solution. More precisely, this solution relies on a software composition approach based on the AOC (Atomic and Optimal Component) model proposed in this thesis.

This proposed component model is designed to provide a flexible mechanism to remedy the limitation of useful but unwanted components' members. This AOC model allows selective composition of only the required component members as per application requirements and according to variable business domain contexts.

As for the limitation of chaotic and amalgamated composition of code chunks, the design of the AOC (Atomic Composition) model promotes operational (functions or methods) reusable code chunks to full-fledged components, thereby simplifying and making the customization, adaptation, reuse and composition of reusable code chunks easier and straightforward.

To get rid of the impact of component constructor subjectivity, newly designed and constructed reusable software components need to comply with the AOC model, since this model allows compliant components to have enhanced modularity and therefore to exhibit better reuse as will be illustrated in Chapter 6.

For the fourth limitation listed above concerning component version mismatch, a component versioning scheme is proposed. This versioning scheme is based on the de-facto versioning standards used in software development.

Following this introduction, chapter 1 provides a literature review on software components, their architectures, advantages and limitations. In chapter 2, the idea of software composition is presented with an emphasis on the different aspects of software composition, component construction and component-based software construction. In chapter 3, the methodology followed as well as the research objectives of this research project are presented. Additionally, the software component limitations tackled in this research project are illustrated and discussed. Chapter 4 introduces a method to measure unwanted component members. In addition, the prototype of a tool used to perform this measurement is also briefly presented. Chapter 5 presents a preliminary component model as a solution approach. In chapter 6, the AOC model which is an enhancement of the preliminary model is proposed as the solution approach. Chapter 7 provides details on how to construct components as well as software applications complying with the component model presented in chapter 6. In chapter 8, the proposition of a component versioning scheme to prevent component versions mismatches is presented, as well as the prototype tool provided to detect component versions mismatches. In chapter 9, the construction of the prototype compiler used by the component model proposed in this research project is presented. Afterwards, a case study of the AOC model is presented in chapter 10. This is followed in chapter 11 by a presentation of the verification and validation criteria used to assess the solution approach and tools proposed and developed in this research project. Finally, in chapter 12 the conclusion is presented.

CHAPTER 1

STATE OF THE ART

1.1 Software construction paradigms

Software engineering as a process has been evolving over several programming paradigms among which are:

1. The functional paradigm, which is also known as a declarative language driven paradigm, dates back to the 1950s (Hudak, 1989; Ryder, Soffa and Burnett, 2005). This paradigm is characterized by its use of higher-order functions and by its powerful functional composition mechanism. This compositon mechanism allows functions to be passed as arguments to other functions as well as to be returned as values to variables (Hughes, 1990). It is claimed that constructing software using a functional language is quicker, succinct and closely follows mathematical notations (Hudak, 1989). From the perspective of software components, the compositional mechanism offered by the functional paradigm is attractive among other things because of the considerable composition and reuse potential it offers.
2. The procedural paradigm, which is also known as the imperative paradigm, started with the development of COBOL in the 1950s (Li and Abraham, 2002). It is based on the grouping of code instructions into procedures or routines and grouping variables into data structures upon which those procedures or routines act and realize certain computational tasks or actions in an algorithmic-like manner. Currently, the use of this computational paradigm is secondary due to the better features offered by more recent and competing paradigms. It can be argued that this paradigm played a helpful and transitional role to the object oriented paradigm.
3. The OOP (Object Oriented Programming) paradigm has been around since the 1960s but received serious attention only at the beginning of the 1980s (Ryder, Soffa and Burnett, 2005). This paradigm is now a mainstream technology and has been adopted widely in industry. By and large, the object-oriented paradaigm has provided significant advantages

over other paradigms. It brought the concepts of object abstractions, data encapsulation, polymorphism and inheritance, which have considerably contributed to software reuse and reduced software complexity, with all their derived advantages.

4. The component-oriented paradigm was originally developed at the beginning of the 1990s (Durham, 2001) to provide better software reuse and object exchange. Although the object-oriented paradigm has subtle differences with the component paradigm the later has built upon and inherited many of the interesting characteristics and advantages offered by the object-oriented paradigm. However, it is not necessary for components to be constructed using an OOP language.

1.2 History of components

The interest in the evolution of the component technology is relatively new considering the modern history of the information technology sector, which dates back to the second half of the twentieth century. Historically, the term software component has been first referred to by McIlroy (McIlroy, 1968) in his paper on "Mass Produced Software Components". In 1985, Ledbetter & Cox (Ledbetter and Cox, 1985) used the term software IC (Integrated Circuit) to designate what is currently know as a software component. It is not until the beginning of the 1990s that the term software component has been widely used with the emergence of both the IBM (International Business Machines) SOM/DSOM (System Object Model/Distributed System Object Model) and Microsoft COM/DCOM (Component Object Model/ Distribute Component Object Model) component technologies (Rosemary, 1998; Sherlock and Cronin, 2000). It is noteworthy to mention that the surging interest in components came at a time when the object-oriented paradigm was accumulating significant successes. Obviously, the component paradigm is not a replacement of the OO (Object Oriented) paradigm, but can be thought of as a complementary extension of it.

1.3 Component definition

Before delving into the details of this research work, it is necessary to elaborate briefly on the concepts relevant to the research activities described in the subsequent chapters. Since the major issues tackled in this research project concern software component, it is intresting to define what a component is. As illustrated in Table 1.1, software component definitions have proliferated, yet there is currently no consensus on a specific single definition (Hull, Nicholl and Bi, 2001).

Generally speaking, a software component refers to a multitude of software replaceable parts and units. Those parts can be classes, packages of classes, units of methods and even complete standalone executable applications with public connection points for interactions.

Table 1.1 Definitions of a component in the software engineering literature

Component Definitions
Paul Harmon (Hull, Nicholl and Bi, 2001) defines a software component as a module having interfaces that it publishes.
Jacobson (Jacobson) defines a component as a physical replaceable part of a system that realizes and conforms to a set of interfaces.
Philippe Krutchen (Hull, Nicholl and Bi, 2001) defines a component as a replaceable part of a system used in the context of a resilient architecture to fulfill a well-defined function. Such a component is characterized by its large independence and non-triviality, in addition to conforming and providing a set of physical interface realizations.
Szyperski (Szyperski, 1998) defines a software component as a composition unit of contractually specified interfaces having only explicit dependencies. Such a component should be ready for composition by third-parties.

Table 1.1 Definitions of a component in the software engineering literature (continued)

Microsoft (Microsoft Corporation, 1995) defines a component as "a reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort".
Kozaczynski (Kozaczynski and Booch, 1998) defines a business component as the representation of the implementation of a business process or concept. It is a reusable element, composed of the entire software artifacts that are necessary for the description, implementation and deployment of the concept in a larger business system.
Heinemann (Heinemann and Council, 2001) provides the following definition: "A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard. A component model defines specific interaction and composition standards. A component model implementation is the dedicated set of executable software elements required to support the execution of components that conform to the model." (Heineman et Council, 2001, p 7) A software component infrastructure is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

The definition given in (Heinemann and Council, 2001) is more interesting than the others, since it emphasizes the relationship between a software component and both its component model and the software component infrastructure. In other words, this definition provides a more formal definition of what a software component is compared to the other definitions.

It can be observed that the above definitions hold even for traditional OO classes. However, those definitions have missed to distinguish OO classes from components and they do not address the necessary key services on which the component is dependent to accomplish its

computational tasks. Key services usually are are not useful on their own, but exist to provide services for other components such as logging, caching, transaction management and security. For this reason, it is interesting to provide yet another component defining a component to be the set of code elements providing a public principal computational service. Additionally, the component may depend on a variety of key services so that it fulfills its computational task. The rules governing its existence, interaction and relationships with other components as well as the services it depends on constitute its component model. Furthermore, it may or may not have other outside dependencies.

In spite of the differences reflected in the definitions of a software component listed in Table 1.1, those definitions still provide a common understanding of what is a software component and what are its main characteristics. In particular, a component must:

- Have interfaces (connection points) and interfaces' implementations for interaction;
- Provide a nearly independent behavior;
- Have a standard model to which it complies;
- Have an infrastructure necessary to permit its composition, deployment and proper functioning;
- Provide or easily create its executable format so that it can be dealt with as a black box.

1.4 Component versus object

For a better understanding of components, it is important to distinguish them from 'objects'. Components and objects have a number of similarities, but also have subtle differences. Components have similarities with OO classes (Szyperski, 1998) in that a component consists of a collection of static code elements, i.e. both of them are blueprints. OO classes are among the basic ingredients of components. Since components' popularity happened right after the OOP wave, it might be perceived that components can be constructed out of classes only. In reality, the notion of components extends beyond the OO perspective of classes. Components can aggregate a variety of code elements other than OO classes. In particular, components can be constructed out of procedural code elements such as COBOL

(COmmon Business-Oriented Language) modules, UNIX shell scripts, or C operations and subroutines, etc. Like classes, components can also own members accessible through designated interfaces.

The common ground between components and objects is that they both play the role of containers. The difference between them is that components can be represented by any type of code elements, whereas objects are only instances of OOP languages classes. In conclusion, the comparison between components and objects is an awkward one; it makes more sense to compare components with classes since they both represent similar concepts, i.e., code elements. For clarity and illustration purposes a summary of the differences and similarities between components and objects is presented in Table 1.2.

Table 1.2 Differences and similarities between components and objects

Components	**Objects**
Components are static, similar to classes.	Objects are dynamic runtime elements and represent class instances.
Components can have attributes similar to class attributes.	Objects can hold runtime dynamic attributes of their instantiated classes.
Components have interfaces.	Objects can own runtime interface instances for their instantiated classes.
Unless they are constructed from classes, components are simpler than objects: Non-object-oriented components do not have polymorphic and inheritance mechanisms as do objects.	Objects are advanced entities when compared to instances of non-object-oriented code.

Table 1.2 Differences and similarities between components and objects (continued)

Components tend to be generally easier to compose than objects. They clearly define their provided and required interfaces, so that they can be plugged into a variety of contexts with no prior knowledge of those contexts; they only fulfill the contract defined by their interfaces.	Objects are compositional but tend to have higher dependency on their context, a particularity hindering their composition. For instance, when an object class is changed, this might trigger a recompilation of several parts of the application in order to accommodate the change.
Components can be expressed in any type of code elements related to various programming paradigms such as classes, COBOL modules, Unix shell scripts, etc.	Objects only exist with in the context of the OO paradigm.
Components tend to be deployed in packages such as J2EE packa ged ear files and .Net DDL.	Objects are difficult to use in executable form, since they generally have outer dependencies without which they can not be used. In other words, they are useless outside of their executable environment.
Components tend to be have coarse and large grain sizes	Objects tend to have smaller grains compared to components

1.5 Advantages of software components

Software engineering is not yet mature enough and well established as an engineering domain. The evolution rhythm of this engineering domain is quite rapid, taking into account the relatively short timeframe since its inception in the last half of the 20th century. The catalysis of software engineering paradigms has culminated and yielded the component-

based paradigm. This new paradigm stands out as one of the essential tenets sustaining the software engineering domain towards maturity. The expectations are that components will:

1. Increase software reuse.
2. Reduce development efforts and time to market costs.
3. Promote and encourage the use of standards.
4. Provide better interoperability and distribution with the emergence of mediators such as ORB (Object Request Brokers).

Will components stand up to the expectations? While it is recognized that they have had a positive impact, measuring the depth of the impact is not an easy task due to a multivariety of factors.

1.6 Component models

The software component technology has been used by both researchers and practitioners to increase the reusability, productivity and quality of software applications, and to decrease their time to market, maintenance and customization costs (Rashid, 2001).

There exist several component models. The next section presents brief overviews of CCM (CORBA Component Model), COM/DCOM/.NET, and Sun EJB (Enterprise Java Bean) and Java beans component models.

1.7 CORBA: a component reference model

OMG (Object Management Group),a consortium of several vendors and researchers from all over the world, has adopted the CORBA (Common Request Broker Architecture) model. CORBA has emerged as a cross-platform, language-neutral architecture and component reference model for distributed applications (Tari and Buhkres, 2001). This architecture offers a standard middleware solution for easier communication, flexible distributed application development and interoperability between distributed heterogeneous applications

running on different operating systems, written in different programming languages and located in different geographical regions (OMG, 2001; 2002).

CORBA's earlier versions 1.0 and 1.1 were centered on the ORB agent, the IDL (Interface Definition Language) and several basic services (Trading, Naming, etc). Version 2.0 has provided and standardized the interoperability between the different vendors of ORBs through the use of the GIOP (General Inter-ORB Protocol) and its specialized and most popular variation, the IIOP (Internet Inter-ORB Protocol) protocol. CORBA's version 3, has tighter integration with Java and the Internet (Siegel, 2001; Tari and Buhkres, 2001).

1.7.1 Object management group (OMG)

The OMG group effort has led to the concept of a reference object model and a reference architecture known as Object Management Architecture (OMA). The purpose of the reference model is to define the way objects are distributed across the different platforms, whereas OMA describes the interaction between those objects.

1.7.2 Reference object model

The CORBA object model is based on an abstract model similar to the traditional object models. The CORBA object model provides a well-defined and encapsulating interface that separates and isolates the clients from the supplier of services. The CORBA object model is divided into two parts: semantic part (client) and implementation part (Siegel, 2001; Tari and Buhkres, 2001).

Object semantics: The semantic part is responsible for describing the meaning of the object as perceived by the client (i.e. object creation and identity, requests, operations, signatures, etc.).

Object implementation: The implementation part describes object implementation concepts related to the behavior realization of objects such as the execution and activation engines,

and state update of the system. The implementation part is divided into two models: the execution model and the construction model (Siegel, 2001; Tari and Buhkres, 2001):

1. The execution model defines the way services are performed. Upon a client request for a service, an operation is to be executed by an *execution engine* (abstract machine) whose responsibility is to define a dynamic context for the method execution. The execution of a method is referred to as method activation.
2. The construction model describes the way services are defined. It is responsible for realizing a request behavior through mechanisms that define the object state, object creation, methods execution selection infrastructure, and association of appropriate methods with newly created objects. In short, the object implementation is the definition of the concept that enables the creation of an object and makes it available to collaborate to the fulfillment of a client service.

1.7.3 OMA

The OMA is the highest-level specification (Mowbray and Ruh, 1997). It defines an abstract object model architecture (Ruh, Herron and Klinker, 1999) that describes object concepts and terminology leaving the implementation details to implementers. The OMA architecture defines the components, interfaces and protocols of the architecture. Also, it divides objects into four categories (OMG 2001):

1. CORBA Services such as Naming, Transaction and Life Cycle facilities;
2. CORBA horizontal Facilities such as Time and Internationalization facilities;
3. CORBA domain (vertical) facilities which can be related among others to the financial and health care domains;
4. Application Objects such as developer-provided objects.

The ORB (Object Request Broker) is a cornerstone element in the OMA architecture. It is in charge of transparently performing any required communication (unidirectional or bidirectional) between the various object categories. The ORB has the responsibility of finding the implementation of any requested object, transparently activating the object if it is

not active, dispatching the request to the object and returning the outcome when available to the client. OMA ORB specification has been enhanced in order to provide Inter-ORB interoperability between the different ORBs. ORB interoperability is specified by the GIOP and IIOP, a specialized version using TCP/IP protocol and networks.

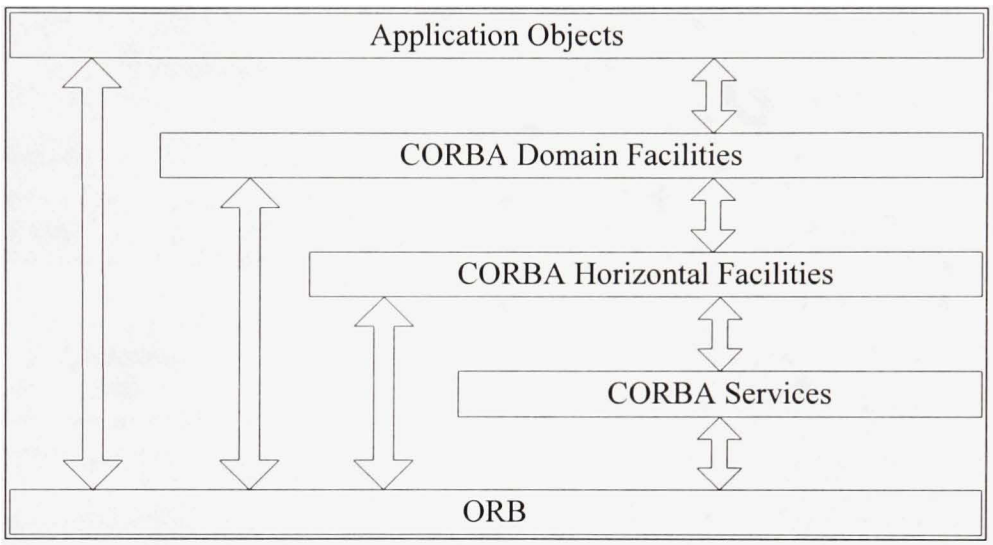

Figure 1.1 OMA architecture.
(Tari and Buhkres, 2001, p. 44, F 2.2)

1.7.4 CCM

CCM is built to overcome the limitations observed in the CORBA standard prior to version 3. Many extensions have been added to CCM. As illustrated in Figure 1.2, the concept of container hosting servant object (component implementations) has been adopted to provide a standard mechanism to interact with components. Furthermore, CORBA components have been equipped with new features for their definition, assembly and deployment. CORBA components can be either of two types: basic or extended (OMG, 2006).

Basic components: CORBA Basic components do not support inheritance. Furthermore, they neither provide interfaces nor use them. Additionally, they cannot generate events.

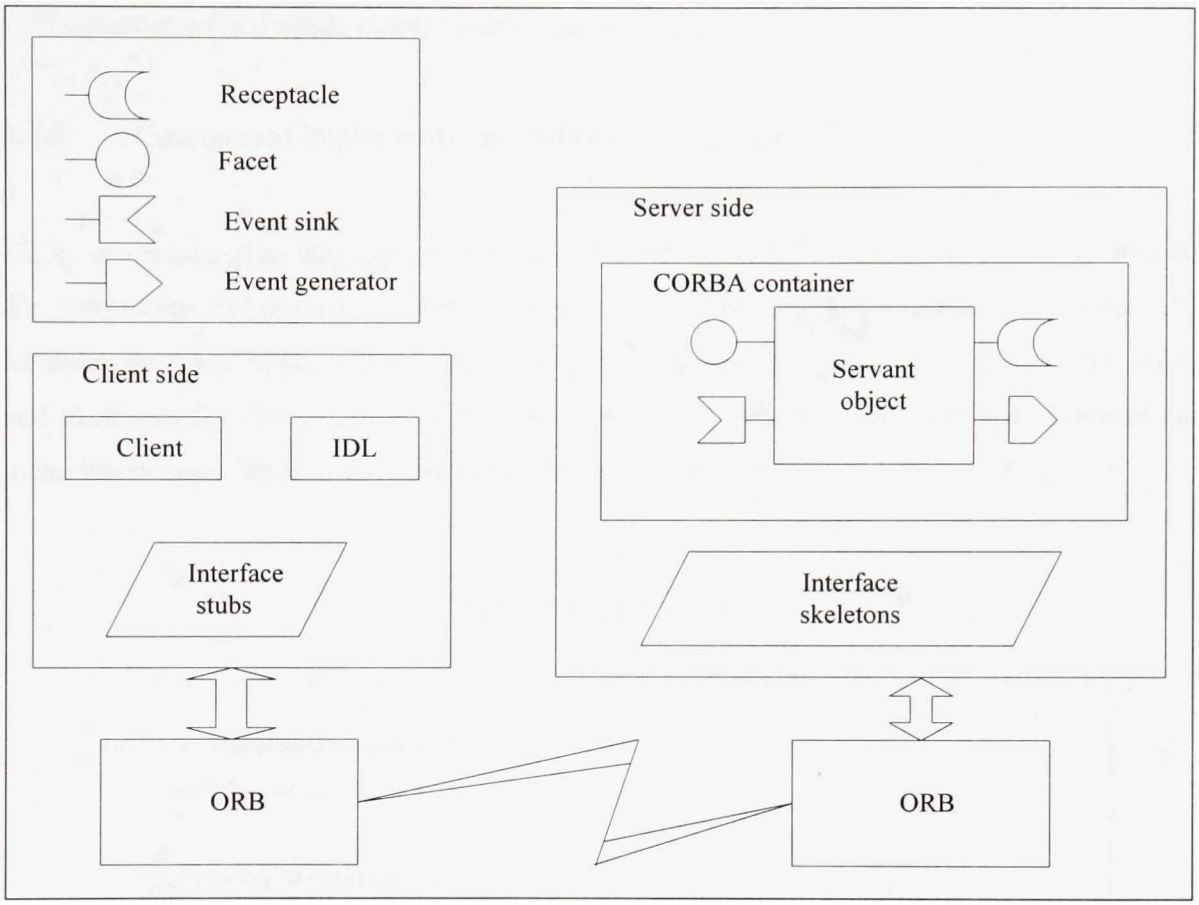

Figure 1.2 CORBA Component Model.

Extended components: Extended components are enriched with features allowing to extend other components and to exihbit "provide" and "use" interfaces. In addition, extended components can generate and receive events - an indispensable mechanism to support asynchronous computation. Interaction with an extended component is done via its ports which are of four types:

1. Facets are partial views of the component functionalities which can be used by clients;
2. Receptacles are connection points through which the component receives references to other components on which it invokes operations;
3. Event sinks are connection points through which the component receives asynchronously references to other components;

4. Event generators are event channels through which the component receives events for consumption and sends events to other components.

1.7.5 Component implementation definition language (CIDL)

CIDL is a declarative language extension of CORBA version 2 IDL which adds constructs for component definitions, implementations and port binding as illustrated in Table 1.3. Modules enclosed inside CIDL scripts are passed through a compiler to generate the stubs and skeletons for the component. The developer then uses the stubs and the skeletons to make interactions between the client and servant components.

Table 1.3 CIDL script

```
Module BankManagement {
  Typedef  string name;

  Interface Withdrawal {
     void withdraw();
  // definition of the account details
  };
    Component Account {
       provides Withdrawal withdrawal;
  }
};
```

1.7.6 CORBA as a middleware

As a middleware technology, the objective of CORBA is to provide a better technology for the development of distributed applications. The developer's focus will be on the business logic of his application (business logic) rather than on the details of application networking

and application distribution (Ruh, Herron and Klinker, 1999). Like other middleware technologies, CORBA strives to support the software engineering characteristics of simplicity, scalability, portability, adaptability, maturity, interoperability and reliability.

1.8 COM/DCOM/.NET

COM, DCOM and .NET components are Microsoft technologies that define a component model and architecture for local and distributed applications. The origin of COM can be traced back to OLE 1.0 (Object Linking and Embedding), a Microsoft technology used to facilitate data interchange between different applications. The COM model has been jointly developed by Microsoft and DEC (Digital Equipment Corporation) (Sherlock and Cronin, 2000). In Windows 3.x, Microsoft used OLE technology to enable components or objects living locally within the same system to communicate and share data. In 1994, DEC and Microsoft collaborated to produce the specification for the distributed OLE version which has known several name changes: NetOLE (network OLE), DCOM and finally simply COM.

1.8.1 COM architecture

According to the Microsoft specifications (Microsoft Corporation 1995), COM is an object-based programming model designed to allow applications to be interoperable in a collaborative, easy manner independent of COM suppliers. The interoperability between applications using COM technology is provided through a set of definitions and implementation mechanisms. Microsoft's COM specifications define a component as:

> "A reusable piece of software in binary form that can be plugged into other components from other vendors with relatively little effort" (Microsoft Corporation, 1995, p. 11)

COM foundations and concepts: The COM object model is based on a client/server architectural model as shown in Figure 1.3 adapted from Microsoft's COM specifications. The client object requests a service from the implementation object which provides the

service. The main foundations and concepts of the COM family of products are presented next:

- A COM client is any application that requests the COM to instantiate, initialize and release an implementation object known as the server.

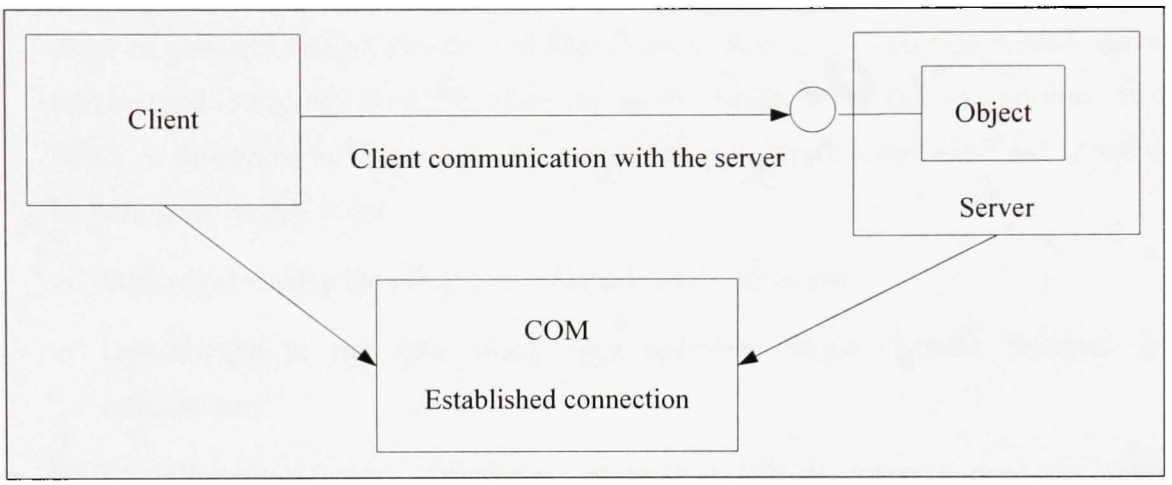

Figure 1.3 Client-server communication.
(Microsoft Corporation, 1995a, p. 13, F 1.2)

- A server is composed of classes which can be instantiated and used by a client. COM servers can be either of two types (Rosemary, 1998):
  - A DLL (Dynamic Link Library).
  - A Cross-EXE stand-alone executable form.

    As a DLL it can be located either In-process in the client workspace or Cross-network in a separate remote process. As an EXE, it can be in the form of a Cross-process, i.e. located in the same local machine but live in a different process or Cross-network in a separate remote process (for example NT services).

- An interface defines the contract between the client and the object. It states that the object that implements this interface has the obligation to provide the responsibilities (functions) defined by this interface.

- Communication between applications using the COM standard is realized through sets of function calls represented by interfaces.
- IDL-Interface Definition Language is used to describe a COM interface and its included set of methods and attributes to define remote procedure calls.
- MIDL (Microsoft Interface Definition Language) compiles the IDL interface definition in order to generate several classes and files (proxy, stub, etc.). These generated artifacts will be used to compile the COM client and server applications. The process done by the MIDL is illustrated in Figure 1.4. The communication between the client and server can be performed in two ways:
  - Statically by using the client proxy and the server stub, and
  - Dynamically at run time using Type Libraries which contains interface type information.
- GUID (Globally Unique Identifiers) are unique 128-bit integers used to identify interfaces and object implementations in the global object space so that conflicts caused by similar object names are avoided. The MIDL is responsible for generating the GUID of classes and interfaces.
- Proxies and Stubs are both generated by the MIDL and are transparent to the client. As shown in Figure 1.4, a proxy lives in the same address space of the client while the stub lives in the server address space. The role of a proxy is to marshal the parameters to be sent through RPC calls. On the server side, the stub receives an RPC call and unmarshals the parameters of the server function call. The reverse operation happens once the server is done with the call processing.
- COM location transparency is realized through the single programming model for all object types (local and remote). The client's accesses to servers are done via interface pointers. If the server is not local, COM uses the proxy object which invokes the appropriate remote procedure call to the remote process of the server. A similar situation can be found on the server: calls to the functions of an object interface are also done in a similar fashion when the client is local; otherwise, the caller is a stub object provided by

the COM who receives the remote procedure call from the proxy in the client process and changes it into an interface call of the server object.

- COM implementation object can implement multiple interfaces. Clients know about the different interfaces implemented by one object through a call to the *QueryInterface* which is provided by any COM interface. The call to the *QueryInterface* is called interface negotiation, a process in which the client asks the object about the service it can provide, i.e. the interfaces it implements. After a client gains access to an object, he receives a pointer to the *unknown* interface which permits the client to control the lifetime of the object. The unknown interface is an integral part of any COM component.

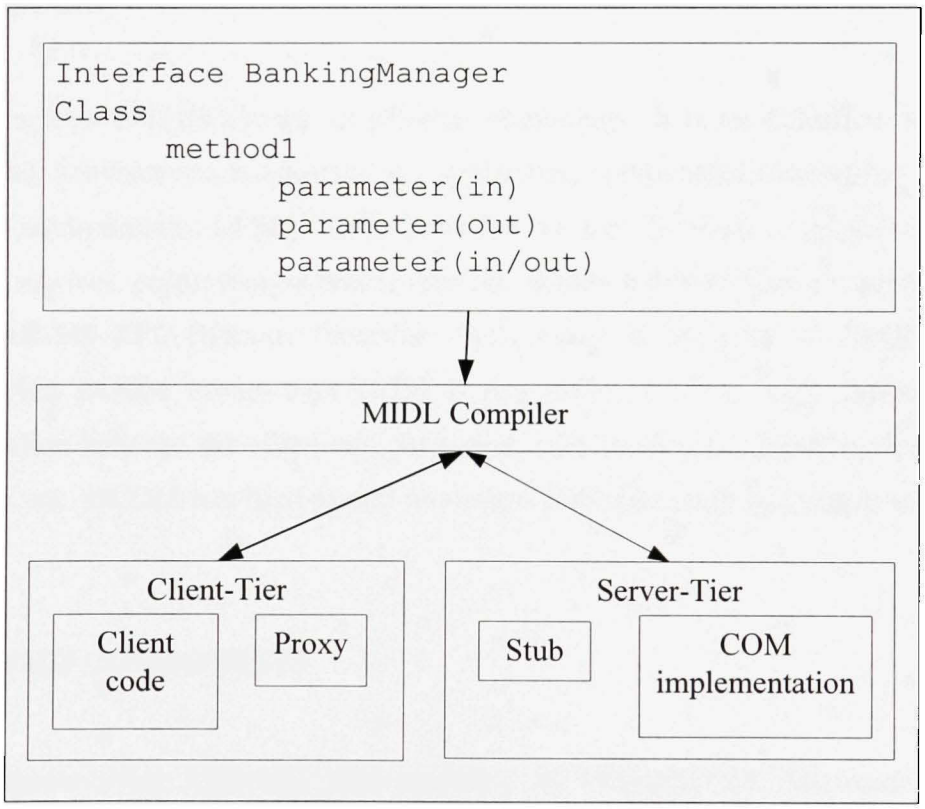

Figure 1.4 MIDL processing.

(Rosemary, 1998, p. 31, F 5.3)

1.8.2 Binary standard

The COM model defines an interoperability binary standard. The creation of objects and the communication between client and server objects is realized through standardized mechanisms independent of the programming language and the application that uses the object services. The binary standard is necessary to allow different vendors and developers to write applications which interoperate without recompilation or specialized code to orchestrate the communication between objects; the communication management is the responsibility of the COM library.

1.8.3 DCOM

DCOM is a Microsoft distributed middleware technology. It is an extension of the COM model which provides the mechanism for application components interactions in network distributed environments. DCOM is the basis for NT services since it provides distributed messaging services, object request broker services, distributed and data connectivity services. DCOM uses MS RPC (Remote Procedure Call) which is based on the DCE RPC. The extension that DCOM brings over COM is the ability to have more control over the communication between the client and the server with respect to aspects such as security, transaction, etc. DCOM has been ported to several platforms, amongst which are DEC and Solaris.

1.8.4 .NET COMPONENTS

.NET components are Microsoft next evolution of COM/DCOM. Microsoft wanted to eliminate the problems experienced with COM/DCOM, specifically the problem of "DLL hell". Even though the technologies of COM/DCOM and .NET have several commonalities, they have nevertheless many differences (Lowy, 2003):

1. .NET components do not inherit the `IUknown` interface as COM/DCOM components do.

2. .NET components use an enhanced garbage collection technique while COM/DCOM components use a technique based on reference counting.
3. .NET components implementation is made easier. For instance, no IDL interfaces have to be declared as is the case with the implementation of COM/DCOM. Furthermore, .NET components require no registry entries, thus making the deployment and relocation of .NET component much easier.
4. .NET component type uniqueness is based on relative scoping and namespaces rather than on Global Unique Identifies (GUID) as is the case with COM/DCOM.

1.9 Sun Java components

The Java programming language was developed as a response to facilitate software development with simple constructs (no pointers) and to bring application portability across platforms by abolishing incompatibilities between different platforms and even different implementation of the same programming languages (C++ on Windows and Unix).

1.9.1 JavaBeans model

The JavaBeans component model defines a lightweight model for component development. This model is kept simple; it was not developed with component distribution in mind. The JavaBeans Model was intended to allow for RAD (Rapid Application Development) via manipulation of Java Bean components visually inside easy to use GUI IDE (Integrated Development Environment) environments.

Java Beans : According to the JavaBeans API specification (Sun, 1997), a Java bean is a reusable software component which can be controlled and manipulated using a visual builder environment. Certain Java beans can be simple GUI components like buttons; others can be graphically invisible but still be visually manipulated by a bean tool builder. From a complexity reduction point of view, a bean should be viewed as a black box with outside

known functionalities, but its internal content is unknown. The distinguishing and unifying features of a typical Java bean are:

- Introspection support used by beans builder tools to discover the working aspects of a bean such as its methods, properties and events. The introspection mechanism plays an important role in facilitating the development of applications based on Java beans.
- Customization support performed to personalize the appearance or behavior of a bean. Usually, this task is done via a visual list of the beans' properties which can be changed by the user to suit a particular need in a bean's behavior.
- Event support so that developed component beans are able to communicate and interact with each other. On the conceptual level, events provide the mechanism through which a change of state that a particular bean undergoes is propagated. The event generating bean is the source and the receiving bean is the target listener. A typical event would encapsulate a chunk of information useful to the bean receiving the event.
- Properties support a mechanism used to represent and customize the behavior of a Java bean through its properties. A Java bean property is an attribute associated with a bean; such an attribute can be read or written by using appropriate methods such as the conventional *setter* and *getter* methods.
- Persistence support that guarantees the state of customized components in case of reloads. Beans persistence is realized through the Java serialization and externalization mechanisms.

Java Beans in distributed environment: Basically, Java beans are considered to run locally in the application address space. However, they can be used in distributed environment through the use of Remote Method Invocation (RMI) and IIOP protocols as shown in Figure 1.5 from (Sun 1997). Java RMI is a Java protocol that supports serialization of objects to be passed from one virtual machine to another across a network. The IIOP protocol permits Java beans to talk through the use of RPC and Java IDL with CORBA servers which can be implemented in different languages.

1.9.2 Entreprise Java beans (EJB)

EJB components are business components used in multi-tier systems (Sun, 1999; 2006). EJB version 1 specification was introduced in 1998. Version 2 was introduced in 2001 with a major improvement of allowing less costly interaction between client and EJB components collocated within the same deployment environment. Version 3, the latest was introduced in 2006 to simplify the complex and intensive syntactical constructs which have characterized the previous versions. This version has strived to ease the development and deployment of EJB components by introducing annotations and lessening the amount of XML configuration to be done.

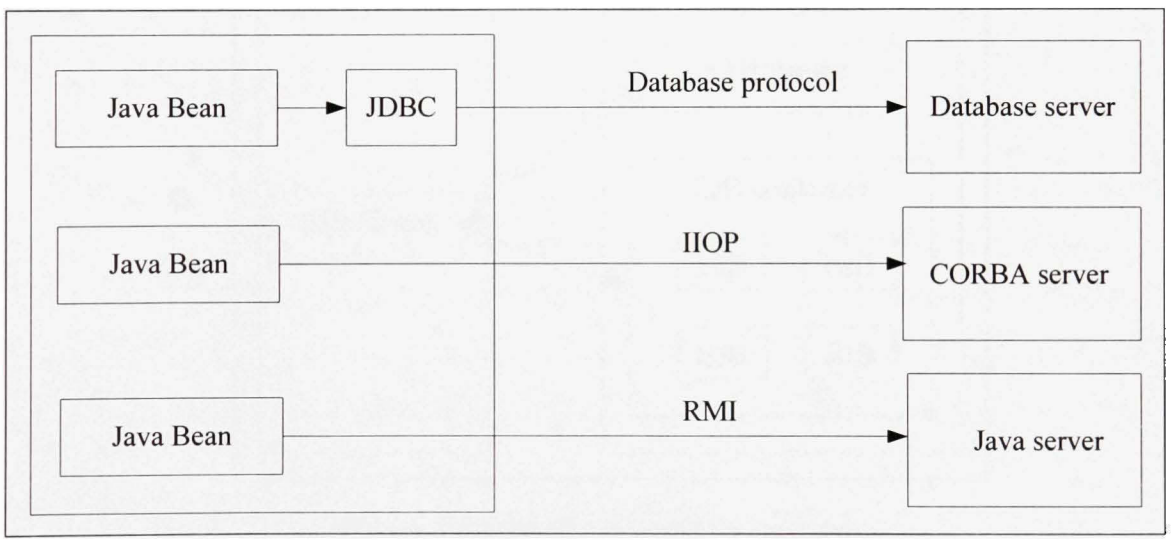

Figure 1.5 Interaction between Java Beans in a multi-tier system.
(Sun, p .12, F 2.1)

When deployed, EJBs live within containers which provide them with various kinds of services. Typical services used by EJBs are transaction monitoring, lifecycle management, component lookup, and communication backend services. Since those services will be

provided for deployed EJBs by the container, developers do not need to implement them. Consequently, it is advocated that using EJB components requires less development effort on behalf of application constructors. This reduction in effort results in channeling the focus of developers to what is really important, i.e the business requirements to be implemented.

The enterprise Java beans model has been developed to provide flexibility, scalability and security to applications. Enterprise Java Beans are portable reusable components written in the Java programming language and, above all, interoperable. By specification, EJB compliant components can be deployed in "any" container with little effort and without being compiled because they are interoperable as shown in Figure 1.6.

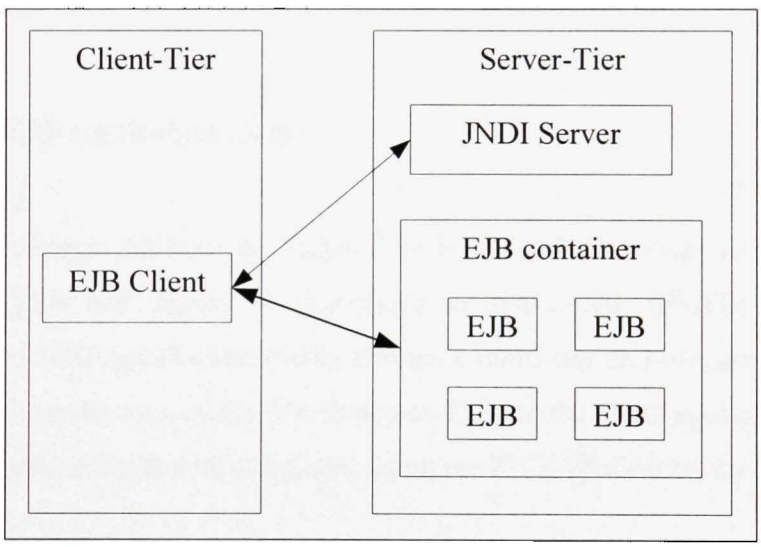

Figure 1.6 EJB living within a container.

There are three types EJBs: Session Beans, Entity Beans and Message-driven beans.

- Sessions Beans represent the type of component which can be either *statefull* or *stateless,* but not persistent in a database. A *statefull* session bean acts like a proxy for a client application and can hold transient data that represent a client state at various stages during a session. Once clients disconnect, the bean is terminated and garbage collected by

the EJB container. A *stateless* bean does not maintain a state thus can be used by multiple clients and is usually used for generic functionalities commonly used by many clients.

- Entity beans represent business objects existing under persistent storage in relational databases. Each entity bean instance is unique in that it possesses a primary key and typically represents a database table row. The state of an entity bean persists beyond its presence in the EJB container. Entity beans can be shared by multiple clients thus transaction management becomes a necessity. Transactions management can be left to the EJB itself or delegated to the EJB container.
- Message-driven bean permit J2EE applications to interact with synchronous and more importantly asynchronous messages. The asynchronous message-processing mode, which makes it possible to avoid tying up resources, is what distinguishes them most from the other EJB types.

1.9.3 J2EE EJB application model

J2EE (Java 2 Enterprise Edition) is designed with enterprise services in mind. Enterprise services are complex and should be automated in distributed, flexible, scalable, secure environments. The J2EE application model defines a multi-tier architecture for applications; so that Enterprises services are easily distributed as EJBs to the client applications. Figure 1.7 shows how the applications can be structured using the J2EE application model.

The J2EE application model has three elements: the client-tier, the middle tier and the enterprise-tier.

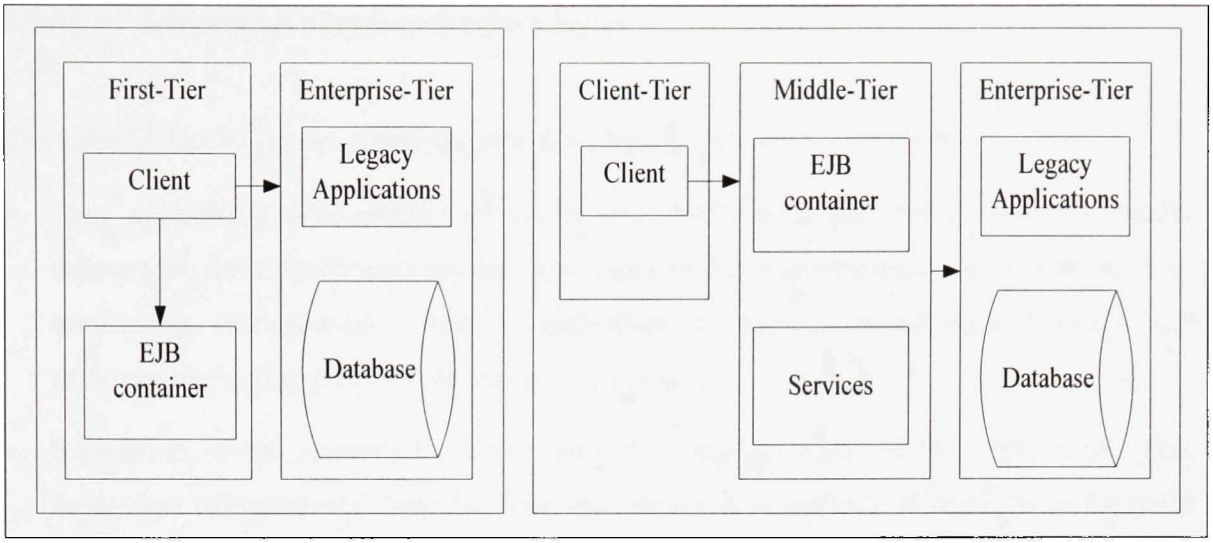

Figure 1.7 EJBs in Multi-tier architectures.
(Sun, 2006)

The client tier can be in the same process (executing environment) of the middle tier and usually provides a representation of the application services results. The separation between the two allows distinguishing the client of the service from the provider of the service. It must be noted that the client can be another middle tier application.

The middle tier application is where EJBs live. These EJBs live in EJB containers which are responsible to provide the infrastructure for the communication, life-cycle, security, persistence, transaction monitoring services needed by those EJBs. Consequently, the application constructor is relieved from providing those infrastructure responsibilities and hence the application development is eased and simplified, permitting a better application reuse of EJB components.

The enterprisetier groups the legacy applications and database systems and is usually located on a different machine for security and performance reasons.

1.9.4 Advantages of enterprise Java beans

The use of EJBs has several reported benefits (Sun, 2002) among which are:

- Easy application development of distributed enterprise applications, since the model relieves the developer from implementing many of the implementation intricacies such as transaction management, security, authorization and communication between the different interacting parts of a distributed application.
- Separation of the presentation logic from the business logic of the application. This separation offers several benefits. The application development is reported to be more modular and flexible. For example, the experts in presentation development can concentrate on the client part while business logic experts focus on the EJBs development part.
- Enterprise beans are written in Java and thus, are portable components. These components can be assembled to build new applications which can be executed in any compliant EJB container.

1.9.5 Limitations of enterprise Java beans

As with any product, improvement through evolution is the way through which newer versions emerge. Among the limitations from which the EJB technology suffered, in particular in versions 1 and 2, was the complexity of implementing EJBs and the intensive effort spent on maintaining and manipulating several XML configuration files in which many properties related to the deployment, security, and transaction management of EJBs have to be set. Performance-wise, it has been claimed that the EJB entity beans have been associated with slower applications (Johnson, 2002). This limitation, has been behind the development of EJB3 specifications as well as the emergence of several frameworks based on lightweight containers such as Guice (Google, 2007), Spring (Interface21, 2003) and Pico containers (Google, 2007; Interface21, 2003; PicoContainer, 2008). Such lightweight containers are known to have better performance results compared to J2EE containers. The Spring

framework is a typical example of such frameworks which provide light weight containers while providing more or less the same service as the ones provided by EJB containers.

Table 1.4 Summarized comparison between the major component models

CORBA	COM/DCOM	Java components
Is a set of specifications; implementation is left to vendors. Being specifications, which are sometimes ambiguous or left to the judgment of implementer contributed to its adoption decline.	Specification and implementation controlled by Microsoft.	Specification and implementation led by Sun within a community process.
Uses RPC, GIOP, IIOP communication mechanisms and protocols for network communication.	Uses RPC as a communication mechanism.	Uses RMI mechanism over JRMP protocol for network communication.
Uses IDL to define interfaces inside modules. Interfaces are stored into an interface repository.	Uses IDL to define Object interfaces. COM objects can be queried about the interfaces they support. IDL can be used to generate stubs to communicate with CORBA components.	Interfaces are available in the Java language. Introspection can be used to discover the interfaces implemented by Java components. IDL is provided to generate stubs to communicate with CORBA components.

Table 1.4 Comparison summary between the major component models (continued)

Components and interfaces are identified by unique names registered in the interface and implementation repositories.	Components and interfaces have GUIDs, i.e. unique identifiers.	Java components can be registered and accessed via JNDI (Java Naming and Directory Interface).
Components are interoperable thanks to the infrastructure provided by ORBs. They can be written in any programming language.	Components are written in C++ and based on the COM architecture. The COM architecture has been ported to other platforms.	Components can run on any platform provided a JVM (Java Virutal Machine) host is installed.
Errors are caught through thrown exceptions.	Errors are signaled via a specialized error data structure.	Errors are caught through thrown exceptions.
Components live in a container.	Live without a container.	Components live in a container.
Coarse-grained	Coarse-grained	Coarse-grained

1.10 Comparison between CCM, Microsoft and Sun components

In the previous sections of this chapter a brief overview of the major component models has been presented. All of the presented models provide support for distributed cross-network component interactions. In this section, a comparison between these models, inspired by the work of (Goplan, 1998), is presented. The comparison is conducted from both a technical and architectural perspectives and is illustrated in Table 1.4. For instance, CORBA components put the emphasis on the interaction of distributed components across different platforms and

programming languages. SUN EJBs have been geared more toward distributed multi-tier architectures. Other components, such as Java beans and ActiveX components put the emphasis on the GUI aspects. CCM has been extended by adopting container architecture similar to EJB container architecture.

1.11 Compositional languages

One of the goals of software engineering is to transfer the construction process to higher levels of automation with the aim of reducing software complexity and achieving better software reuse (Abran *et al.*, 2004). In parallel with this is the goal of using component composition in software construction. Compositional languages are used to interconnect, compose and connect software code elements together.

Due to the recent interest in compositional languages, the 2004 version of the SWEBOK Guide (Abran *et al.*, 2004) has not yet identified such languages as a separate type of programming language. However, the SWEBOK Guide still identifies configuration, toolkit, script and general-purpose programming languages which can be considered more or less as compositional languages. Even though compositional languages emphasize the composition relationship between components, they still have lots of commonality with other types of programming languages. In particular any programming language can be considered as a compositional language since it allows the composition of various types of code elements.

Compositional languages have started to emerge and still belong to the research domain. Among the various compositional programming languages are:

- IBM BML (Bean Markup Language) (Coplien, 1992; Weerawarana, Curbera and Duftler, 2001);
- Piccola (PI Calculus based Compositional Language) (Achermann *et al.*, 2001) and;
- CoML (Component Markup Language) (Birngruber and Kepler, 2001a).

1.11.1 Bean markup language (BML)

BML, a declarative component composition language, aims at inter-connecting Java Bean components to construct software applications. The BML Language is experimental and has been developed at IBM research laboratories. BML is not intended to create classes or business logic: it is intended to compose Java bean components to perform computational tasks. The syntax used by BML to define the composition and interconnection relationships between an application Java Bean components is XML (eXtensible Markup Language) (IBM, 1998; Johnson, 1999). BML permits the developer to declare Java bean components, compose them via hierarchical associations and let them collaborate on a computational task via definition of the inter-connection between those components. The interconnection and composition relationships are represented by method calls and event production and consumption.

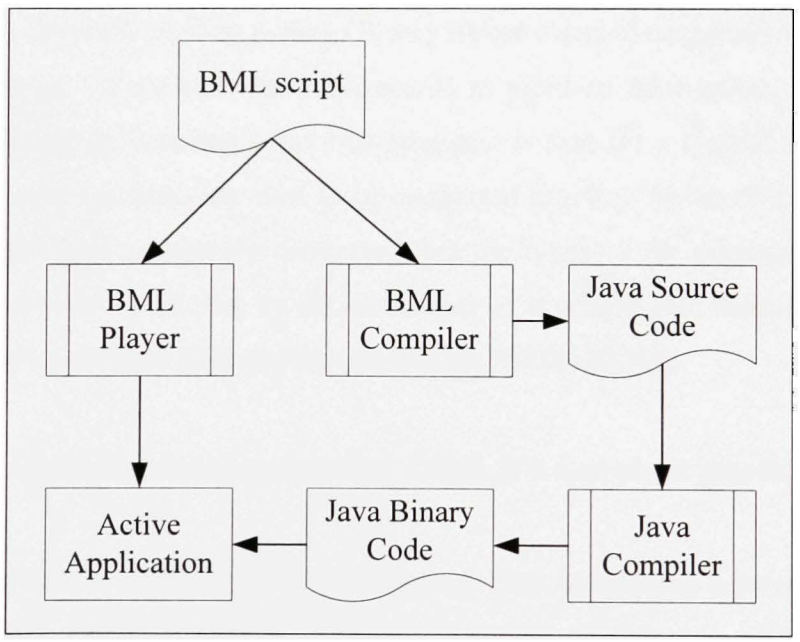

Figure 1.8 Application construction using BML.

Application construction and execution using BML: To construct an application the user declares the Java Beans and the inter-connection relationships which exist between those components in an XML script file having a "bml" extension. The execution of the constructed application is done in either of two ways as illustrated in Figure 1.8:

- First, through direct interpretation and execution of the ".bml" script file which is passed to the BML player to parse, interpret and ultimately execute.
- Second, the ".bml" script is passed through the BML compiler which parses its content, transforms and generates its equivalent Java source code.

Next, the developer passes the Java source code to a Java compiler to generate the Java binary which can be ultimately executed in a JVM.

1.11.2 Component markup language (CoML)

CoML is a compositional language developed at the University of Linz in Austria. The current implementation is used to connect binary object oriented components. This language is used to compose binary software components to yield an application. The assumption made in developing applications using this language is that for a typical application there exists a set of components which need to be connected together. Some of those components have known types and are already connected, but the types of the others are not precisely known until they will be chosen by the developer in a wizard-like manner from a set of predefined component types (Birngruber and Kepler, 2001a; 2001b).

The process of application development using CoML is a dual-phase process as illustrated in Figure 1.9.

- In the first phase, a script in CoPL (Component Plan Language), a proprietary language using Java-like syntax is used to connect the known components with other unknown components but defined over the set of predefined components. This script plays the role of a reusable abstract application.

- In the second phase, the developer feeds the CoPL script along with the selected component types which satisfy the requirement into the composition descriptor generator to yield an XML file, i.e. the CoML script which describes the components and their composition relationships.
- Later on, the CoML XML file is passed to a binary generator tool to produce the binary code of the application.

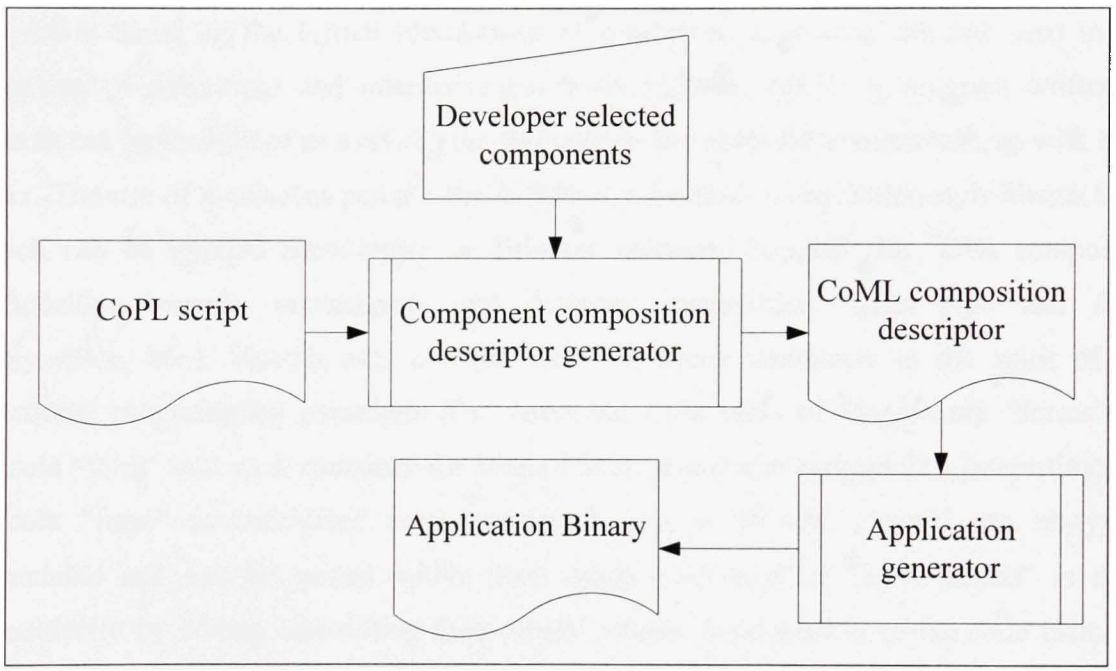

Figure 1.9 Application development using CoML.

What is special about CoML is that it is generated semi-automatically along with interactive developer input to produce an XML component composition descriptor. The interactive input is "type" information to be injected in the concrete application after generating the CoML code. The injected component type is not an unknown type which is created dynamically, but rather a predefined known type to the script, i.e., applications constructed using CoML are statically typed as is the case with Java.

1.11.3 Piccola compositional language

Piccola, a declarative compositional language belonging to the research domain, has been developed at the University of Bern in Switzerland. It is intended to be a general purpose language for software composition (Achermann, 2002; Achermann *et al.*, 2001; Achermann and Nierstrasz, 2002).

Piccola is based on the formal foundations of π-calculus, a process calculus used in the modeling of concurrent and interactive proc esses (Milner, 1993). A program written in Piccola can be thought of as a set of processes which use channels to communicate with each other. The use of π-calculus permits the definition of various composition style abstractions which can be applied accordingly to different software domains (i.e., GUI component composition, events, components and listeners composition, Unix pipe and filter composition, etc.). Piccola uses a small set of syntactic constructs in the spirit of the functional programming paradigm. The structural code units of Piccola are "forms". A Piccola "form" acts as a container for labels (data) or services (behaviors). Interestingly, a Piccola "form" is considered as a component per se. Piccola "forms" are un-typed, immutable and can be nested within each other. Extension of those "forms" is done dynamically by adding and setting their labels' values. Organization of the code elements used in the construction of an application using Piccola favours the use of architectural styles (Coplien, 1992) to suit particular application domains.

Constructing applications with Piccola is done via the definition of a script file with "picl" extension. This file is loaded into a console application for execution.

1.11.4 Aspect-oriented programming (AOP)

AOP languages provide a mechanism for composing cross-cutting concerns scattered across different classes in an object-oriented application (Ramnivas, 2003; Rashid, 2001). Those concerns consist of common code chunks spread across several source code classes. Method

logging and transaction interception are typical examples of such concerns. As illustrated in the application code shown in Table 1.5, Table 1.6 and Table 1.7, AOP provides an elegant mechanism to strip out those common code chunks and put them in a new "modular" structure known as an "aspect". By doing so, the original code becomes less entangled, more readable and therefore exhibits nicer modularity.

Table 1.5 Class advised by logging aspect

```
public class AccountManager {

    public void withdraw(int amount) {// withdraw logic
    }

    public void deposit(int amount) {// deposit logic
    }

}
```

Table 1.6 Aspect and its advice declarations

```
// aspect structure
public aspect Logging {

    // pointcut and its set of matched joinpoints
    pointcut operation(): call(* AccountManager.*(..));

    // before advice
    before() : operation {
        System.out.println("operation start: ");
    }

    // after advice
    after() : operation: {
        System.out.println("operation end: ");
    }
}
```

From an implementation point of view, each place in the code from which a common chunk is striped of is referred to as a `pointcut` (see Table 1.7). A `pointcut` is an execution point

in the application control flow and refers to one or more of what is know in AOP as `joinpoint`. At each join point in the application, one or more code chunks can be injected with custom computation logic. An injected code chunk is referred to in AOP as an `advice`.

In Table 1.5, the class `AccountManager` is defined. This class has two methods for deposit and withdrawal. Since those two methods affect the banking account, logging their execution details is important especially at execution beginning and end. Traditionally, the way of logging those details is by inserting logging instructions directly at the beginning and at the end of each of the `AccountManager` class illustrated in Table 1.5. Another modular and elegant way of logging those details is to use AOP to put the logging logic into a separate structure (aspect + advices) as shown in Table 1.6 and weave this aspect with the application to obtain the same runtime behavior.

Obviously, the definition of the logging aspect and its logging advices provides a clean and modular application code; however, this definition is not enough to make use of aspects. AOP is a more involved process where both the defined aspect and application code must be passed into an aspect waver (compiler) to weave (integrate) the aspect with the rest of the application code. The process of aspect weaving can be done either statically or dynamically with the application as illustrated in Figure 1.10 and Figure 1.11 respectively:

Static aspect weaving is a process during which aspects are woven either with source or binary byte code of the application before application deployment or execution. By contrast, dynamic aspect weaving is the process during which aspects are woven inside the target classes at application deployment or execution.

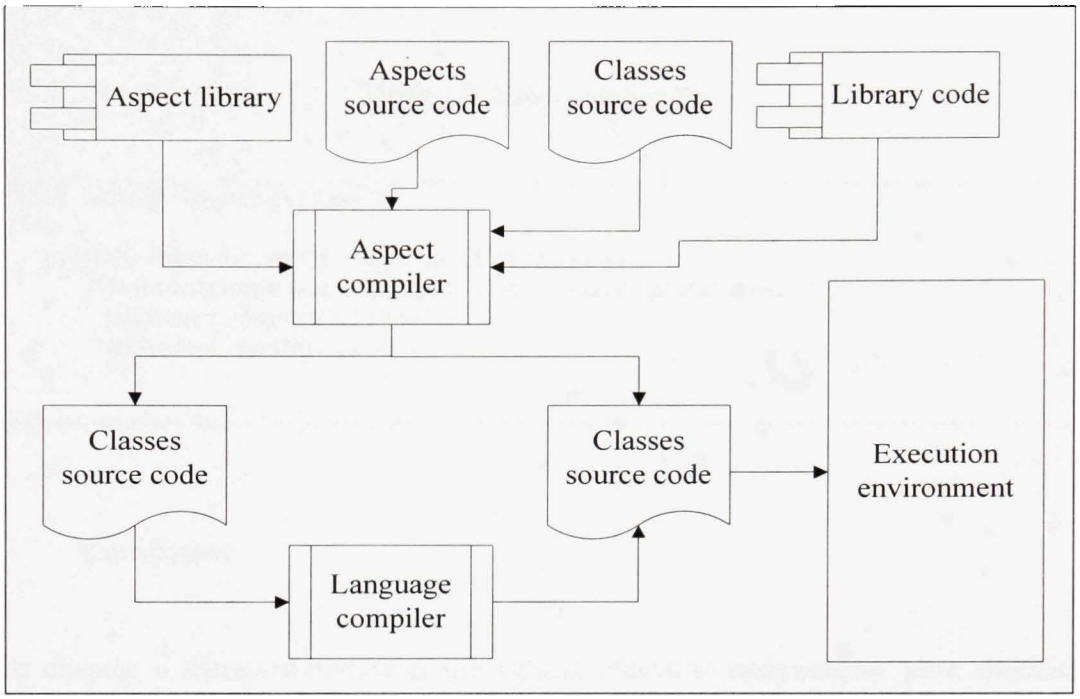

Figure 1.10 Aspect static weaving process.

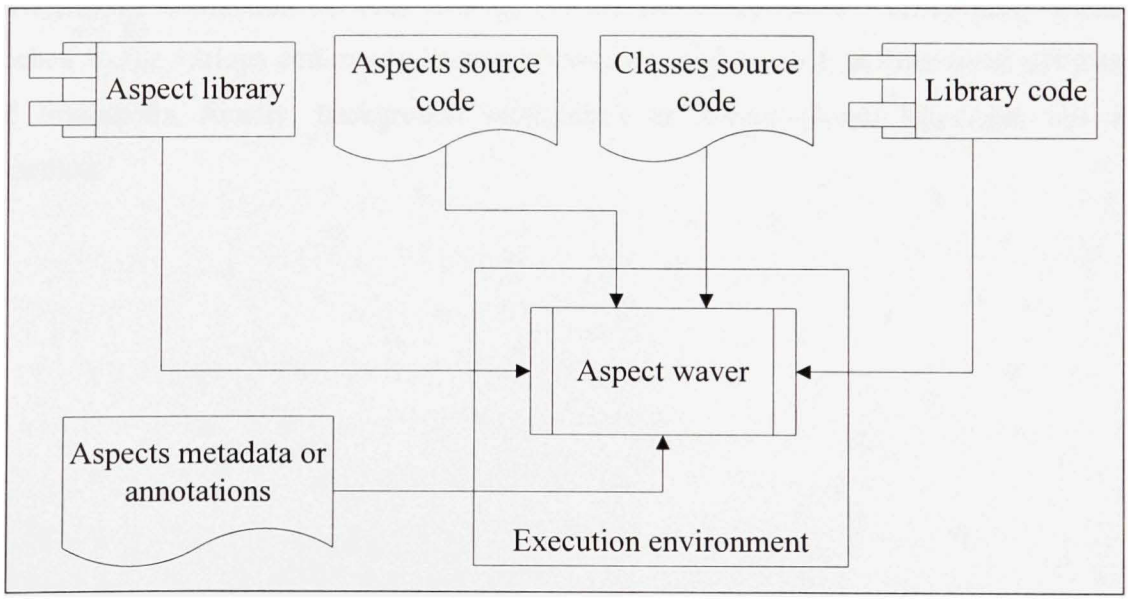

Figure 1.11 Aspect dynamic weaving process.

In Table 1.7, the main application is defined methods are invoked.

Table 1.7 Aspect application

```
public class Application {

    public static void main(String[] args) {
        AccountManager manager = new AccountManager();
        manager.deposit(100);
        manager.withdraw(50);
    }
}
```

1.12 Conclusion

In this chapter, a literature review of the aspects related to components, their architectures and composition has been presented. The purpose of this overview was to introduce the important concepts and aspects related to software components and construction of software application based on components composition. The overview herein presented aims at clarifying the distinction between classes, objects and components. Furthermore, it briefly touches on the various commonly known component models, their architectures advantages and limitations. Finally, background information on compositional languages was also presented.

CHAPTER 2

SOFTWARE COMPOSITION

2.1 Introduction

Software composition can not be considered as a completely new software development paradigm even though it might be perceived likewise. In essence, there exists a form of software code elements composition in nearly all software development paradigms. For instance, in functional and procedural languages operations or functions are composed together to produce other operations or functions. Similarly in OOP, classes are composed to form other classes and so on. However, what is to be considered a novelty nowadays is the use of the component element as a basic building block in software construction. The usage of this new code element (Component) is primarily driven by the various advantages it provides.

2.2 Composition layers

Software applications as they are currently constructed use an amalgamation of various reusable software code elements. Often, those code elements belong to a variety of business domains organized in hierarchical composition layers. Among these layers, the following are distinguished:

1. Module or package composition. In this layer, modules or packages are composed with other modules, packages, classes, operations, and programming language instructions (namespace uses, package imports) to yield other modules or packages.
2. System, component and class composition. In this layer, packages, subsystems, classes and operations are composed with each other. An essential discriminator of this layer from the previous is the composition of classes and operation.

3. Aspect-oriented and View-oriented programming composition. In this layer, aspects, class fragments representing class views are composed with classes or other Aspects and views to yield newer classes.
4. Operation and macro composition. In this layer, operations, programming language instructions and macros (pre-defined replaceable programming language instructions) are composed to yield new operations.

2.3 Effects of granularity on reuse

A noticeable observation of the composition layers discussed in the previous section is the granularity of software elements which take part in the composition process. Obviously, as the granularity decreases, the reuse potential increases. Typically, granular elements are composed of less granular ones and so on the composition recurrence continues. However, the reuse potential of fine grained elements can be increased provided that those elements are structured in such a way that favors their reuse and composition. Obviously, the level of granularity plays a considerable role in the possibilities of reuse.

2.4 Software composition types

Software code elements can be composed either structurally or behaviorally.

2.4.1 Structural software composition

Structural composition refers to the composition of code elements in a common physical structure. The relationships which exist between the various composed code elements are dictated by the relevance of the code structures to each other as well as the functional requirements the software has to satisfy. Structural composition is used as a mechanism to structure state, i.e. the data hierarchies in single blocks. For instance a "Person" component can be composed structurally of two components: "Name" and "Family name".

2.4.2 Behavioral software composition

Behavioral composition refers to the composition of code elements in terms of computational services they provide for each others. The typical presence of behavioral composition can be observed in the context of operations and functions calling other operations or functions. Behavioral code elements manipulate or operate on structural code elements. To illustrate the idea of structural versus behavioral composition, consider the case of a software application which provides matrix addition functionalities. The single data structure holding the elements of the matrix is a compositional structure illustrating structural state composition. However, calling a function to add up two matrices represents a behavioral composition of two code elements (the caller and callee functions).

2.5 Software composition categories

Software code elements can be composed at various phases of the software lifecycle. Roughly speaking, software composition which is happening during the various software lifecycle stages can be categorized as being either static or dynamic (Nierstrasz and Tsichritzis, 1995).

2.5.1 Static software composition

Composition of code elements during the lifecycle of a software prior to deployment into production environment falls into the static composition category. In this category, code elements are composed statically often via a code element editor such as text editor or dedicated IDE such as Eclipse (Eclipse Foundation, 2008). The composition follows the interaction scenarios devised by the developer to satisfy the requirements of end users.

2.5.2 Dynamic software composition

Composition of code elements while the software is deployed in a production environment falls into the composition of dynamic software category. In this category, the composition activities are mainly driven by the end user interactions with the software. The composition activities represent part of the interaction between different code elements of single local or multiple distributed applications to yield out the desired computational result. Typical examples of composing code elements inside a single application can be experimented when composing JavaBeans at runtime inside a specialized GUI-equipped IDE. Equally, the composition of J2EE, .NET, CORBA and Web services code elements, illustrates the case where dynamic code element composition occurs between multiple distributed applications implemented in a variety of technological flavors.

The ability to provide dynamic composition requires a kind of sophisticated provision in the form of code element metadata. This metadata is introspected at runtime so that code element services (interfaces) are discovered and then composed with other services in such a way to satisfy end-user requirements. Obviously, code element introspection at runtime comes with a a runtime overhead which affects negatively the overall application performance.

Nevertheless, dynamic composition of code elements has its advantages. For instance, it postpones the composition decision untill runtime which provides better flexibility and control over the software behavior. Moreover, dynamic composition offers the ability to remove the need to stop-deploy-start maintenance cycles in the event of certain code elements modifications or upgrades.

2.6 Component-based software engineering (CBSE)

Software engineering processes have evolved through several programming paradigms: from the functional and structured moving to the object-oriented and recently to the component-oriented paradigm as of the beginning of the 1990's.

CBSE is divided into two distinct processes (Ghosh, 2002; Rashid, 2001): component engineering and application engineering. The first deals with the analysis and development of domain-generic and domain-specific components, while the second deals with software application development by assembly, composition, integration and plugging of components such as COTS (commercial off-the-shelf) and other in-house developed components. Current software development processes have a mix of both processes. The use of two distinct processes is an ambitious goal since it necessitates abundance of components on the one hand and flexible straightforward assembly and composition mechanisms on the other hand.

The Software Engineering Institute at Carnegie Mellon University (SEI, 2003) uses the term CBSD to refer to the process of software development by the assembly and integration of software components. Essentially, the terms CBSE and CBSD refer more or less to the same process. The focus of CBSE is on the development of software by assembling and integrating COTS and other existing types of components with an emphasis on composition rather than on programming (Clements, 1996). It assumes that certain software parts are common to several software applications, therefore exhibit an interesting reuse potential.

2.7 Component based software construction

Software construction can be considered as a sub-process which matches the implementation phase in the software development life cycle. Software construction per se is a software engineering act which encompasses the activities of software coding, validation and unit testing. According to the SWEBOK (Software Engineering Body of Knowledge) Guide (Abran *et al.*, 2004), this sub-process must be instantiated taking into account four general principles (*reduction of complexity, anticipation of diversity, structuring for validation and the use of external standards*) as well as the tools used by this sub-process such as compilers, code generators and development tools.

As software design breaks software down into smaller parts for construction, those parts are expected to comply with the general principles of software construction. Interestingly enough, component-based software construction meets those principles. For instance:

1. Components can reduce the complexity of an application since they are expected to be modular reusable parts which can be bought from specialized suppliers instead of being developed in-house;
2. Components are reusable and replaceable parts, thus they meet the anticipation of diversity principle;
3. Components are expected to be modular and therefore their validation is expected to be easier;
4. Finally, software components generally conform to their own component models, and therefore they comply with standards.

The SWEBOK Guide (Abran *et al.*, 2004) identifies three styles of software construction: linguistic, visual and formal. These styles are general and are applicable to almost any software development process. Component based software construction may use any particular style or a combination of those styles.

2.8 Drivers behind software component composition

The interest in software component composition is driven by their ability and suitability to satisfy several requirements:

1. Components intrinsically imply their ability to indefinite composition possibilities. In other words, components are believed and expected to be easier to compose than other types of code elements. In particular, components are easy to adapt and customize to suit current and evolving requirements.
2. Components are often empowered with abilities (services) to respond to many environmental and business constraints unlike classes. For instance, in addition to their intended business computational service, components make use of star services (transaction, lifecycle, caching, etc.) to provide their own services.

2.9 Conclusion

In this chapter, the principle of software composition is presented. Various aspects related to software construction by component composition are discussed. In particular, composition layers and effects of granularity are discussed. Equally, composition types and categories are presented. Finally, component based software engineering and component based software construction as well as the drivers behind the interest in software construction based on component composition are also presented.

CHAPTER 3

RESEARCH METHODOLOGHY AND RESEARCH ISSUES

3.1 Introduction

Among the various programming languages, object oriented languages have been receiving noticeable attention by developers and researchers. One interesting feature of these languages is their efficiency in narrowing the gap between real world objects and the abstract representation of those objects in terms of software elements. With the emergence of the software component technology, the component abstraction is used as a means to promote software composition. Although software components have been considered as a new computational revolution, they are still far away from providing the sought panacea for the "software crisis". In particular, software components still suffer from a number of limitations.

3.2 Goal and Objectives of this research

The goal of this research project is to improve the construction of software using components. In particular, this research aims at promoting software composition by evolving software construction from a process dominated by code writing towards a process relying increasingly on components composition.

The objectives of this research project are:

1. To provide a software component model to remedy for some of the limitations facing software construction by components composition;
2. To provide a reference implementation for this component model;
3. To provide a measurement method to measure components unwanted members;
4. To provide a prototype tool to measure components unwanted members;
5. To provide a component versioning mechanism;

6. To provide a prototype tool to detect component versions mismatch.

3.3 Research methodology

The research methodology used to achieve the objectives of this research involves the following steps:

1. Identify and illustrate a number of limitations facing the composition of software components. The identified limitations are listed below and are discussed in detail in the next section:
   a. Unwanted component's members;
   b. Chaotic composition and amalgamation of code elements;
   c. Suboptimal component reuse;
   d. Component version mismatches.
2. Propose a measurement method and implement a prototype tool to measure component's unused members (see chapter 4).
3. Propose a component model to remedy for the unwanted component's members limitation (see chapter 5).
4. Propose an enhanced version of the previous component model. Not only this model remedies for all the limitations identified in this research work, but it overcomes limitations of the previous model. This new component model is presented in chapter 6.
5. Illustrate the process of software construction and component based software construction based on the proposed enhanced component model (see chapter 7).
6. Propose a component versioning scheme to detect component version mismatches in chapter 8. Furthermore, a tool to detect component version mismatch is presented in the same chapter.
7. Implement a compiler for the solution component model. This compiler is presented in chapter 9.
8. Conduct a case study to illustrate how the proposed component model improves software construction in chapter 10.

9. Evaluate the solution approach, tools and effort expended in this research work in chapter 11.

3.4 Software component limitations

Software component composition can be either static, which happens normally during software construction and before deployment or process, or dynamic which occurs at runtime. To achieve fluid and straightforward composition components need to be easily customized and adapted to current and evolving requirements. The limitations addressed in this research work concern more static component composition and to a lesser degree, dynamic component composition. The limitations addressed in this research are explained next in details.

3.4.1 Unwanted components' members limitation

OO oriented software classes (components) used by a variety of applications, also commonly known as reusable software elements, exhibit a stiff structure when it comes to the reuse of their individual members (data or operations). A typical example of this limitation can be observed with the String class of the Java API. Based on our industrial experience, only a subset of all the operations of this reusable class are being used by the same application. Obviously, the unused set of those operations lingers in the application without being used. A stripped down version of this class including only the required operations for a particular application would have a smaller size and therefore would be more appealing to be used for the following reasons:

1. Such a stripped down version, reduces the loaded class code memory foot print, a condition beneficial to memory-scarce environments particularly in portable devices.
2. Accessibility, manipulation and tempering with unused members may lead to side effects and therefore compromise the hosting application integrity and security if exploited inappropriately or inadvertently.

3. It may reduce network traffic in distributed applications where dynamic code transfer happens as is the case with Java web start applications.

Typically, reusable software components bundled in APIs and containing hosts of members (data and functionalities) tend often to possess coarse to large-grained sizes. Since the set of functionalities required by an application varies according to its particular context, an excessive number of unwanted (unused) functionalities will be present when using such reusable software components. An illustration of such an application particular context is shown in Figure 3.1.

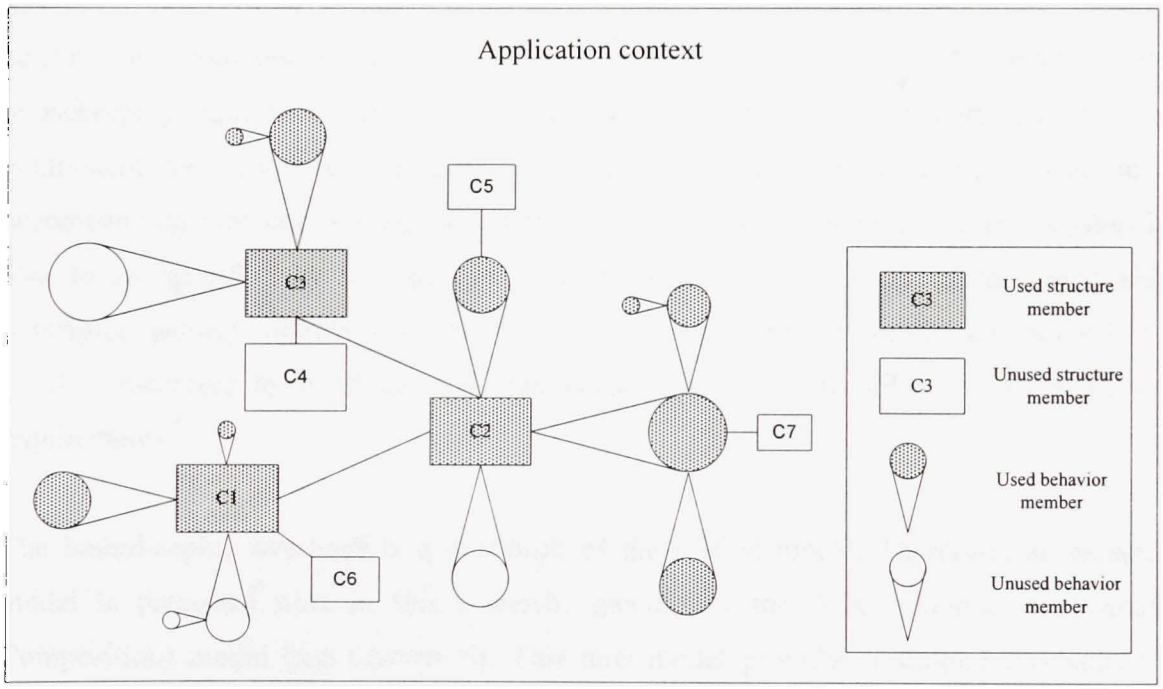

Figure 3.1 Illustrative structure of used and unused application members in a particular context.

To remedy this limitation, compositional wrappers have been suggested in (Al-Hatali and Walton, 2002). Even though this solution hides unwanted data members, this remedy is not efficient for several reasons:

1. It adds another layer of indirection (indirect access) when the members of the components need to be accessed.
2. The code size of unwanted members is not reduced, but increased with the newly added wrapping layer.

In this research project, we propose an initial model, the Compositional Structured Component Model (CSCM), discussed in chapter 5, to remedy to the limitation of unwanted component's functionalities (Msheik, Abran and Lefebvre, 2004). To do so, the CSCM model makes use of composition metadata so that composition of selective component functional composition is made possible. This model provides a mechanism to get rid of unwanted members by isolating them from being included in the application. However, constructing components and applications according to the CSCM model requires housekeeping tasks to be performed by the software constructor. In particular, for each component the composition relationships between members have to be defined in an accompanying xml composition descriptor file. Furthermore, dependency relationship has also to be specified by the software constructor also in the body of the composition descriptor. During application construction later on, the software constructor may select or deselect members by modifying the composition descriptor according to his application requirements.

The housekeeping overhead is a drawback of the CSCM model. Therefore, an enhanced model is presented next in this research, specifically the AOC (Atomic and Optimal Composition) model (see Chapter 6). This new model provides a simpler mechanism to define the composition and selection of component members compared to the way it is defined using the CSCM model. In particular, the AOC model uses a simple textual file to specify the selection of required component members instead of annotations. More importantly, the AOC model is equipped with additional features which address other component limitations unaddressed by the initial CSCM model and tackled in this research project.

3.4.2 Chaotic amalgamation and composition of code chunks limitation

Currently, software applications constructed using components result in ambiguous and chaotically amalgamated compositional code chunks. Often, application construction uses a mixture of various code elements and chunks such as components, objects, operation, and programming language instructions. Those code elements would be amalgamated, composed, assembled and wired in freely organized forms. In other words, satisfying requirements is the only necessary condition to be met by the application, irrespective of the way its code elements are structured and organized. For instance, the code in Table 3.1 and Table 3.2 which concern a simple phone directory application (read a phone directory text file and display its entries) can be written in a variety of ways while still delivering the same behavior.

Table 3.1 Phone directory monolithic application example part 1

```
package acModel.monolithic;


public class PhoneDirectoryApp{

  public List<Entry> readEntries(File file)
                          throws IOException {
      FileInputStream fileInputStream =
             new FileInputStream(file);
      InputStreamReader inputStreamReader =
              new InputStreamReader(fileInputStream);
      BufferedReader bufferedReader =
             new BufferedReader(inputStreamReader);
      String line = null;
      List<Entry> entries = new ArrayList<Entry>();
      while((line = bufferedReader.readLine()) != null) {
          String[] values = line.split(",");
          Entry entry = new Entry();
          entry.firstName = values[0];
          entry.lastName = values[1];
          entry.internationalCode = values[2];
          entry.areaCode = values[3];
          entry.phone = values[4];
      }
      return entries;
  }
```

Table 3.1 Phone directory monolithic application example part 1 (continued)

```
  public static void displayDirectory(List<Entry> entries) {
      for (Entry entry : entries) {
          System.out.println(entry.firstName + " "
              + entry.lastName
              + ", " + entry.internationalCode
              + "(" + entry.areaCode + ")" + entry.phone);
      }
  }

  public static void main(String[] args) throws IOException {
      PhoneDirectoryApp app = new PhoneDirectoryApp();
      String filePath = "C://ProgramFiles//EclipseWorkspace//"
        + "ABC_Component_Model//phoneDirectory.txt";
      File file = new File(filePath);
      List<Entry> entries = app.readEntries(file);
      displayDirectory(entries);
  }
}
```

Even though, the code of this application is small, it shows monolithic signs. For instance, the same code found in Table 3.1 can be provided in many variations in particular as shown in Table 3.3.

Table 3.2 Phone directory monolithic application example part 2

```
package acModel.monolithic;

public class Entry {

    private String LastName;

    private String firstName;
    private String internationalCode;
    private String areaCode;
    private String phone;

    public String getLastName() {
        return lastName;
    }

    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
```

Table 3.2 Phone directory monolithic application example part 2 (continued)

```
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getInternationalCode() {
        return internationalCode;
    }
    public void setInternationalCode(String internationalCode) {
        this.internationalCode = internationalCode;
    }
    public String getAreaCode() {
        return areaCode;
    }
    public void setAreaCode(String areaCode) {
        this.areaCode = areaCode;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

Table 3.3 Phone directory refactored monolithic application example part 1

```
package acModel.monolithicRefactored;

public class PhoneDirectoryApp{

    private BufferedReader getBufferedReader(File file)
                                throws IOException {

        FileInputStream fileInputStream =
            new FileInputStream(file);
        InputStreamReader inputStreamReader =
            new InputStreamReader(fileInputStream);
        BufferedReader bufferedReader = new
            BufferedReader(inputStreamReader);
        return bufferedReader;
    }
```

Table 3.3 Phone directory refactored monolithic application example part 1 (continued)

```
    private Entry bindValuesToEntry(String[] values) {
        Entry entry = new Entry();
        entry.setFirstName(values[0]);
        entry.setLastName(values[1]);
        entry.setInternationalCode(values[2]);
        entry.setAreaCode(values[3]);
        entry.setPhone(values[4]);
        return entry;
    }

    public List<Entry> readEntries(File file)
                          throws IOException {
        BufferedReader bufferedReader
                       = getBufferedReader(file);
        String line = null;
        List<Entry> entries = new ArrayList<Entry>();
        while((line = bufferedReader.readLine()) != null) {
            String[] values = line.split(",");
            entries.add(bindValuesToEntry(values));
        }
        return entries;
    }

    public static void displayDirectory(List<Entry> entries) {
         for (Entry entry : entries) {
            System.out.println(entry.getFirstName() + " "
                + entry.getLastName() + ", "
                + entry.getInternationalCode() + "("
                + entry.getAreaCode() + ")"
                + entry.getPhone() );
          }
     }

     public static void main(String[] args) throws IOException {
        PhoneDirectoryApp app = new PhoneDirectoryApp();
        String filePath
               = "C://ProgramFiles//EclipseWorkspace//"
           + "ABC_Component_Model//phoneDirectory.txt";
        File file = new File(filePath);
        List<Entry> entries = app.readEntries(file);
        displayDirectory(entries);
    }
}
```

Since code chunks belong inherently to different composition layers, their composition leads to chaotic amalgamations and ambiguous composition relationships. This limitation is caused by:

1. Heterogeneous compositional code structures. In OO programming the class structure is a container for the class members (variables and operations). Obviously, the heterogeneous structural nature of those members may lead to the following issues:
   a. First, the composition of such structures is done at free will by the developer (Humphrey, 2006). In other words, the composition of those structures is not governed by clear and well identified rules; therefore, it may lead to chaotically amalgamated composition which eventually renders the composition of those structures ambiguous and obscure. For instance, due to a variety of circumstances, developers might be tempted to add new instructions to operations or add new operations to a given component instead of using functionally equivalent operations belonging to other existing components. In such situations, not only the addition of instructions or operations leads to code duplication, but eventually it also makes other developers perplexed as to which implementation to use. In this respect, code duplication, a bad *code smell* (Fowler *et al.*, 1999), is the result of contextually misplaced instructions or operations caused by misplaced chaotically amalgamated and composedcode chunks. For example, the operation `readEntries` in Table 3.1 which amalgamates a set of instructions to read phone entry related data, might already have been constructed probably by another developer in a slightly different form as shown in Table 3.3. Often developers depending on their experience, in-house development guidelines and availability of development time, resort to refactoring as a means to reduce the severity of code duplication and the number of bad code smells (Fowler *et al.*, 1999). Nevertheless, refactoring is a voluntary activity which might simply be ignored since it does not affect the computational behavior of the application.

Table 3.4 Refactored operation

```
package acModel.quiteRefactored;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class FileUtils {
    public static BufferedReader getBufferedReader(File file)
                                  throws IOException {

        FileInputStream fileInputStream =
            new FileInputStream(file);
        InputStreamReader inputStreamReader =
            new InputStreamReader(fileInputStream);
        BufferedReader bufferedReader =
            new BufferedReader(inputStreamReader);
        return bufferedReader;
    }
}
```

b. Second, in OO programming operations members can not be composed in isolation of their container classes. In other words, OO oriented constructs do not provide enough constructs for reuse (Mcdirmid, Flatt and Wilson, 2001; Szyperski, 1998). As a result, to use those operations developers have either to use/inherit the whole component or duplicate the components' operations in another implementation. If the component is inherited, this might lead to side effects where unwanted component members are accessible without being needed (Al-Hatali and Walton, 2002; Msheik, Abran and Lefebvre, 2004). On the other hand, if the functionality is duplicated this leads to chaotic amalgamation of code chunks as illustrated for the operation `getBufferedReader` duplicated in the component of Table 3.3 and which already exists in the component shown in Table 3.4.

2. Inappropriate and inaccurate assignment of responsibilities to the appropriate composition level or element during the inception or evolvement of code elements. Unfortunately, when packages, components and operations are misplaced due to a variety of reasons, this leads to ambiguous amalgamation and composition of code chunks. To

remedy this situation, code restructuring and refactoring (Krishnamurthy, 1994; Mens and Tourwe, 2004) can be performed on the misplaced code elements. Furthermore, the use of various patterns (Gamma *et al.*, 1994; Larman, 2004) helps assigning appropriate responsibilities to corresponding code elements during software inception. Evolving and maintained code elements driven by new requirements or by new software reuse contexts, necessitate customization and adaptation. Nevertheless, inappropriate customization or adaptation of evolving code elements increases their complexity. The increase in complexity can partially be attributed to the chaotic amalgamation and composition of inappropriate customization and adaptation of those elements. Again, resorting to code refactoring and restructuring provides a remedy to such a situation. In both cases, (i.e. inception and evolvement of code elements), the success of the remedy depends on the extent of the refactoring and restructuring performed on those elements. Refactoring and restructuring of code elements comes at a cost. Typically, the extent of refactoring is governed by several factors among which are developer's will, experience and subjectivity. In conclusion, code refactoring and restructuring does not attain optimal levels in many cases.

3. Elastic component operation size in terms of instruction number. The variability of a component operation size, i.e. the ability to increase and shrink the instruction set of an operation while still preserving the consistency of its computational result is problematic. For instance, for a particular requirement one developer devises a component with ten behaviors (operations), another developer might provide the equivalent component functionality by writing five operations. In an extreme situation, a third developer might put all instructions in one monolithic operation resulting in a *code smell*. Analyzing the preceding scenario it can be observed that chaotically amalgamated code grows in direct proportion to the instruction set size of an operation. Consequently, it can be inferred that small size operations provide less obscure and ambiguous code as shown in the code of Table 3.3 which is a refactored version of the code shown in Table 3.1.

As a remedy to this limitation, we propose the AOC model which uses a homogenous structure to define both data and operation members. Traditionally in pure OOP, operations

structures cannot exist on their own, therefore, they are defined within the structure of their host component.

The use of a homogenous structure to define operation members enhances and facilitates the composition and reuse of those operation members individually. Giving operations members a homogenous structure similar to component's data members is inspired from the work on "functor objects" (Coplien, 1992; Odersky, 2009) and closures in functional programming languages. In functional programming languages, functions are first class citizen, i.e. they can be passed as parameters in the same way objects are passed in OOP.

3.4.3 Sub-optimal component reuse limitation

Even though software reuse has shown frequent success, nevertheless it has not delivered on its promises as expected (Mili *et al.*, 1999a; Mili *et al.*, 1999b; Shiva and Shala, 2007). Among the issues hindering software reuse are:

1. Identification of components is not a fluid and straightforward process. The absence of extensive, clear, detailed, up-to-date and precise documentation in addition to component search and identification tools is an obstacle to software reuse (Habermann, 1988; Mili *et al.*, 1999a; Nada *et al.*, 2000). More importantly, the absence of globally connected and searchable component spaces and repositories to look for and identify components hinders the identification of suitable components (Shiva and Shala, 2007).
2. A significant number of components are not reusable out of the box without further customization and adaptation. Component customization and adaptation to fit software requirements might not be simple and therefore might require extensive modification or wrapping to be done by the software constructor (Waldo, 1998). Often, this situation is a deterrent for software constructors and frequently lead s the software constructors to devise their own components from scratch. The extent of customization and adaptation on a component is impacted to a considerable extent by how much modular is the component itself.

The concept of software modularity pioneered by Parnas (Parnas, 1972) is a synonym of high cohesion and low coupling (Laplante, 2007). Modularity has been advocated since a long time as a means to achieve better software maintainability, reliability and especially reuse (Cai and Huynh, 2007 ; Hitz and Montazeri, 1995; Zhao and Xu, 2004). Even though OOP languages, and by extension COP (component oriented programming) languages exhibit better software modularity compared to procedural programming languages (Ferrett and Offutt, 2002), unfortunately OOP and COP languages still suffer from inappropriate assignment of code responsibilities to classes and components. Inappropriate assignment of code responsibilities to classes and components yields chaotically amalgamated code which is translated into less modular components and eventually to less software reuse. Several reasons stand behind the creation and existence of less modular software elements:

1. Cost of software modularity: Software modularity is costly to achieve in terms of resources and time allocated to the development of modular components.
2. Difficulty to capitalize on the value of modularity (Sullivan *et al.*, 2001): Software is constructed using agile methodologies favors short term software development iterations with little reliance and adoption of best practices (Dinakar, 2009). Industrial software vendors to some extent take advantage from current status quo since it permits them to roll-out periodically newer component versions at relatively lower costs and consequently position their products against their competitors and thus to harvest profits from the situation. Similarly, for in-house developed software, management decisions aim to reduce costs most of the time. Therefore, unless being parts of APIs, fewer in-house developed components pass through a modularization process for potential reuse into other projects within the same enterprise.
3. Loose development rules: Development rules put in place by organizations are often loose leaving the decision to construct optimally modular components entirely in the hands of developers.
4. Developer experience: If the budget and time-to-market allow it, experienced developers tend to deliver modular code.

5. Developer subjectivity: The extent of modularity can be impacted by the developer subjectivity. For instance: for one developer, a certain degree of modularity is satisfactory, while it might not be the case for another one.
6. Modular component are difficult to construct: Due to the numerous iterations needed to deliver the software complying with the requirements, software component are difficult to construct especially at design time (Sullivan *et al.*, 2001).

Optimal software reuse is an ultimate general goal of software engineering. Optimal software component reuse *per se* can be thought of as the ability to reuse software components with a minimum of customization and adaptation. Attaining optimal software reuse is an extensive task which necessitates the satisfaction of a variety of requirements. Even though developing coherent, consistent and modular software applications by means of design patterns and *code smells* refactoring lead to a better modular code and increased software reuse, neither of those two techniques has optimal modularity and consequently optimal software reuse as objectives to attain.

Attaining optimal component reuse necessitates achieving optimal modularity, i.e. optimal cohesion and coupling. In this respect a particular code element is considered optimally modular if:

1. It constitutes an atomic code element and equally provides an atomic behavior. An atomic code element is an element which cannot be divided into one or more atomic code elements without breaking the atomic behavior it provides.
2. It is recursively composed of a strict minimum of optimally modular code elements and it cannot be further divided into more than one composite code elements without breaking the behavior it provides. In other words, it can only be composed out of optimally modular composite code elements or atomic code elements.

Equivalently, a component is considered as being optimally modular if:

1. It constitutes by itself an atomic component and equally provides an atomic behavior. An atomic component is considered to be a code element which cannot be divided into one or more atomic components without breaking the behavior it provides.
2. It is recursively a composite component and is composed of a strict minimum of other optimally modular composite or atomic components.
3. It cannot be further divided into one or more composite components without breaking the behavior it provides.

Observing the code in Table 3.2 and Table 3.5 gives insights and helps understand the concept of optimal component modularity. The code in Table 3.2 shows a phone directory component `Entry` which has been further modularized to result in an optimal component composed out of two optimally modular components as shown in Table 3.5. Obviously, the setter and getter operations have been removed for the sake of clarity and ease of comprehension.

Table 3.5 Optimally modular component Entry

```
public class Entry {
    private  Phone phone;
    private Person person;
}

public class Phone {
    private Integer internationalCode;
    private Integer areaCode;
    private Integer localNumber;
}

public class Person {
    private String firstName;
    private String lastName;
}
```

Current development processes and practices aspire to, but do not seek to, achieve optimal software component modularity. Hence, attained modularity levels are achieved quite often by coincidence. In addition to the reasons mentioned above hindering software from being

modular, the lack of models which govern, guide and lead to the construction of optimally modular components restrains components from attaining optimal modularity.

3.4.4 Component versions mismatch limitation

Software components interfaces are used for inter-component communication in addition to being used as a means to reduce coupling. Interfaces are distinguished by their names and a set of signatures defining the input and output to those interfaces. Essentially, component interfaces act like an empty shell structure for components while components are actually implementations, i.e. both the shell and the flesh inside of it. From a programming language viewpoint, both interfaces and their implementing components are essentially "types". During their operational life cycle, components and their interfaces are subject to modifications and upgrades. Even though components abide by the contract dictated by their respective interfaces, incompatibilities may arise and thus may cause faulty behavior and probably loss or corruption of data. The following two scenarios show how such incompatibilities can be produced.

1. Newer component versions may provide incompatible behavior. When a new component implementation is constructed and deployed, the contract with its interface is not enough to prevent an erroneous behavior from happening. For instance, the `TaxSystem` component in Table 3.7 uses the `TaxCalculator` component shown in Table 3.8. In turn, the `TaxCalculator` component implements the `TaxCalculatorI` interface defined in Table 3.6. Assuming that at a certain point of time, a new version of the TaxCalculator (see
2. Table **3.9**) is made available for the `TaxSystem` component. Then, the calculation of sale tax using this new implementation will be erroneous.

Table 3.6 Definition of the TaxCalculatorI interface

```
public interface TaxCalculatorI {
     public double calculateTax(double amount);
}
```

What is striking in this scenario is that the new implementation of the `TaxCalculator` component, as its predecessor, abides by the contract dictated by the `TaxCalculatorI` interface, but the actual behavior leads to different results. In conclusion, this illustrates that even though an interface definition has not changed, different implementations of this interface might cause faulty behaviors.

3. Newer component implementation having dependencies on third party components unavailable for an older version. For demonstration purpose, the `Logger` component is used as a third-party component by the second new implementation of the `TaxCalculator` interface shown in Table 3.10. Being a third-party component, the `Logger` component may not be available to the `TaxSystem` component, thus causing a faulty behavior during run time (The `Logger` component is available at compile time, but not at runtime).

Table 3.7 Component using an implementation of the TaxCalculatorI interface

```
public class TaxSystem {

     TaxCalculatorI taxCalculator = new TaxCalculator();

     public double calculateTax(double amount) {
          return taxCalculator.calculateTax(amount);
     }
}
```

Table 3.8 First implementation of the TaxCalculatorI interface

```
// old component implementation
public class TaxCalculator implements TaxCalculatorI{

    public double calculateTax(double amount) {
         return amount * 0.2;
    }
}
```

Table 3.9 Second implementation of the TaxCalculatorI interface

```
// new component implementation
public class TaxCalculator implements TaxCalculatorI {

    public double calculateTax(double amount) {
      return amount * 0.1;
    }
}
```

Table 3.10 Third implementation of the TaxCalculatorI interface

```
// another new component implementation
public class TaxCalculator implements TaxCalculatorI {

    public double calculateTax(double amount) {
        Logger log = new Logger();
        Log.info("calculating tax using third implementation");
        return amount * 0.1;
    }
}
```

3.5 Conclusion

In this chapter, the methodology used to pursue this research project was presented. Equally, In particular: the component unwanted members, the chaotic amalgamation and composition of code chunks, the suboptimal software component reuse and the component versions mismatch.

CHAPTER 4

MEASURING COMPONENT UNUSED MEMBERS

4.1 Introduction

Software components have emerged as an important paradigm to address several traditionally known problems such as complexity, reuse and reduction of software development costs (Li, 1999; SEI, 2003). Although the software component technology has been undergoing continuous enhancements, it still suffers from a number of limitations among which is the presence of components' unused members.

To illustrate this limitation, it is helpful to observe what happens during the process of software application construction. To construct a particular application, a set of existing components can be used. Each of those components incorporates a set of members of which a size-varying subset is actually used to satisfy the functional requirements of the application. Consequently, a subset of unused members will persist in the deployed application. This subset of unused members provides no functional value to the hosting application. Furthermore, those unused members consume memory and network resources and might compromise application security or integrity if they are exploited inappropriately or inadvertently. Furthermore, they contribute to application bloating and code amalgamation.

Among the reasons that lead to the existence of components' unused members is that software components have a tendency to be constructed in coarse to large-grained sizes. In other words, there exists a direct relationship between the size of a component and the number of unused members it possesses. Nevertheless, the set of useful and required members provided by a particular component varies according to the particular software application context. Typically, during the construction of software application families, considerable effort is expended on the wrapping, adaptation and customization of the functionalities of components shared by the various constituent applications.

To tackle the presence of components' unused members, it is quite interesting to assess the extent of their effects. One way to perform such assessment is by measuring their number and the size of the memory they consume and occupy. For this purpose, the CUMM (Components Unused Member Measurement) method is developed in this research project to:

1. Measure components' unused members (attributes and behaviours);
2. Measure via ad-hoc statistical formula percentages of unused members and their memory consumption on a per component and a per application basis;
3. Measure the degree of a component members' generality.

4.2 Reference model used to develop the CUMM method

Software measurement can help evaluate software quality attributes and in making better decisions and controlling software and its development process. In this respect, IEEE software engineering standard (Institute of Electrical and Electronics Engineers, 1990) defines software engineering as:

> "The application of a systematic, disciplined quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software" (Institute of Electrical and Electronics Engineers, 1990).

Software measurement theory allows for the mapping of the empirical domain to a numerical domain. In other words, software objects are measured in terms of quantitative values. Several software measures have been proposed over the years, but the problem lies in the way the empirical quantitative values of those measures are mapped to semantically meaningful qualitative values (Zuse, 1997). Research efforts have been concentrated on providing sound models, methodologies and measurement frameworks for the design and use of software measures so that these measures are established for a software engineering

discipline in a similar way to the measures used in other engineering disciplines (Jacquet and Abran, 1997; Zuse, 1997).

Traditionally, and by analogy to the way other engineering disciplines define measurement methods, several software measurement methods have been defined in the form of mathematical formulas that lead to the calculation of numerical values. The results of calculations are then used in various types of models for evaluation and decision-making purposes. It is observed that few of these measurement methods have been defined according to well-defined measurement processes.

Therefore, to develop the CUMM method on a sound basis, the measurement process model suggested in (Jacquet and Abran, 1997) and illustrated in Figure 4.1 is used. This high-level model sets up a four-step road map to be used in the design and validation of software measurement methods. The first step requires the:

> "definition of the measurement method objectives, design and selection of the metamodel for the objects to be measured, the characterization of the concepts to be measured, and the definition of the numerical assignment rules" (Jacquet et Abran, 1997, p 130).

The second step requires the construction of the metamodel using the appropriate software documentation and the application of the measurement method to calculate the resulting numerical values. The third step requires the analysis, documentation and auditing of the measurement result. In step four, the actual exploitation of the measurement result will be carried out.

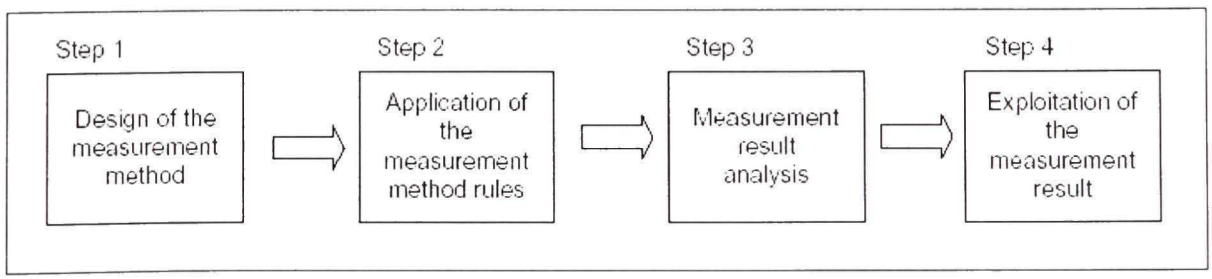

Figure 4.1 Measurement process – high-level model.
(Jacquet and Abran 1997, p 129)

4.3 Overview of the CUMM method

The CUMM method applies to software entities which are components constructed using object-oriented programming languages. An entity measured by the CUMM method can be:

1. A complete software application treated as a whole and single component;
2. A subsystem component;
3. An ordinary object-oriented class considered in the context of the CUMM method as whole component.

The CUMM method measures the static memory consumption of the elements of an entity. Put differently, the CUMM method does not calculate the memory consumed by an entity's dynamic objects created at runtime. In this respect, an entity's memory consumption is considered as the recursive summation of the memory consumed by the binary code of the entity and its aggregate classes in a recursive manner.

4.4 Development of the CUMM method

Based on the measurement process model shown in Figure 4.1, the development of the CUMM method is carried out according to the following steps:

4.4.1 Step 1: Design of the CUMM method

The design of the CUMM method follows 4 substeps:

Substep 1: Definition of the objectives. The objectives of the CUMM method are to measure, within an enclosing entity context: i) the number of the software entity's unused members (attributes or functionalities), and ii) the memory consumption of the entity's unused members. This enclosing entity context might be an outer component context or an application context. In the CUMM method context, an application is an aggregation of one or more components associated recursively with each others. The application and its aggregate

components can be considered as one component per se. The measurement of unused members of a component within an enclosing entity context refers to the unused members of the component itself and the unused members of its nested component set in a recursive manner. It is important to note that a component's inherited members are actually implicit members of that component, and therefore they are all treated uniformly by the CUMM method. The intended users of this method are developers, architects and project managers; however, other stakeholders can use the measurement method results for control and decision-making purposes.

Substep 2: Design and selection of the metamodel. The CUMM method must permit the measurer to measure an entity's unused attributes and functionalities and their memory consumption in a quantifiable manner. As suggested by the measurement process model shown in Figure 4.1, to measure an entity, a metamodel of that entity must be designed or selected. A CUMM measurable entity can be instantiated according to the generic entity metamodel given in Figure 4.2. As depicted, this metamodel does not necessarily imply the real physical or logical composition relationship of an application and its component set. In practice, applications are aggregates of components, which in turn can be aggregates of other nested components.

Substep 3: Characterization of the concept to be measured. The measurement of unused attributes or functions of an entity is calculated based on the measurable subcharacteristics of the measured entity. The generic metamodel shown in Figure 4.2 characterizes the members of a measured entity on two bases: use basis and memory consumption basis. For instance, an entity member can be exclusively either used or unused. Similarly, an entity member consumes memory resources whether used or unused. A component member or functionality is considered unused if it has never been referenced, either in the code of its enclosing entity context or in the code of its nested and aggregate components.

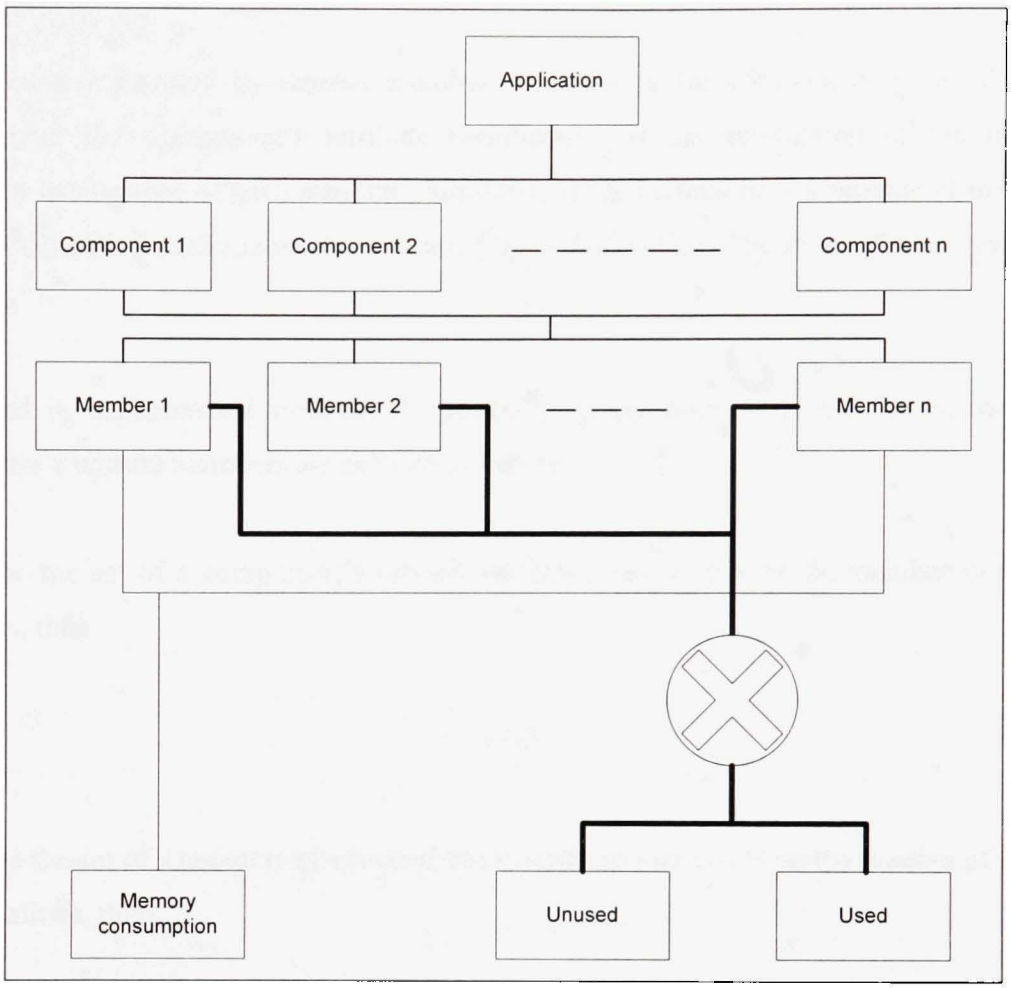

Figure 4.2 Generic metamodel representation of the application and component entities.

Substep 4: Definition of the numerical assignment rules. The numerical assignment rules permit the calculation of: a) the number of unused members of a component, and b) the memory occupied by these unused members.

The number of unused members (attributes or functions) is calculated by counting the number of times each member type is referenced within the code of the measured entity context. Obviously, unused members will have zero reference value. The number of members having a zero reference value is effectively the number of unused members. According to the component member being measured, the unit of the measurement result is "attribute/per component", denoted "ac", or "function/per component" denoted "fc".

The memory consumed by unused members of a particular component is calculated as follows. For the component's attribute members, it is the summation of the memory consumed by the type of each attribute. Similarly, for the component's functional members, it is the summation of the memory consumed by each function. The measurement result unit is a "byte".

Expressed in mathematical formula, the numerical assignment rules for the number of a component's unused members are defined as follows:

Let A be the set of a component's unused attributes and $u_a \in \mathrm{N}$ be the number of unused attributes, then

$$u_a = |A| \tag{4.1}$$

Let F be the set of a component's unused functionalities and $u_f \in \mathrm{N}$ be the number of unused functionalities, then

$$u_f = |F| \tag{4.2}$$

Similarly, the numerical assignment rules for the memory consumed by unused members are defined as follows: Let m_{ai} be the memory consumed by the i-th unused attribute reference of a component. Then, t_{ma} is the total memory consumed by the unused attribute elements in A and is calculated as

$$t_{ma} = \sum_{i=1}^{|A|} m_{ai} \tag{4.3}$$

Let m_{fi} be the memory consumed by the i-th unused functionality of a component. Then, t_{mf} is the total of memory consumed by the unused functionality elements in F and is calculated as

$$t_{mf} = \sum_{i=1}^{|F|} m_{fi} \tag{4.4}$$

4.4.2 Step 2: Application of the CUMM method

The application of the measurement method requires three substeps:

Substep 1: Gathering the software documentation related to the entities subject to measurement. The documentation can be either the source or reflective binary code of (a) the entity outer context, (b) the entity itself, and (c) the entity's nested entities.

Substep 2: Constructing the software model by instantiating the generic metamodel. The software model is constructed by instantiating the generic metamodel in Figure 4.2 taking as input the documentation artifacts gathered in substep 1. The constructed model leads to the identification of the measurable characteristics of the entity subject to measurement.

Substep 3: Applying the numerical assignment rules. The application of the numerical assignment rules makes it possible to assign quantifiable values to the measurable characteristics of the entity subject to measurement, and eventually to calculate the measurement results.

4.4.3 Step3: Derived statistics of the measurement result

When the results are ready, they must be documented in a presentable format. Furthermore, different types of analysis and derived statistics can be calculated out of the result. For

instance, the following derived statistics for the CUMM method are potentially useful to provide value to the user of the CUMM method:

1. Measuring the percentage of unused attributes of a component: This is done by using the result calculated by applying the measurement method on unused attributes and by calculating the total number of attributes.
2. Measuring the percentage of unused functionalities of a component: This is done by using the result calculated by applying the measurement method on unused functionalities and by calculating the total number of functionalities.
3. Measuring the percentage of unused attributes' memory consumption of a component: This is done by using the result calculated by applying the measurement method to measure the memory consumption of unused attributes and by calculating the total memory consumed by used attributes.
4. Measuring the percentage of unused functionalities' memory consumption of a component: This is done by using the result calculated by applying the measurement method to measure the memory consumption of unused functionalities and by calculating the total memory consumed by used functionalities.
5. Measuring the degree of a component's attribute generality of a component: This is done by measuring the degree of a component's attributes generality with respect to the set of applications that makes use of this component. The result of this measure depends on the percentage of unused attributes of a component and on the number of applications which use the component. The degree of a component's attribute generality is a percentage calculated by the summation of the percentages of a component's unused attributes in each application where the component is used, and then dividing the summation result by the number of these applications.
6. Measuring the degree of a component's functional generality: This is done by measuring the degree of a component's functional generality with respect to the set of applications that make use of this component. The result of this measure depends on the percentage of unused functionalities of a component and on the number of applications which use the component. The degree of a component's functional generality is a percentage calculated by the summation of the percentages of a component's unused functionalities in each

application where the component is used, and then dividing by the number of these applications.

4.4.4 Step 4: Exploitation of the result

Finally, the results can be exploited to exercise the desired control and to make appropriate decisions. The results give indicators to the users of CUMM so that appropriate actions based on objective observations can be taken. For instance, unwanted members can be removed from the respective components to reduce the application memory foot print.

4.5 Example

To illustrate the applicability of the CUMM method, a simple dummy application example is presented to illustrate the usage and applicability of the CUMM method. In particular, the example application is used to illustrate the application of step2 (application of the method) and step3 (illustration of the derived statistics) of the CUMM method. The example application consists of three classes `ClassA`, `ClassB` and `ClassC` where `ClassA` contains `classB` and `ClassB` contains `ClassC` as shown in Table 4.1. In the context of the CUMM, this application can be considered as a single component.

Step 2: Application of the CUMM method

Assumptions

Simplifying the computation in this example is important since the computation is done manually. For this reason the following assumptions have been set and used in the computation process:

1. A component member count of code lines are considered to be equivalent to its memory consumption.

Table 4.1 Simple dummy application

```
public class ClassA {
    int bint1 = 1;
    ClassB classB1 = new ClassB();
    ClassB classB2 = new ClassB();
    public void am1() {
        bint1 = (classB1.getBint1() + classB1.getBint2());
    }
    public void am2() {
        am1();
    }
}

public class ClassB {
    private int bint1 = 1;
    private int bint2 = 2;
    private ClassC classc;
    public int getBint1() {return bint1;}
    public void setBint1(int bint1) {this.bint1 = bint1;}
    public int getBint2() {return bint2;}
    public void setBint2(int bint2) {this.bint2 = bint2;}
}
public class ClassC {
    int cint1;
    public void cm1() {}
}
```

2. Every attribute or function code instruction consumes one byte, which is not true in real life since the actual memory consumption depends on the type of attribute and machine instruction each line of code represent in terms of binary code.
3. When counting a component's lines of code, empty lines and comments are ignored.

Substep 1: This step requires the gathering of documentation artifacts to be used during the measurement process. In this respect, Table 4.1 is the artifact used in this step. The `ClassA` application contains references to two components: `ClassA` and `ClassB`. Therefore, to apply the CUMM method, the source or binary code of those two components is needed. Similarily, `ClassB` contains references to two componets: `ClassB` and `ClassC`. Consequently, the source or binary code of those two components must be present to conduct the measurement.

Substep 2: This step requires the construction of the software model by instantiating the generic metamodel shown in Figure 4.2 via the use of the documentation artifact gathered in substep 1 as input. The software model for the `ClassA` application shown in Table 4.1 is illustrated in Figure 4.3. To characterize the memory consumption of an unused member, in this example, we will not compute the memory consumed by each member, since this is a complex process and requires analysis of the class binary code, a task which is better done by an automatic tool. Instead, for each method we will use the number of Java code lines to give us an approximate indication of the amount of memory consumed by unused members.

Substep 3: To obtain the measurement results, we use the information presented in Figure 4.3 and Table 4.2. The numerical rules have to be applied for each of the components. The measurement results are calculated for the `ClassA` component only. The measurment results for the components `ClassB` and `ClassC` follow a similar pattern:

- **`ClassA` component:** This component contains 2 functions and 3 attributes. In this example, the method `am2` is assumed to play the role of main method (application starter application in Java) and therefore it is considered as a used member. In the context of the CUMM method this method must be considered as an unused member since it is not referenced by any other member. One of the attributes is primitive and the other two refer to the same component, i.e. `ClassB`. As a result this nested component has to be analaysed and included in the measurement as a component.

- `ClassB` **component:** This component contains 4 functions and 3 attributes. Two of the attributes refer to a primitive type and the third refers to the `ClassC` component which in turn has to be included in the measurement as a component.
- `ClassC` **component:** This component contains 1 method and 1 attribute. The attribute is a primitive and therefore need not to be analysed as a component. As can be observed from the code of Table 4.1 none of the members of this component are used by the example application.

The count of unused attributes is calculated in terms of attribute per component unit (ac) for component `ClassA` and its nested ones to be:

$$\boxed{u_a = 2} \tag{4.5}$$

The number of unused functionalities is calculated in terms of functions per component (fc) to be:

$$\boxed{u_f = 3} \tag{4.6}$$

The total memory in bytes consumed by the unused attributes is calculated to be:

$$\boxed{t_{ma} = \sum_{i=1}^{|A|} m_{ai} = 2} \tag{4.7}$$

Based on the assumptions mentioned above, the total memory in bytes consumed by the unused functionalities is calculated to be:

$$\boxed{t_{mf} = \sum_{i=1}^{|F|} m_{fi} = 5} \tag{4.8}$$

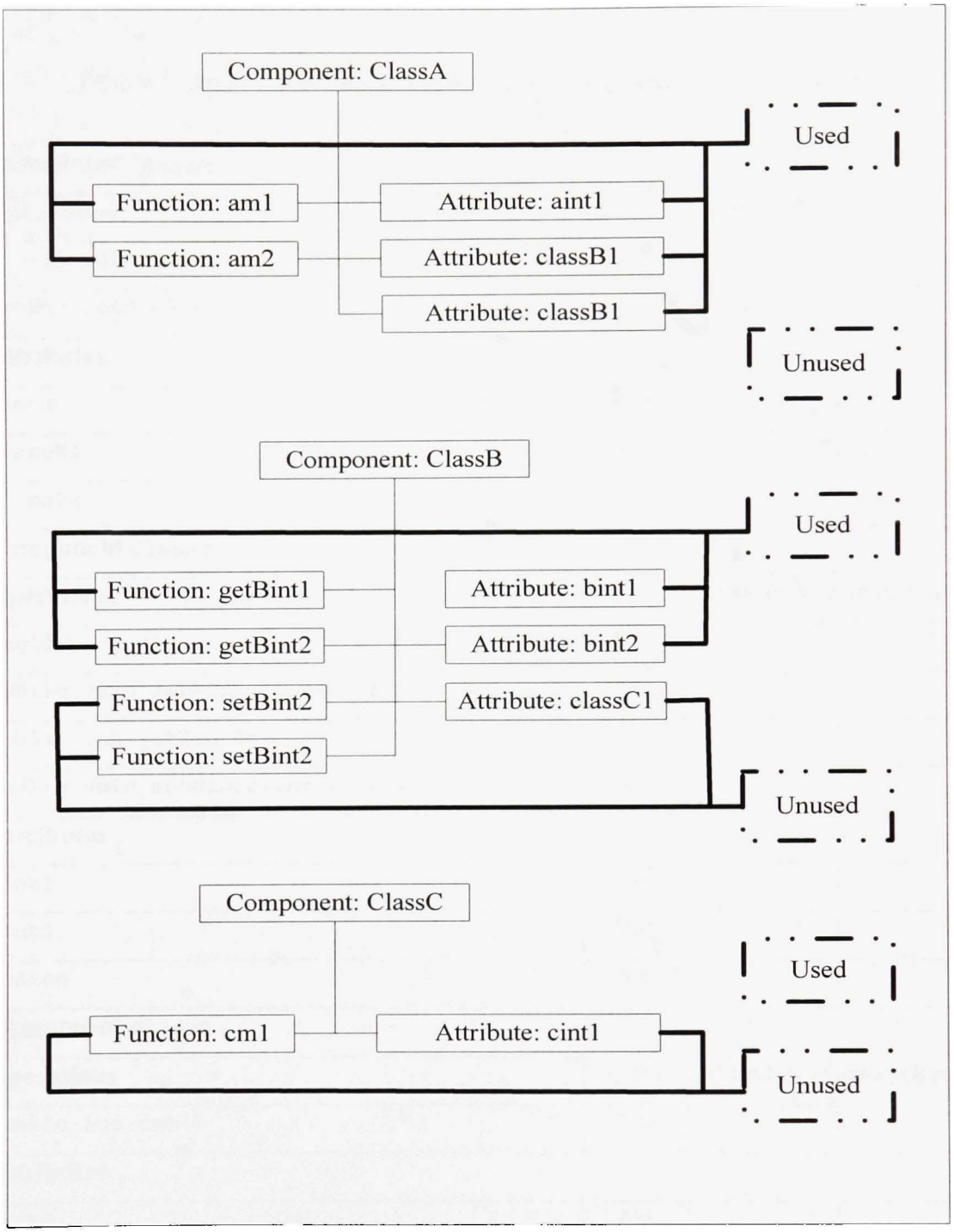

Figure 4.3 Metamodel instance for the simple dummy application.

Table 4.2 Application component members' usage and lines of code count

Component `ClassA`		
Operations	Used	Number of instructions
`public void am1()`	Yes	2
`public void am2()`	Yes	2
Attributes		
`aint1`	Yes	1
`classB1`	Yes	1
`classB2`	Yes	1
Component `ClassB`		
Operations	Used	Number of instructions
`public int getBint1()`	Yes	2
`public void setBint1(int bint1)`	No	2
`public int getBint2()`	Yes	2
`public void setBint2(int bint2)`	No	2
Attributes		
`bint1`	Yes	1
`bint2`	Yes	1
`classc`	No	1
Component `ClassC`		
Operations	Used	Number of instructions
`public int cm1()`	No	2
Attributes		
`bint1`	No	1

Step 3: Calculation of derived statistics based on the measurement results of the example application as represented by the `ClassA` component:

1. Percentage of unused attributes of the `ClassA` component is equal to:
   (2 unused attributes) / (7 total attributes) * 100 = 28.57%.
2. Percentage of unused functionalities of the `ClassA` component is equal to:
   (3 unused operations) / (7 total operations) * 100 = 42.85%.
3. Percentage of unused attributes memory consumption of the `ClassA` component is equal to:
   (2 unused bytes) / (7 total bytes) * 100 = 42.85%.
4. Percentage of unused functionalities memory consumption of the `ClassA` component is equal to:
   (6 unused bytes) / (14 total bytes) * 100 = 42.85%.
5. Degree of the `ClassA` component's attribute generality cannot be calculated since the `ClassA` component is not used in the context of other components in this example.
6. Degree of the `ClassA` component's functional generality cannot be calculated since the `ClassA` component is not used in the context of other components in this example.

4.6 CoMet: a CUMM automated measurement tool

4.6.1 Overview

Automation of the CUMM would reduce considerably the measurement effort that would otherwise be expended had the measurement process been performed manually. Consequently, CoMet (Component Measurement) has been created to automate the CUMM measurement method. CoMet takes as input a component or an application binary code. Then, it performs measurement tasks such as data collection, analysis and calculation based on the input code. Finally, it reports the measurement results. The automation process of the CUMM method is illustrated in Figure 4.4. CoMet has been developed as a Java prototype tool to automate the CUMM measurement process. CoMet is an experimental tool developed to measure components written in Java.

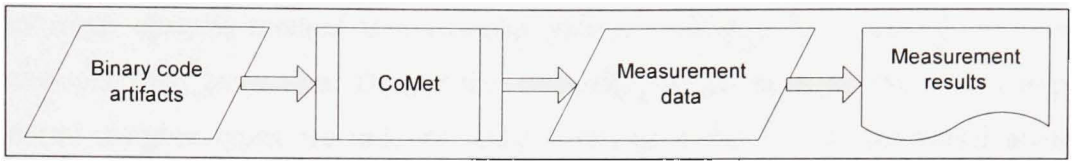

Figure 4.4 CoMet measurement process.

4.6.2 CoMet realization

The current version of CoMet takes as input only the binary code artifacts of a software component to calculate the measurement results. There are several reasons behind this:

1. For certain components, especially proprietary ones, the only code artifact available for measurement is the binary code;
2. It is relatively much easier to conduct measurement activities on the reflective binary code than on the source code, since the former is cleaner and more condensed;
3. It takes less effort to develop the tool to measure binary code instead of source code especially that a number of open source tools can be used to reduce the development effort.

Internally, CoMet is developed as a collection of Java classes which depend on BCEL (Byte Code Engineering Library), an apache open source library specialized in manipulation of Java byte code (Apache, 2001).

Moreover, CoMet is equiped with a GUI interface through which the user specifies the component to be measured and observes the measurement results reported when the calculation has been terminated (see Figure 4.5). Additionally, the application binary code of the component to be measured must be available on the classpath so that CoMet can perform the measurement properly. As output, it generates a report detailing the measurement results. The information presented in the report shows the number of unused members (attributes and methods) and the percentage of their usage, as shown in Figure 4.5.

CoMet input analysis method is a two-part process – binary code analysis followed by measurement data extraction. During the execution of the first process, the component referenced member types are indexed using a recursive depth-first transversal algorithm. When the recursive algorithm terminates, the analysis of member methods begins. The analysis of member method results generates a call graph and collects relevant measurement data. The second major step consists in:

1. Transforming the measurement data collected into meaningful measurement information through the application of the CUMM calculation formula;
2. Presenting the measurement results to the user.

4.6.3 CoMet in action

To illustrate the features of CoMet, a simple test application example is measured. This application contains a component consisting of the class ShowWelcomeMessage with one data member and one method member.

As shown in Figure 4.5, CoMet calculated the total number of methods used by this component as equal to 869. Obviously, the number of methods comes from:

1. The inherited methods from the Object father class of all Java classes;
2. The methods of all the classes (String, System, etc.) used by the ShowWelcomeMessage;
3. Recursively, the methods inherited by those classes used by the ShowWelcomeMessage component.

The ShowWelcomeMessage component in its current state makes use of 448 methods. Expressed as a percentage, the ShowWelcomeMessage component makes use of only 51.55 percent of the loaded methods. As for the number of data members of the ShowWelcomeMessage component, the same argument holds as for the method members.

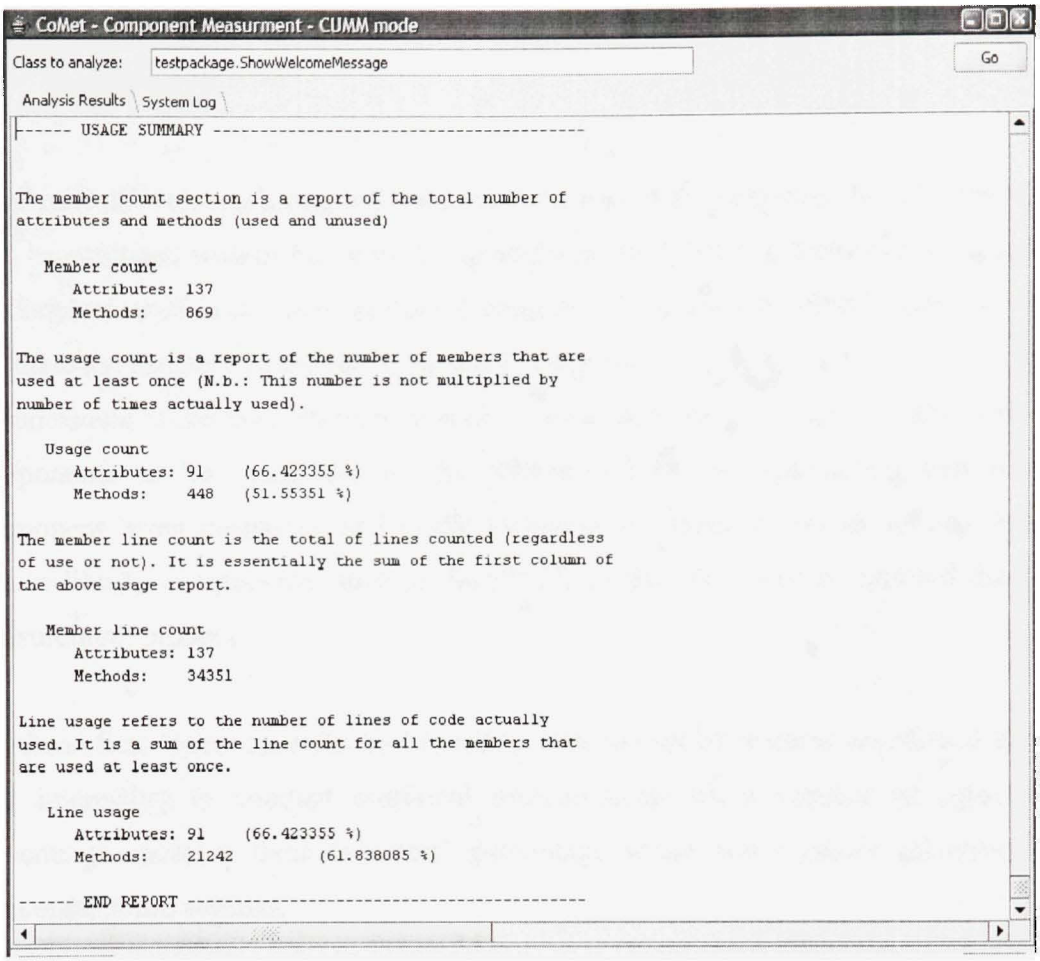

Figure 4.5 CoMet GUI.

4.6.4 CoMet limitations

CoMet current implementation has a limitation related to component members used within the body of the component's functions. If any of the component's members are used solely within the body of a member method, they are not taken into consideration when performing the measurement. Consequently, measurement results are not completely accurate in such circumstances. The development of CoMet is meant to provide a proof of concept. Nevertheless, the measurement results still provide a relevant indicator of the number of a software component's unused members and their usage in a software application. The implementation of this prototype is provided in Appendix I on a CDROM.

4.6.5 Future work

On the practical level, producing accurate results is one of the important features that CoMet needs to be equipped with in future work. In addition, the following features can be added:

1. The implementation of a new module to analyze a component member's byte code and to calculate the memory consumed by unused members.
2. Enhancement to the user interface in such a way to give the user more control over: a) the components to be measured, b) the path-setting of the application that uses the component being measured, and c) the inclusion or exclusion, on an optional basis, of certain library components, such as Java API, so that they will be ignored during the measurement process.

On the theoretical level, once the tool is ready with the set of features mentioned above, it will be interesting to conduct empirical measurements on a number of open source components to measure their members' percentage usage and memory consumption in different application settings.

4.7 Conclusion

In this chapter, the CUMM measurement method was presented with a set of derived statistics to analyze the measurement results which pertain to aspects related to the unused attributes and functionalities per component and per application. Furthermore, CoMet, the automatic tool to conduct the measurement of software components' unused members is presented.

The results calculated by CoMet according to the CUMM method have a cross-cutting impact on a number of (ISO/IEC, 2001) quality characteristics and subcharacteristics. For instance, unused functionalities could indicate that the security subcharacteristic of applications making use of these functionalities might be an issue. Similarly, the memory

consumption by unused attributes and functionalities indicates in turn an impact on the efficiency characteristic. In the same vein, the maintainability characteristic is impacted by the number of unused functionalities as to the effort that might be required when adapting and customizing the concerned software components.

CHAPTER 5

COMPOSITIONAL STRUCTURED COMPONENT MODEL (CSCM)

5.1 Introduction

The unwanted functionalities exhibited by software components in particular application contexts are caused by the tendency of the size of software components to be coarse to large-grained. In such components, the set of useful and required functionalities provided by a particular component varies according to the particular software application context. Typically, during the development of software application families, considerable effort is expended on the wrapping, adaptation, and customization of the functionalities of components shared by the various constituent applications. In this chapter, the Compositional Structured Component Model (CSCM) is proposed as an initial model to:

1. Handle the issue of unwanted component functionalities; and
2. Provide an easier and more flexible approach for software customization, adaptation, and reuse.

The idea behind this model is to develop components with physically disjoint functional fragments called compositional interfaces. At composition time, the application developer selects the functionality fragments needed to form the basis for a new software component.

5.2 Compositional Structured Component Model

The Compositional Structured Component Model (CSCM) (see Figure 5.1) is designed to construct software components through selective functional composition based on component metadata composition descriptor instances. This selective composition property provides flexibility for the adaptation and customization of components, as well as for facilitating software maintenance and helping to more readily achieve software reuse.

The CSCM model can be seen as an extension to the object oriented model which provides compositional capabilities. Consequently, a CSCM component instance is an object with an

enhanced capability allowing selective functional composition of disjoint compositional parts.

In the context of the CSCM model a compositional part is a method implemented independently outside of the component implementation. We call such a part a compositional interface for being independently implemented and physically disjoint from the component's implementation. CSCM component's compositional interfaces perceived as methods from within a component implementation unlike Java interfaces which are types *per se*.

Table 5.1 Reserved words of CSCM components

Reserved words	**Role**
component	Declares the beginning of a CSCM component
compositional	Declares a compositional interface member

The proposed CSCM model is generic and can be implemented in various programming languages. From an object oriented perspective, CSCM components instances can be considered as objects since they possess and exhibit similar properties and characteristics to those of objects. CSCM components support inheritance; however, they are provided with a powerful composition and retrogression mechanism which allows CSCM component to either include or exclude compositional interfaces according to the information provided by the component metadata instance. The CSCM mechanism of composition and retrogression is based on:

- An extension to the syntax of an object oriented programming language to support compositional members as presented in Table 5.1;
- The use of the composition principle to selectively include the required functionalities suitable for a software application;

- The use of the delegation principle which permits the dispatching of the component methods' calls to the compositional interfaces of the component.

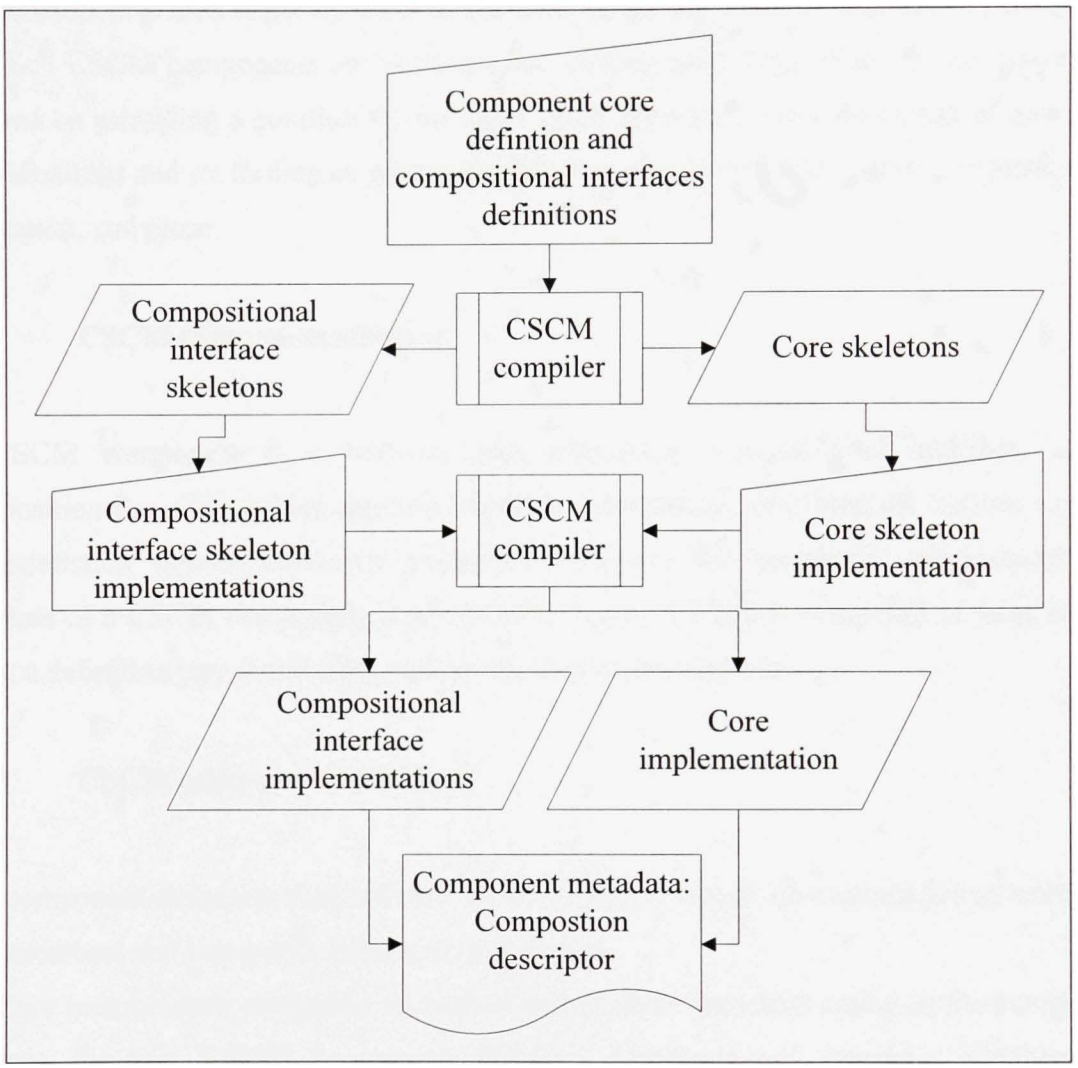

Figure 5.1 CSCM component construction process.

A CSCM component compositional interface differs from CSCM ordinary methods in that they are selectable via the component metadata composition descriptor instance. In other words, the same component in two different applications might have different subsets of compositional interfaces, depending on the functional requirements needed by the hosting application.

The CSCM model does not handle the aspects of distribution, synchronization, and interoperability. Therefore, CSCM components rely for these aspects on the underlying mechanisms provided either by other component models or by host programming languages in which CSCM components are implemented. Indeed, the current scope of our research is focused on providing a solution for the issue of components having an excess of unwanted functionalities and on finding an easier approach to components composition, customization, adaptation, and reuse.

5.3 CSCM component structure

A CSCM component is a software part possessing compositional interface, and a composition descriptor which captures metadata information specifying the various aspects, characteristics, dependencies and properties necessary for functional composition. The structure of a CSCM component is presented in Figure 5.2 and is composed of three logical parts: a definition part, a metadata part and an implementation part.

5.3.1 CSCM component definition

The component definition part includes the definition of two distinct categories of members: core members and compositional interface members.

1. Core members are composed of method and attribute members acting as the component core for any CSCM component instance. Compositional interface members are selectively available for composition through CSCM component instances.
2. CSCM components instances behave almost like objects. Without their compositional members (interfaces) CSCM component instances are indistinguishable from ordinary objects.

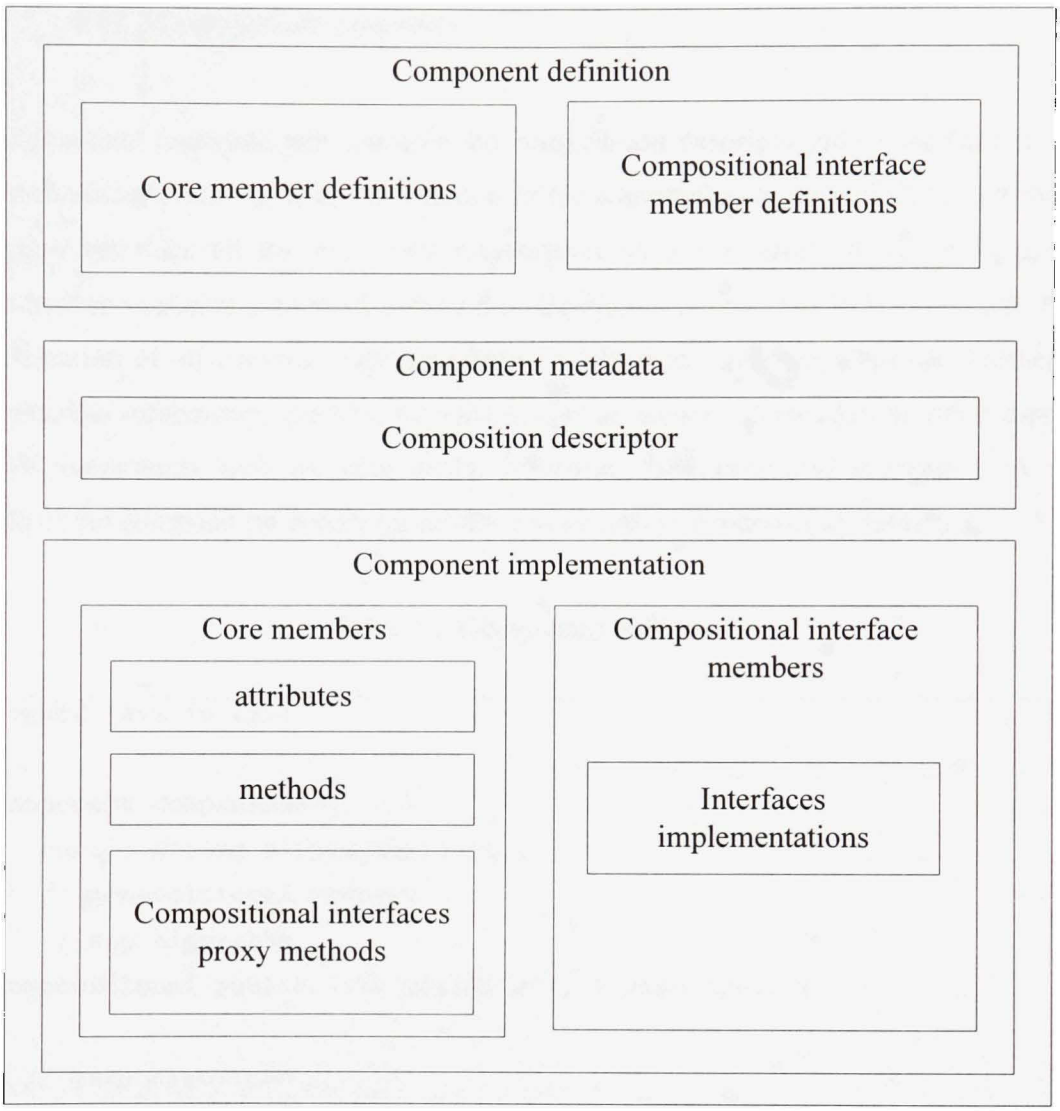

Figure 5.2 CSCM component structure.

The definition of a CSCM component has to be done using the host programming language in addition to the syntax constructs of Table 5.1. The definition of core members and the definitions of compositional interface members are necessary to generate the component core skeleton as well as the compositional interface skeletons.

As illustrated in Table 5.2 a CSCM component definition has to be defined in a file similar to the way object oriented source classes are defined. This definition defines the component "Compressor" with one core method and two compositional interfaces.

5.3.2 CSCM component metadata

The component metadata part contains the composition descriptor which captures in XML format the descriptive metadata information of the component. In particular, the composition descriptor provides all the necessary information on a component's members and their dependencies so that at composition-time the selection of an interface will also result in load-time selection of all compositional members on which the interface depends. Furthermore, this metadata information can also be used to specify useful information on other aspects of CSCM components such as, constraints, licensing, cataloging and indexation. A partial sample of the composition descriptor for the component is illustrated in Table 5.3.

Table 5.2 Component definition

```
    import java.io.File;

    Component Compressor {
       public String displayUsage(){…}
       // compositional members
       // zip algorithm
    Compositional public File zip(File f, String oper) {
    }
      // gzip algorithm
    compositional public File gzip(File f, String oper){
    }
}
```

5.3.3 CSCM component implementation

The CSCM component implementation part contains one core implementation class for the component core members and a different class for each compositional interface. Though logically related, a component's compositional interface implementations are physically

disjoint, thereby providing compositional capabilities, flexibility, and a method for easier software development, maintenance and reuse.

The core class is connected to the classes of compositional interfaces via a composition relationship as illustrated in Figure 5.3.

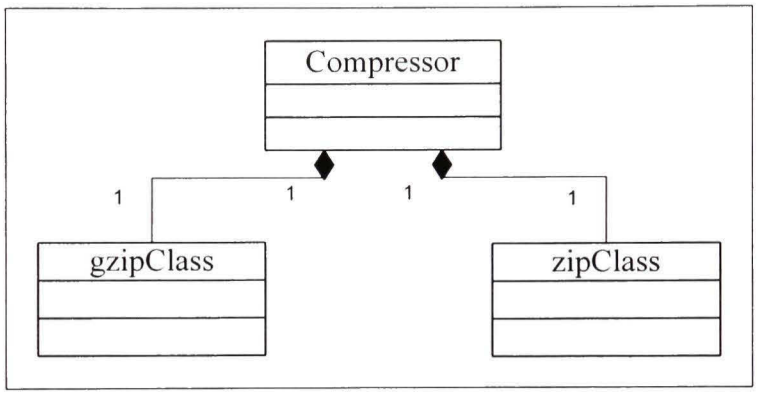

Figure 5.3 Relationship between the component core class and the component compositional interfaces.

Besides the implementation of core members, the component core class also provides a proxy delegate method for each compositional interface. The implementation of the CSCM composition mechanism is based on the composition and delegation principles. Furthermore, the implementation of compositional interfaces as separate classes helps also realizing the implementation of this mechanism.

Typically, object methods can access one another. This is also true for CSCM compositional interfaces; they can access each others through the delegation mechanism which uses the core instance to dispatch access requests to the concerned interface members by calling their proxy methods.

5.4 CSCM component-based software construction process

The CSCM component-based software construction process is divided into two processes: the component construction process and the application construction process.

Table 5.3 Partial illustration of the composition descriptor for the component defined in Table 5.2

```
<component name"Compressor">
  <import-list>
    <package>java.io.File<package>
  </import-list>
  <core-members>
    <methods>
      <method name="displayUsage" return-type="String"/>
        <modifiers-list scope="public"/>
        <parameters/>
        <dependency-list/>
          <methods>...</methods>
          <attributes>...<attributes>
        </dependency-list>
      </method>
    </methods>
  </core-members>
  <compositional-members>
    <attributes/>
    <interfaces>
      <interface name="zip" selected="true"
            return-type="File"/>
        <modifiers-list scope="public"/>
        <parameters>
          <parameter Type="File" name="f">
          <parameter Type="String" name="oper">
```

Table 5.3 A partial illustration of the composition descriptor for the component defined in Table 5.2 (continued)

```
            </parameters>
            <dependency-list/>
              <methods>
                <method name="displayUsage"
                     return-type="String"/>
                <modifiers-list scope="public"/>
                <parameters/>
                </dependency-list>
            </dependency-list>
          </interface>
          <interface name="gzip" selected="true"
                               return-type="File"/>
            <modifiers-list scope="public"/>
            <parameters>
              <parameter Type="File" name="f">
              <parameter Type="String" name="oper">
            <parameters/>
            <dependency-list/>
              <methods>
                <method name="displayUsage"
                            return-type="String"/>
                <modifiers-list scope="public"/>
                <parameters/>
                </dependency-list>
              <methods>
            </dependency-list>
          </interface>
        </interfaces>
    </compositional-members>
</component>
```

5.4.1 Component construction process phases

The construction process of a CSCM component passes through four phases as illustrated in Figure 5.4:

1. Component core definition and compositional interface definitions;
2. Compilation and generation of core skeleton and compositional interface skeletons;
3. Core implementation and compositional interface implementations;
4. Compilation of the skeleton implementations and generation of the composition descriptor.

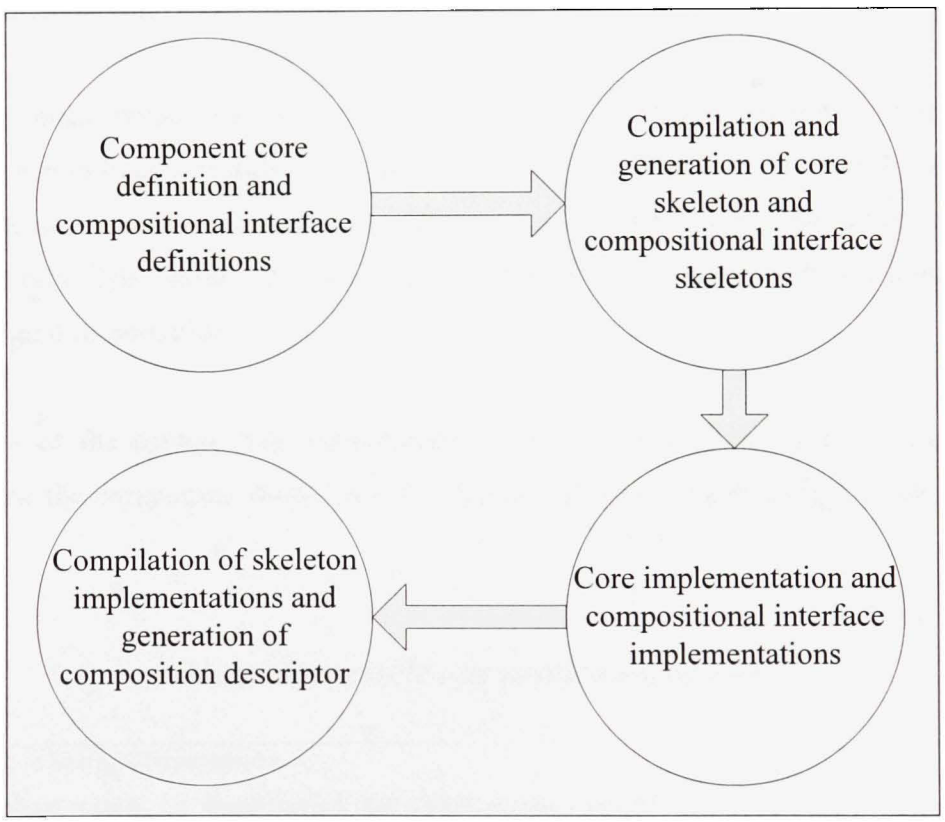

Figure 5.4 CSCM component construction process phases.

During the first phase, the component core members and compositional interface members must be defined by the software constructor in a similar way to that in which object-oriented

classes are defined. The output of this phase is the component definition source code as illustrated in Table 2.

During the second phase, the component definition source code is passed to the CSCM compiler for compilation. As illustrated in Figure 5.1, the outputs of this phase are the core skeleton as well as the skeletons for each compositional interface.

The implementation of the component skeletons has to be done by the software constructor in the third phase. The output of this phase (see Figure 5.1) is the component implementation source code which includes the implementation of the core skeleton as well as the implementation of each compositional interface skeleton.

During the fourth phase, the component compositional interface skeleton implementations and core skeleton implementation are passed again to the CSCM compiler (see Figure 5.1) to generate the composition descriptor as well as the source and binary code of the component implementation. The output of this phase is a CSCM component ready for composition, inheritance and instantiation.

Illustrations of the source code implementation for the core skeleton and compositional skeletons for the component shown in Table 5.2 are presented respectively in Table 5.4 and Table 5.5.

Table 5.4 Component core implementation code.

```
public class Compressor {
  /* this code is generated by CSCM compiler */
  java.util.Hashtable compositionalInterfaces;
  Compressor(new File("compositonDescriptor"){
      initializeCompositionals();
}
```

Table 5.4 Component core implementation code (continued)

```
    Public void initializeCompositionals(){
        compositionalInterfaces = new
              java.util.Hashtable();
   /* for every selected interface in the composition
      descriptor create an instance of the interface and
      store in the hash table compositionalInterfaces */
    }
  // core method
    public String displayUsage(){
        return new String();
    }
    // compositional interfaces
    Private File gzip(File f,String oper){
       return((gzipClass) compositionalInterfaces.get("gz"))
                   .gzip(f, oper);
    }
    private File zip(File f, String oper){
       return((zipClass) compositionalInterfaces.get("z")).zip(f, oper);
    }
}
```

Table 5.5 Compositional interfaces skeletons implementation code

```
  Public class zipClass {
    /* The code is generated by the CSCM
       Compiler */
  Compressor __This;
  zipClass(Compressor comp){
        __This = comp;
  }
  public File zip(File f, String oper) {
        __This.displayUsage();
```

Table 5.5 Compositional interfaces skeletons implementation code (continued)

```
            return new File("testTextFile.txt");
    }
}

Public class gzipClass{
      Compressor __This;
      gzipClass(Compressor comp){
        __This = comp;
      }
    public File gzip(File f, String oper){
          __This.displayUsage();
          return new File("testTextFile.txt");
       }
}
```

5.4.2 Software application construction process

The process of software application construction using CSCM components requires the completion of two tasks: first, the software constructor has to select the component's compositional interfaces needed (to satisfy the application requirements) via the components' composition descriptors. Second, the software constructor has to use and manipulate the components as if it were an ordinary object oriented class.

Whenever an instance of a CSCM component is to be created with a different combination of functionalities, this instance must be provided with the appropriate composition descriptor instance. For example, to select a particular interface implementing a required functionality, the developer has to set the value of the parameter "selected" of the compositional interface element to "true" in the composition descriptor of the component as shown in Table 5.3. The

name of the file containing the composition descriptor instance must be passed at instantiation as a parameter to the component's constructor.

5.5 CSCM Java inheritance issues

CSCM component are object oriented types equipped with a composition mechanism which after transformation by the CSCM compiler are mapped to ordinary Java components. In this section, we explain how the object-oriented inheritance mechanism is handled by a Java implementation of the CSCM model.

5.5.1 CSCM and Java class inheritance

CSCM components may inherit other CSCM components as well as ordinary Java classes using the same syntax rules of Java class inheritance (i.e. using the reserved word "extends"). Consequently, the same rules that apply to class inheritance apply also to CSCM components inheritance with minor differences. When a CSCM component inherits another base CSCM component, not only it inherits the composition descriptor of the base component, but also its compositional interfaces. However, when a CSCM component inherits a Java class, the members of the inherited base class will be available to other CSCM components as core members only.

5.5.2 CSCM and Java interface inheritance

CSCM components may inherit Java interfaces using the Java "implements" keyword. Similar to the way in which Java classes implement interfaces, CSCM components have to fulfill the contract dictated by the interface they implement. Moreover, the inherited members will be available as core members only.

5.6 Limitations

Even though the CSCM offers a solution for component unwanted members, it suffers from a number of limitations:

1. In the context of CSCM only functional members are selectively composed. This means that unwanted (state) data members can still be present in the CSCM components and ultimately lead to similar problems that the CSCM tackles.
2. The composition descriptor of CSCM components is expressed in XML. Though this format is well structured and user-friendly, it is significantly verbose and requires the software constructor to expend considerable effort for both the definition of components and for the selective composition of component members.
3. From an implementation viewpoint, the CSCM model has been designed to use an internal structure to hold a reference to all compositional member instances whether they have been selected or not. In other words, if the compositional member is not present in the component (set to null) a reference holding its name is still stored internally. Though it is minimal, the internal structure holding a reference to a compositional member incurs an additional overhead in terms of memory consumption to the component.

5.7 Conclusion

In this chapter, the CSCM was presented. This model permits the construction of software components with variable lists of functionalities selected according to components' composition descriptor instances at runtime. Consequently, it provides a remedy for the limitation of components' unsed members.

CHAPTER 6

ATOMIC AND MODULAR COMPOSITION MODEL

6.1 Introduction

In this chapter, the Atomic and Modular Composition (AOC) model is presented. The AOC model can be considered as an evolving and enhanced model compared to its predecessor (CSCM). First, the AOC model is designed to overcome the limitations of which the CSCM model suffers from. Second, it tackles additional software component limitations unaddressed by the CSCM model as will be discussed subsequently in this chapter. In particular, the AOC model aims at remedying the following limitations which have been discussed and illustrated in chapter 3:

1. Unwanted component's members;
2. Chaotic composition and amalgamation of code elements;
3. Suboptimal component reuse limitations.

Apart from remedying to the limitation listed above, the AOC model has an impact on a variety of software component construction aspects such as component modularity, reuse, testing, fast code comprehension and evolution.

In an attempt to reduce the complexity of code, increase its modularity and make it more manageable, programming paradigms came up with their own particular constructs. For instance, the functional programming paradigm uses the "function" (expressed in terms of a lambda expression) construct as the major container construct to decompose the code into modular and manageable chunks. Even though the function construct represents elegantly the computational behavior of a function from a mathematical viewpoint, the decomposition of a software application into functions still shows signs of code amalgamation as discussed in chapter 3. Hence, room for improvements still exists.

In the same vein, the procedural programming paradigm resorted to structuring the code into coarse-grained containers (modules) composed of finer-grained containers of data (data structures) and behaviors (procedures). Again, procedural languages modules and procedures are prone to chaotic code amalgamation similar to functional languages modules and functions. Eventually, new constructs are required to alleviate for chaotic amalgamation of code.

Equally, OOP have brought significant improvements with the notion of objects which mirror abstractions of real world objects and their interaction in real life systems. In OOP, the class is the container which encapsulates data and behaviors in one unit. The ability to emulate real life object decomposition and object interaction has considerable positive impact on the decomposition of code elements and their interactions. Nevertheless, code amalgamation can creep in object behaviors as has been demonstrated in section 3.3.2

Lately, the component paradigm has emerged aiming initially among other things to bring enhancements to the distribution of objects and their composition. However, the component paradigm *per se* has not come up with a completely new compositional physical structure such as class, procedures and functions, albeit it has brought in the virtual component structure. Essentially, the component structure is constructed out of a combination of compositional container structures employed by the existing programming paradigms.

At inception, the primary objective of the component paradigm was to permit transparent distribution of components and allow their interaction at runtime (dynamic composition) in a transparent and more flexible way. Basically, CORBA and COM/DCOM/.NET and EJB components have been augmented with metadata and helper constructs to achieve transparent distribution and interactions in distributed environments.

6.2 AOC model tenets

The AOC model is a component model relying on the straightforward composition of components and more importantly on the restructuring of the basic reusable element found in object oriented programming, specifically the class element. The tenets on which stands the AOC model are:

1. Promotion of class behaviors to full-fledged compositional components. First, as a reminder, the class construct in OOP encapsulates stateful (types) and behavioral (functions or methods) elements in one container. While type elements are full-fledged first class citizens, class behavioral elements are subordinates to the class and play a secondary role within the class construct as opposed to *type* members as can be depicted in Figure 6.1. This figure illustrates the code of a traditional Java component having as *type* `Person`. This component is composed out of two other components named (firstName and lastName) and both having the same *type* `String`. Additionally, this component has a single behavior called `display`. It is intresting to note that the `display` behavior can not be used on its own unless the `Person` component is used. Put differently, *type* elements are easily composed and reused with other *type* elements. On the contrary, behavioral elements are reused indirectly and only through the use of their container class. The AOC model uses the mechanism of functor objects (Coplien, 1992; Odersky, 2009) to make behavioral class members first class citizen whereby they have their own structural containers through which they can be composed with other classes as shown in Figure 6.2. This figure illustrates the code of the same component shown in Figure 6.1 right after being transformed into an AOC component. It is important to note that the `display` behavior has now its own structure. Precisely this component is now a full-fledged component having as name `Displayer`. This new structure allows the displayer behavior to be reused or not without being referenced through the `Person` component. More importantly is the way the `Person` component now composes the `Display` behavioral component as shown in Figure 6.2, i.e. the `Display` component is defined like any other component. It is worth observing how much the code of the

`Person` component is neater, cleaner and concise and does not have any behavioral implementations, but only component definitions.

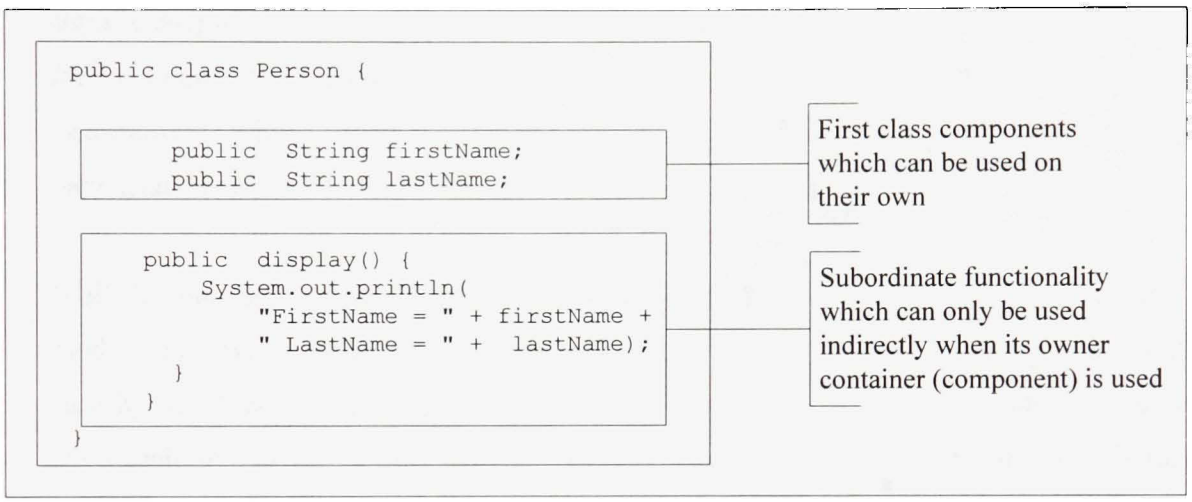

Figure 6.1 Structural relationship between a traditional component and its code elements.

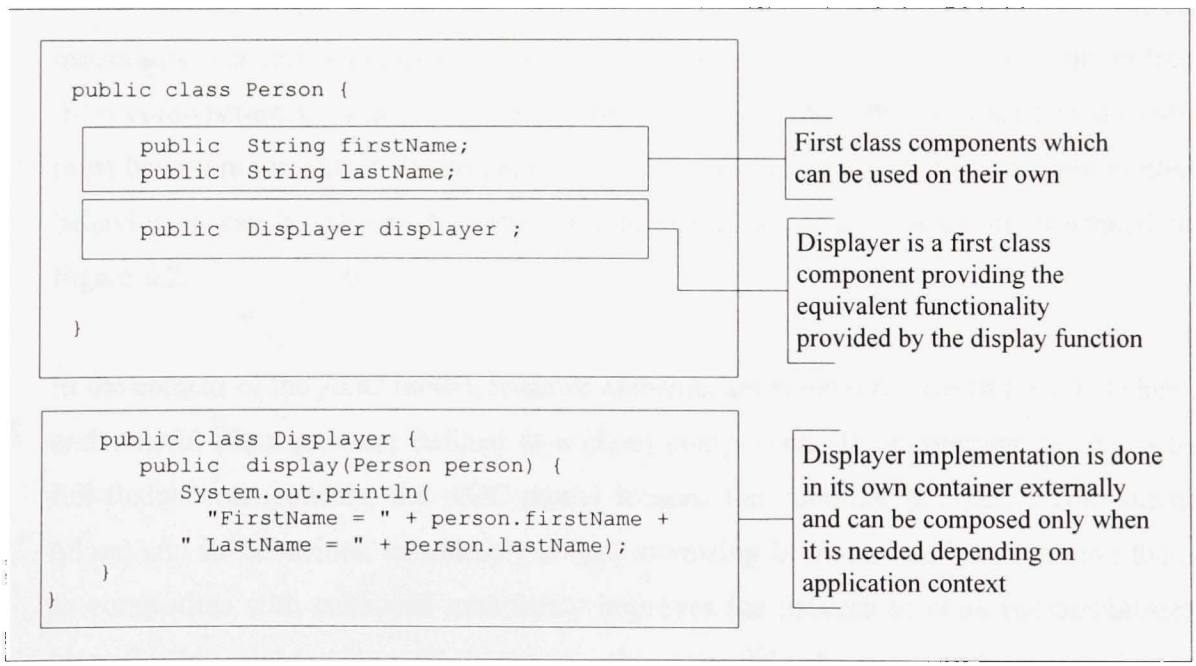

Figure 6.2 Structural relationship between an AOC component and its composite components.

2. Use of atomic or composite behavioral components with enhanced modularity. The most significant contributions of object oriented programming is the use of the object abstraction to decompose software into objects which emulate real world objects. The significance of this abstraction stems from: a) its ability to provide a natural mechanism for software decomposition into objects, b) the straightforward understanding of decomposed objects' code, i.e the decomposed objects' responsibilities match closely their equivalent real world object responsibilities.

   Well decomposed objects which closely match real life ones, are easier to understand, modify and evolve since their responsibilities can usually be predicted relatively with ease by developers. Typically, when objects responsibilities and representations diverge from their real word counterparts, chaotic amalgamation and bloating of code chunks occur and result in ill-composed and unoptimally modular code chunks.

   To avoid chaotic amalgamation and code bloats, we propose the breakdown of large or coarse grained bloated components into atomic or composite components with enhanced modularity. An atomic component can be described to be an indivisible code element free from composition with any other component. A component with enhanced modularity must have a minimalistic size to provide a concise meaning and to capture a pinpointed behavior as can be observed in the code of the `Displayer` component illustrated in Figure 6.2.

   In the context of the AOC model, reusable elements are behavioral (operations in a class) and stateful (data structure defined as a class) components. By promoting behaviors to full-fledged components, the AOC model loosens the coupling between a component (class) and its behaviors. In addition to that, atomizing behaviors and constructing them as composites with enhnaced modularity improves the process of class responsibilities identification and assignment, increases the potential of reuse, makes it easier to understand atomic behaviors intended use and makes their testing and evolution simpler and easier.

3. Declarative selection and composition of composite components. Since in OOP the class is the basic reusable element, any class member can be used and manipulated even though it is not needed by the constructed computational tasks. For instance, different application domains and contexts may use a variable number from the set of available class members. This scenario typically leads to unwanted members as mentioned in section 3.3.1. What is missing is the fine grained control mechanism in which only the required class members will be reused by a software application according to its context and business domain.

   In this research, the use of declarative definition of composite components is proposed as a remedy to the limitation of a software component unwanted members. To construct a particular application, the software developer needs only to select and define the required composite component members leaving out any unwanted members. The use of declarative definition of composite components works in concert with the other two tenants of the AOC model. Since in the context of the AOC model all class behavioral members are full classes, then those classes can be composed with other components in a fluid and straightforward manner.

The main drivers behind the AOC model are to:

1. Leverage software composition to a new height, achieve better software reuse and consequently improve the software construction process. Such a new height promotes software construction from being merely an instructional composition process towards a higher level being mostly a software component compositional process.
2. Encourage software constructors to think in terms of compositional components and how to compose and wire them to construct a software application rather than in terms of code instruction amalgamation and writing. Put differently, using the AOC model software constructors have to shift their focus by spending more effort on composing reusable components rather than on amalgamating code instructions.

6.3 Structure and anatomy of AOC components

6.3.1 AOC structure

An AOC component is a software replaceable element acting as a compositional structure or as a container for other AOC components. Basically, an AOC component possesses an OO class structure. AOC components are divided into two groups: atomic and composite. Furthermore, they are classified into two classes: stateful and behavioral. AOC groups and classes will be discussed in sections 6.3.2 and 6.3.4 respectively.

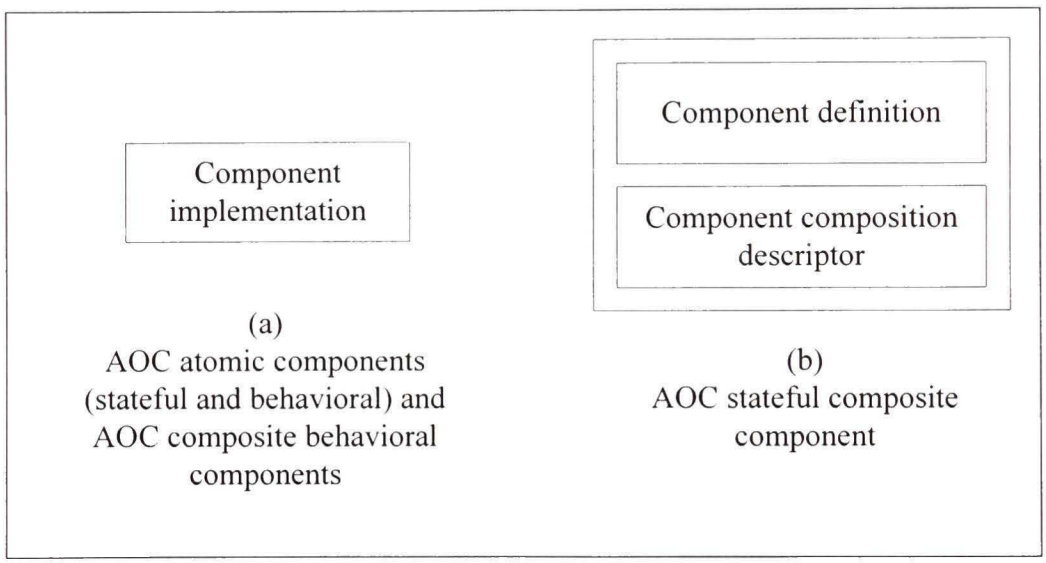

Figure 6.3 AOC component structure.

The structure of an AOC component depends on the group to which the component belongs as shown in Figure 6.3. For instance, Figure 6.3 (a) illustrates the structures of AOC atomic components (stateful and behavioral) and AOC composite behavioral components while Figure 6.3 (b) illustrates the structure of an AOC stateful composite component.

6.3.2 AOC component groups

Atomic: An AOC atomic component is a basic code element with enhanced modularity which can be used to construct other components. A distinctive feature of AOC atomic and enhanced modularity components is that they can not be refactored and divided into further AOC components and still be capable of providing the computational state or behavior they held or provided originally. In other words, if an AOC component can be broken down into distinct ones, this means it is not an AOC atomic or enhanced modularity component. Essentially, the atomicity or enhanced modularity of an AOC component is absolute across the space of software applications. Obviously, to preserve its atomicity and enhanced modularity an AOC component must be significantly concise, pinpointed and consequently of a small size as shown in the atomic AOC tax calculator component of Table 6.1.

Table 6.1 An AOC atomic or enhanced modularity component example

```
public class TaxAmountCalculator {

    public float calculate(float taxableAmount, float taxPercentage) {
        return taxableAmount * taxPercentage;
    }
}
```

Composite components: An AOC composite component is a code element with enhanced modularity which is composed out of atomic components as well as other composite components. For instance the code snippet of Table 6.2 belongs to an optimal composite component named "Person" and which is structurally composed out of two atomic components; the first holds the person first name and the second hold the person family name. A distinctive characteristic of AOC composite components is that they must not have any direct behavioral implementation within their structure in opposition to what is the case in traditional object oriented classes. Being typically application level components, AOC

composite component may have different composition configurations across different software application and business domains.

Table 6.2 AOC composite component example

```
public class Person {

    public String firstName;

    public String lastName;
}
```

6.3.3 AOC component classes

Stateful components: An AOC stateful component acts as a container which hosts data only. The data of an AOC stateful component is held inside one or more atomic or composite AOC stateful components. Furthermore, AOC stateful components are exclusively composed of only stateful components. In other words, AOC stateful components must not possess any direct behavioral implementations. This condition is required to achieve enhanced loose coupling between state and behavior structures. As a result, the achieved enhanced loose coupling is translated into enhanced software reuse. Such loose coupling permits the use of stateful structures without being closely tied to behaviors as is the case with ordinary OO classes.

AOC stateful components can belong to any AOC group (atomic or composite):

- An AOC stateful atomic component is a basic state holder belonging to primitive component types which are provided typically by the programming language. For instance, in Java, the classes "Boolean" and "String" can be considered as being atomic stateful components.
- An AOC stateful composite component is a component composed out of atomic or other AOC composite components as can be observed in the code snippet shown in Table 6.2.

AOC stateful composite components must be coupled with a declarative composition descriptor defining the inner composite components as shown in Table 6.4. The composition descriptor plays a key role in the AOC model since it provides loosely coupled composition relationships between the various components that might be useful to a particular application.

Behavioral components: An AOC behavioral component plays the role of a container or host for the implementation of only one and one unique business behavior. Put differently, an AOC behavioral component is focused, concise and is mandated to carry out a particular computational task. The main reason behind defining a singular behavior inside the body of an AOC behavioral component is to attain enhanced modularity and ultimately enhanced software reuse (see section 3.3.3). Similar to any traditional behavior, an AOC behavior is implemented inside its own class via an operation having arguments and a return type as illustrated in the code snippet of Table 6.1.

Table 6.3 Traditional object oriented class

```
public class Person {

    public  String firstName;
    public  String lastName;

    public  display() {
        System.out.println("FirstName = " + firstName + "
         LastName = " +  lastName);
    }
}
```

A behavior encapsulated within an AOC behavioral component manipulates its parameters without holding any state except local transitional state used to carry out the computational task it is mandated to perform (see Table 6.1). In this respect, AOC behavioral components are kind of *functor* objects which have significant similarities to functions employed in the functional programming paradigm (Kuhne, 1999). In other words, behavioral components

possess a unique and exclusive container and are independent of the component for which they provide service in opposition to how behaviors are subordinated and attached to OO classes. In an OO class the relation between the class and its subordinate behaviors is much stronger since behaviors cannot be used on their own apart from their container class.

The distinction between AOC behavioral components and OO class behaviors can be observed in the code snippets of Table 6.3, Table 6.4, and Table 6.5. Table 6.3, illustrates the code of a traditional OO class named `Person` with a single behavior named `display`. Table 6.4, illustrates the code of a computationally equivalent AOC `Person` component. Nevertheless, the `display` behavior of the AOC version is defined in an AOC behavioral component apart as shown in Table 6.5.

Table 6.4 AOC composite component which can be used as a composition descriptor

```
public class Person {

    public  String firstName;
    public  String lastName;
    public  Displayer displayer;
}
```

AOC behavioral components can be atomic or composite:

1. An AOC atomic behavioral component is characterized by its indivisibility in contrast to traditional behaviors. In other words, an atomic behavior is a code element which cannot be refactored further into other AOC behavioral components while still being able to carry out successfully its original computational task. It should be common to come across atomic behavioral components composed out of a single programming language instruction as illustrated in Table 6.1. Since any software application is represented by a hierarchy of interacting components, atomic AOC components can be thought of as being the leaves in such a hierarchy.

Table 6.5 AOC behavioral component

```
public class Displayer {
    public  display(Person person) {
        System.out.println("FirstName = " + person.firstName +
            " LastName = " + person.lastName);
    }
}
```

2. A composite behavioral AOC component is a component which has dependencies on other AOC behavioral components such as a call to other AOC behavioral components. In an application represented by a hierarchical tree of components, atomic AOC components can be thought of as being any internal non-leave component.

AOC behavioral components inpendently of their group or class must have enhanced modularity. To attain enhanced modularity, they must have small sizes. A number of reasons stand behind the reduced size of AOC behavioral components. First, small size behavioral components provide quick comprehension and understanding of the implementation logic and consequently the role they play in a particular implementation. Second, it is easier to write, test and test-cover considerable parts of the implementation for small sized components (Hughes, 1990; Yu *et al.*, 1998). Third, small size components increase not only the reuse potential of the component but also its ability to be composed with other components. Fourth, being of a small size, AOC components will exhibit higher degrees of cohesion which can be equally translated into higher degrees of software reuse.

Table 6.6, summarizes the relationship between the groups and classes of AOC components. AOC component groups and classes are orthogonal, i.e each component belongs at the same time to one component group and to one component class. It is important to note that the groups and classes of AOC components are conceptual and have no impact or implication on their implementations or physical structures.

Table 6.6 Illustrates the relationship between the AOC component groups and classes

Groups/classes	Stateful	Behavioural
Atomic	Yes	Yes
Composite	Yes	Yes

6.4 Characteristics of AOC components

AOC components have several distinguishing characteristics as opposed to other components. In particular, AOC components are characterized as being: atomic, structured for composition, loosely coupled, highly cohesive, state and behavior insulator, back and forward morphologic, seamlessly reusable in finer grains and fluidly and straightforwardly compositional. Each of these characteristics is discussed next.

6.4.1 Structured for composition

Since AOC components share a common physical envelop, in particular the "class" structure. This structure acts like an exclusive standalone structure either for state or behavior and thus permits easy structural composition of AOC components with each others. Traditionally, OO programming behaviors (methods) live as subordinates within their host class and can only be reused indirectly via their host class. In the context of the AOC model behaviors have their own structure which prevents them from being burried within the structure of their host class.

The composition structure of AOC component is obtained because:

1. Atomic AOC components by definition must be well structured for composition.
2. Composite AOC behavioral components by definition must be will structured for composition. As for AOC stateful components it is the responsibility of the software

constructor to select the required composite components which suit his application requirements via the use of AOC stateful component composition descriptors. Ultimately, structuredness for compositon will be obtained since any unwanted sub-composite components will be discarded from the AOC stateful component structure.

AOC components do not require customization adaptation or wrapping to be reused. To reuse an AOC component, the only task required is to declare it in the composition descriptor similar to way the `Displayer` component is used as shown in Table 6.4.

6.4.2 Loosely coupled

Achieving loose coupling is an important software engineering principle which software engineers try to attain. Existing guidelines to achieve loose coupling are not sufficient to attain this objective because of their inability to enforce a practice to reach it. Loose coupling of AOC component is achieved because:

1. Atomic AOC component are self contained with a small size so that they entertain minimal outside dependencies.
2. Composite AOC Stateful components are composed of only necessary composite components via the composition descriptor.
3. Composite AOC components must not only have small size to reduce the coupling with other components, but the size must be minimal or otherwise the component must be broken down into more than one AOC components.

6.4.3 Highly cohesive

Since coupling and cohesion go together, the same argument used in the previous point applies here. Furthermore, it can be argued that since AOC components possess a minimum of state/behavior as per application requirements, they are highly cohesive.

6.4.4 State and behavior insulator

Since promoting behaviors into structured first class components allows behavioral components to be loosely coupled and since compliant AOC components are separated each in its own structure independently of the class to which they belong, it can be infered that AOC model works as an insulator between state and behavior.

6.4.5 Back and forward morphologic

Inheritance in the OO paradigm is helpful to make reuse by block (i.e. reuse of groups of behaviors and structures). One problem with inheritance is its inability to allow variable subsets or combinations of inheritable elements. Such a feature can only be achieved through composition.

AOC atomic components aim at achieving better reuse by disassociating physically the component from its behaviors and data holding structures. However, the structures holding component behaviors and state are made available to the component in a loosely coupled way. Consequently, component behavior and state structures can be reused easily by other components. In this sense, AOC components possess the ability to evolve their morphology backward and forward. While current component models can inherit elements from their super classes, they do not have the ability to loose such inherited elements. On the contrary, AOC components, via their composition descriptor, can keep or loose composite components according to the software constructor's requirements.

6.4.6 Seamlessly reusable in fine grains

Using the AOC model, reuse is no longer on the coarse grained component level only: the reuse potential reaches optimal levels through the use of fine-grained optimally structured behaviors promoted to first class AOC components. This is an essential differentiator of the AOC component model from other component models.

6.4.7 Fluidly and straightforwardly compositional

Unlike, traditional components which in many cases need to be altered or tweaked so that they fit particular application requirements, AOC components can be easily and in a fluid and straightforward manner be used by software applications. The only step required to use an AOC component is to identify its availability and its suitability to satisfy application requirements. In other words, no customization and few modifications have to be done to use AOC components.

6.5 Advantages of AOC component based software application construction

In a world of continuous requirements evolution and change, the AOC component-based software construction approach attempts to provide remedies for a number of issues which have been facing software construction using traditional application construction approaches. Typically, during application construction, several components can be reused while many other ones might be unavailable and therefore need to be implemented from scratch. Nevertheless, using the AOC component-based software construction approach has several advantages.

6.5.1 Reduce the effort of software construction

Typically, an application is constructed out of newly created components which will be integrated with other components from a variety of libraries. The reuse of AOC component libraries reduces the effort of software construction since the readability and comprehension of AOC components is expected to be easier and faster. The argument to support this claim goes as follows. Since, AOC behavioral component are atomic, possess enhanced modularity and have loosely coupling, this means that their implementation code is short and therefore reading and understanding their implementation code is easier and faster than reading longer amalgamated and less modular code of their counterpart components. Even though

constructing new AOC components results in a large number of components compared to a traditional approach, the increase in the number of existing components is beneficial since it will increase the potential of software reuse on the one hand and it will lead to the creation of specialized AOC components having pinpointed behaviors on the other hand.

6.5.2 Reduced refactoring and adaptation of software components and classes

Since maintenance and evolution of code has been reported as taking 70% of total cost incurred by the overall lifecycle of a software (Trifu and Reupke, 2007), the AOC model will contribute significantly to reduce the cost of this task. Since AOC components are optimally modular, optimally coupled, the evolution of software is mainly adding new features in terms of AOC components. The integration of the new features takes minimal effort since no refactoring or modification to existing components will be required.

6.5.3 Less complex behavioral components

AOC behavioral components are small and provide implementation code for pinpointed behaviors. Consequently, they tend to be simple and less complex.

6.5.4 Contribute to uncover the space of reusable AOC components

Among the objectives of the AOC model is to increase software reuse. While traditional software components exhibit evident signs of software bloating and amalgamation, AOC components have almost none. It is interesting to examine the consequences of converting traditional software components to AOC components. Consequently, each traditional component procedure, function or method needs to be converted into one or more AOC component behaviors. As a secondary effect of such a conversion is the construction of considerably large number of AOC components. Nevertheless, this secondary effect is beneficial, since it contributes to uncover the space (libraries) of reusable AOC components.

6.5.5 Easier testability

AOC behavior al components contain the algorithmic implementation which manipulates state held by AOC stateful components. Since AOC behavioral components have small sizes, they are easier to test. Furthermore, constructed code and implemented testing scenarios and code tend consequently to be shorter and easier to understand, modify and maintain.

6.5.6 Flattened behavioral AOC component space

Traditionally, software components behaviors cannot be used directly without dereferencing their container components. In other words traditional component behaviors are organized in a hierarchical manner within their components. On the contrary, with the AOC model behaviors are first class components and can be accessed directly.

6.5.7 Convergence toward standard components

In the long term adoption of the AOC model can initiate a trend towards the standardization of the reusable components. Essentially, this expectation relies on the advantages and the benefits which can be obtained from using AOC components.

6.6 Limitations of the AOC model

Even though the AOC model provides enhancement to the way software construction has been done, it has a number of limitations:

6.6.1 Large number of constructed AOC components

AOC components have been designed to be of small sizes so that they will be optimally modular and optimally coupled components. In the event of converting a traditional

component to AOC components, then the number of resulting AOC components outnumbers the count of methods found in the traditional component. For instance, a traditional component having 10 behavioral members would result in at least 11 AOC components (one to hold the state and 10 to hold the 10 methods). However, providing the converted methods with enhanced modularity, will increase considerably the number of AOC behavioral components corresponding to those methods.

From a constructor viewpoint, the effect of a large volume of AOC component means accessibility to a large pool of reusable components suitable for a variety of applications. Constructors do not need to refactor or adjust those components to suit their requirements as might be the case using traditional software components. However, constructors would spend considerable time searching, identifying and composing the right components. To help software constructors identify and search for the required components, indexed and searchable libraries of AOC components can be provided.

From a performance viewpoint, the large number of components is a more serious limitation since application performance can be an issue. Fortunately, this can be remedied for by using the technique of code " in lining" which eliminates the need for intensive method calls exhibited by AOC component based applications.

It is essential to recall that the objectives of the AOC component model are to enhance software composition, increase software reuse, and reduce software construction effort. Those objectives require the involvement of software constructor human aspect. Consequently, any approach which aims to achieve those objectives has to take the developer human aspect into consideration. It is obvious that the AOC model is no exception in this respect and takes measures (use of atomic and highly modular components) to achieve the objectives set in this research project. It is worth mentioning that some researchers have stayed away from taking such measures and argue in favor of striking a balance between the size of the atomic behaviors and their effective application reuse. In other words, they prefer medium-sized behaviors. Although, the intention of this argument is understandable, it comes

at a price. For instance, medium to larger grained behavior are less modular and might suffer from code bloating and amalgamation. Furthermore, it contradicts best practice recommendations which favor software modularity, loose coupling and increased software reuse. From the AOC model viewpoint, medium to larger grained behavioral components defeats the essence of the optimal modularity principle.

The debate of using atomic versus medium-to larger grained behaviors currently seems to favor the use of the later approach over the first. It might be argued that the balanced behavior size is beneficial from various viewpoints (i.e. performance, manageable number of components). However, this argument is outdated and might be reasonable in the short term, but definitely looses ground on the long term. As simple as it might be, the answer to managing large pool of components lays in the use of software tools to search, identify and navigate through repositories of components. As for application performance, in-lining can be applied automatically in a systematic way to atomic behaviors. Simply stated, the benefits of using atomic behaviors as suggested in the AOC model outweigh the benefits of using a traditional approach.

6.7 Conclusion

In this chapter the AOC model and AOC components were presented. An overview of the different AOC components groups (atomic or optimal and composite components) and classes (behavioral and stateful) was also presented. Additionally, a discussion of AOC component characteristics is also presented. Finally, the advantages and limitations of the AOC model were also discussed.

CHAPTER 7

AOC MODEL BASED SOFTWARE CONSTRUCTION

7.1 Introduction

Traditional software construction involves the step of searching through APIs to identify potential reusable code elements in the form of components, classes, operations, etc. If the identification fails, developers resort to writing their own packages, modules, components, classes and operations, etc. The process of writing code involves typically writing programming language constructs and instructions, composing, connecting and wiring those instructions and constructs with pre-existing code. Finally, the software constructor writes test cases to validate the implementation against application business requirements.

Even though the traditional software construction process involves a form of software composition, nevertheless this form of composition is not leveraged efficiently to allow fluid and straightforward software composition at the higher level of components rather than at the level of instructions. The aim of the AOC model is to push the construction process towards higher levels of composition. In particular, the AOC model aims to promote software construction from a process characterized by chaotic amalgamation of code instructions towards a process relying increasingly on the composition of components characterized by their atomicity and enhanced modularity.

In this chapter, a process of software construction using the AOC model is presented.

7.2 Software construction using the AOC model

The AOC model based software construction process spans two phases:

1. the component construction phase and;
2. the application construction phase.

The whole construction process is illustrated through the construction of a sample Java application which makes use of few components to create a person object and display the person information on the screen. The components used by this application are: `String`, `Person`, `PersonDisplayer`, `PersonFactory` and `PersonMold1`. The grouping and classification of these components are shown in Table 7.1.

Table 7.1 Sample application component grouping and classification

Groups/classes	**Stateful**	**Behavioral**
Atomic or optimally modular	`String`	`PersonDisplayer`
Composite	`Person`, `PersonFactory`	`PersonMold1`

Typically, an application requirement can potentially be decomposed into one or more AOC atomic or composite components. Naturally, decomposed components can belong to different domains. It is the responsibility of the software constructor to create components so that they match closely their real word business component counterparts in terms of structure and role. The analysis and design phases of AOC are not tackled in this chapter since the scope of this thesis targets mostly the construction phase of the software life cycle.

7.2.1 AOC component construction phase

The construction phase of AOC components comprises three stages as depicted in Figure 7.1. Essentially, the construction of AOC components is similar to how classes are constructed in object oriented programming languages. However, there exist a number of subtle differences between how AOC components are structured and composed and how traditional OO classes are constructed.

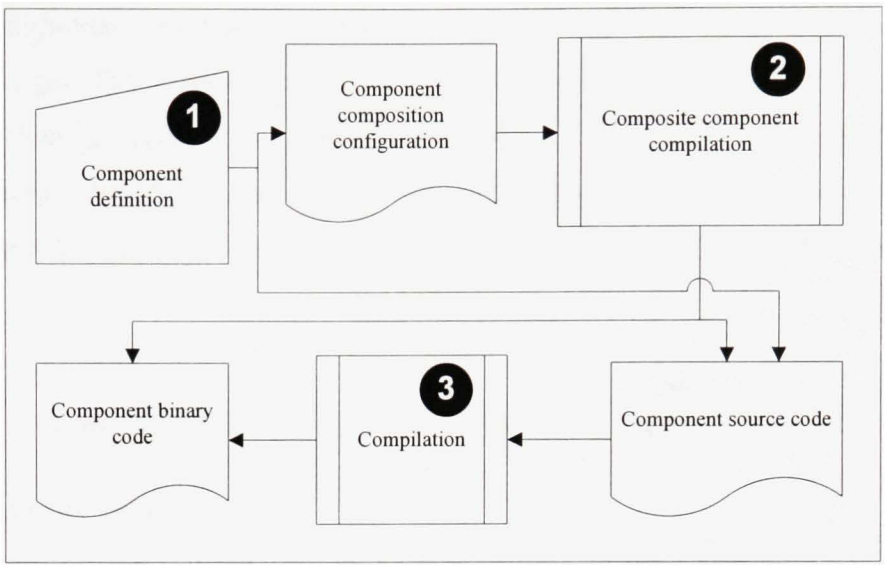

Figure 7.1 AOC component construction process.

Since AOC components are organized into orthogonal groups (atomic and composite) and classes (stateful, behavioral), it makes sense to discuss the construction of AOC components according to the different groups and classes to which they belong.

Stage 1: Component definition

Atomic stateful AOC components: Those components are the very basic and elementary code elements. The set of atomic AOC stateful components is small. They are usually provided by the implementation programming language. Examples of Java atomic AOC components are the `Long` and `Integer` components. Software constructors are rarelly required to define atomic stateful AOC components, but they reuse those components as sub-composites of other AOC components. In the sample application illustrated in this chapter the composition descriptor of the `Person` components makes use of the Java String component to define two instances of this component (`firstname` and `lastname`) as shown in Table 7.2.

Composite stateful AOC components: A composite statefull AOC component is defined inside the body of a composition descriptor. An AOC component composition descriptor

contains the definition of all sub-composite components which hold the state and behaviors of the component. The definition of a composite stateful AOC component is similar to the way an OO class is constructed except that no behavioral implementation must be present within its body as shown in Table 7.2. For the sake of simplicity, the composition descriptor is defined in a text file bearing the name of the component and having "comp" as file extension.

Table 7.2 AOC composite component composition descriptor

```
// Stateful components

public String firstName;
public String lastName;


// Behavioural components

public PersonDisplayer displayer;
```

Atomic behavioral AOC components: An atomic behavioral AOC component is defined like an ordinary Java class which embodies a single method and provides the implementation of the component behavior as shown in Table 7.3. It is necessary for an atomic behavioral component to be of a small size so that it preserves its indivisibility and optimal modularity.

Table 7.3 Example of an atomic behavioral AOC component

```
public void displayName(Person person) {
System.out.println("First name : " + person.firstName + ", Last name
            : " + person.lastName);
}
```

Composite behavioral AOC components: A composite AOC behavioral component is defined in a similar way to how an atomic AOC behavioral component is defined as shown in Table 7.4. The difference between atomic and composite AOC behavioral components is that composite behavioral AOC components can be decomposed into more than one behavioral AOC component. Furthermore, composite behavioral AOC components do not possess composition descriptors similar to those possessed by composite stateful AOC components.

Table 7.4 Example of a composite behavioral AOC component

```
public class PersonMold1 {
    public static Person createPerson(String firstName, String lastName) {
        Person person = new Person();
        person.firstName = firstName;
        person.lastName = lastName;
        person.displayer = new PersonDisplayer();
        return person;
    }
}
```

Stage 2: Composite component compilation

Atomic stateful AOC components: This stage does not impact atomic stateful AOC components, since they do not possess composition descriptors.

Composite stateful AOC components: A composite stateful AOC component composition descriptor which has been defined in the previous stage is passed to the AOC compiler to check its validity and to compile it into either Java source or binary code according to user instrumentation. Typically, composite stateful AOC component source code is generated in this stage as shown in Table 7.5.

Atomic behavioral AOC components: This stage does not affect atomic behavioral AOC components, since they do not possess composition descriptors.

Table 7.5 Example of a composite AOC component and its usage of atomic AOC subcomposite components

```
public class Person {
    public String firstName;
    public String lastName;
    public PersonDisplayer displayer;
}
```

Composite behavioral AOC components: This stage does not affect composite behavioral AOC components, since they do not possess composition descriptors.

Stage 3: Compilation

Atomic stateful AOC components: Atomic stateful AOC components do not exist on their own; they exist only as sub-composites of other AOC components. Therefore, compilation of atomic stateful AOC components is not performed.

Composite stateful AOC components: Composite stateful AOC components have to be compiled into binary Java code only when they have been compiled into Java source code in the stage 2.

Atomic behavioral AOC components: Atomic behavioral AOC components defined in terms of Java source code in the first stage have to be compiled into Java binary code.

Composite behavioral AOC components: Composite behavioral AOC components defined in terms of Java source code in the first stage have to be compiled into Java binary code.

In summary, the AOC component construction phase consists of defining AOC components and compiling them into Java binary code so that they will be ready for reuse during application construction.

7.2.2 AOC component-based application construction phase

Essentially, AOC component-based software construction process is oriented toward the composition of AOC components whith enhanced modularity. One of the interesting aspects of AOC component-based application construction is the composition of a minimal set of required components via components' composition descriptors.

Since AOC components are optimally modular and of small size, this means an abundance of reusable AOC components is expected to exist. Equally, since behavioral AOC components have their own structures and since composite stateful AOC components do not embody any behavioral implementations, assignment of behaviors to composite stateful AOC has to be done via composition. In particular, only the necessary components to satisfy application requirement have to be included in the composition descriptor. This means that the software constructor has to edit and modify the body of the composition descriptor of each component required by the application. It this process of composing components via the composition descriptor which promotes software construction from a rather programming language instruction composition towards a process based more on components composition.

Constructing application using the AOC model resembles in many areas traditional software application construction: however, it has some differences as will be discussed next. AOC component-based application software construction involves the following steps:

1. Identification of components. Similar to traditional software construction, the software constructor has to identify from libraries of AOC components the AOC components which satisfy application requirements. The process of identifying the right AOC components is probably easier since AOC components have small sizes and therefore are easier to read and comprehend.
2. Implementation of unavailable AOC components in libraries. Similar to traditional software construction, a number of components might not be available for a variety of reasons. Therefore, the software constructor has to construct those required AOC components according to the process depicted in Figure 7.1.

3. Composition of various components to construct the application according to requirements. This step is further divided into two substeps:
   a. Composition of composite stateful AOC components. For each identified or constructed composite stateful AOC component the software constructor has to select the set of sub-composite AOC components. This task is done by editing the composition descriptor and adding or removing any component declaration according to application requirements.
   b. Composition of composite behavioral AOC components. This involves the construction of application level AOC behavioral components to compose the various AOC components together according to application requirements.
4. Application compilation. This requires the compilation of all AOC components that are involved in the construction of the application. This step is divided into two sequential sub-steps:
   a. Generation of AOC composite stateful component source code. In this sub-step, the composition descriptor of each AOC composite stateful component will be parsed and validated to ultimately generate the AOC component source code. The process of parsing, validating and generating the Java source code of the composite AOC components is done by calling the AOC compiler. The AOC compiler works as preprocessor, i.e. as a source-to-source transformer. The textual content of the composition descriptor is passed to the AOC compiler to generate the Java source code of the AOC composite component.
   b. Application source code compilation. Since the source code of all the composite AOC components has been generated, the source code of the complete application is ready now for compilation as Java source. Obviously, the source compilation outputs the binary source code of the complete application.

An important difference between AOC application construction and traditional software application construction is the use of dedicated application binary code repositories by the AOC application construction process. Essentially, composite stateful AOC components are made available in one of two forms: as packaged libraries or as free flat textual source files

created by the software constructor. Since composite stateful AOC components are composed at application construction, the final structures of those components are determined exclusively by the requirements of each application. In other words, the same stateful composite AOC component might possess variable subsets of sub-composites in different applications. Consequently, the alteration of composite AOC components composition descriptors packaged into libraries requires access to those composition descriptors. Therefore, the required library packages of AOC components have to be unpackaged into a space where alteration of AOC composite stateful components' composition descriptors as well as generation of their components binary code has to be performed. In the context of the AOC model, this space is referred to as the application binary code repository.

7.3 Conclusion

In this chapter, software construction using the AOC model was presented. First, the phases in which the construction of an AOC component passes were discussed. Next, application software construction using AOC components was discussed also.

CHAPTER 8

AOC MODEL VERSIONING SCHEME

8.1 Introduction

Historically, software component versioning has received little attention. However, the quest for software reuse leads to the evolution of software component versions. Equally, the ubiquity of distributed systems necessitates the exchange of components binary code. Consequently, component version mismatch issues have caused faulty and malfunctioning behavior in their respective applications. In response to such issues, component versioning started to play an increasing role in the detection of software component mismatches. It is worth mentioning that a component version refers to the unique identifier associated with static binary code of the component. It identifies the version of the component used by a software application as opposed to the runtime dynamic identity of an instance of that component.

Current versioning schemes provide limited support to prevent such component versioning mismatch. For instance in Java, a class can be associated with a unique version identifier; however, the detection of a class version mismatch results in throwing a runtime exception which most often results in application halt. Furthermore, evolvable versions (i.e. new component versions) do not necessarily comply with older versions. For this reason, unique version identifiers are not sufficient to handle component evolution and versioning.

An interesting versioning scheme is used in the Open Services Gateway initiative (OSGi) to prevent mismatches between bundles, i.e. jar files (OSGi, 2009). Unfortunately, this scheme does not operate on the component level, but rather on bundles which are sets of components bundled together.

In this research project , a straightforward component versioning scheme is proposed to remedy to this limitation. Interestingly, this scheme fits quite well the AOC component model since all software application code elements are components, a significant difference to the way OO classes are structured with behaviors.

8.2 Component interfaces and interface definition language

Component interfaces can be considered as communication channels between components whether these components are deployed in a local or external computation environments. Component interfaces are often specified using an IDL (Interface Definition Language) as in CORBA and DCOM. However, interfaces can be specified differently via an interface structure as is the case in the Java programming language (see Chapter 1 for more details). Physically, a component interface can be considered as a type containing a set of method signatures whose implementations are provided by particular components. Current IDL languages do not allow the developer to specify interface extra-information which can be used during a component operational life.

A component interface specifies what operations are available through this interface while its implementation provides how those operations are performed. Consequently, component interfaces provide flexibility and loose coupling since they are not concerned with how the implementation will be done.

To overcome inter-components interoperability and development complexity problems, many definition languages have been designed to provide components with interfaces. Component model IDLs provide abstraction layers to reduce development effort as well provide reusable component intercommunication mechanisms within heterogeneous computing environments.

Currently available component models provide little or no semantic properties (C.Schmidt, Levine and Mungee, 1998; Watkins and Thompson, 1998). Component interfaces express

only functional aspects of a component without any consideration for aspects such as interface versioning. Nevertheless, attempts have been made to provide software component interfaces with semantics for different purposes. For instance, approaches to provide component interfaces with semantics and to express aspects such as quality of service and interface versioning have been described in the literature (C.Schmidt, Levine and Mungee, 1998; Exton, 1997; Jacobsen and Kramer, 2000; Watkins and Thompson, 1998).

8.3 AOC component interfaces

Composite AOC components can be considered as interfaces since any defined behavioral component implementation they use is provided in an outsider behavioral component (i.e. in a separate class). Consequently, the AOC versioning scheme proposed in this research is applicable to AOC components whether they are considered and used as interfaces or as components.

8.4 AOC component versioning scheme

One potential solution to remedy for component version mismatches is to provide versioning information with components and make use of this information ahead of runtime so that appropriate handling is performed. To this end, versioning information is provided in terms of annotations which can be queried even at runtime.

The AOC component versioning scheme requires the association of metadata versioning information with each AOC component. The responsibility of specifying and using the information versioning scheme falls on the software developer. The developer of the component must provide each component with its appropriate versioning metadata and must assume the responsibilities of managing this data during the evolving life of the component. The software developer who uses the component must specify the target component version his application requires.

Eventually, to detect any component versioning mismatches, a component version mismatch detector can be run against the application code to detect the presence of any mismatches. Typically, this version mismatch detection can be run only when new versions of the application are deployed. In an alternative scenario, software applications can be instrumented to check at start-up whether newer versions of its components are deployed and calls the version mismatch detector accordingly to check against version mismatches. This later alternative is efficient in the case of runtime application modification where components are created and used dynamically at runtime.

8.5 AOC component versioning scheme implementation

Technical implementation of this versioning scheme requires language constructs to express versioning metadata. As Java is used in this research for prototyping and experimentation, it makes sense to use Java annotations constructs to define AOC component version metadata. For appropriate handling of component version mismatches, versioning metadata has to be specified in two phases: component construction-time and application construction-time as will be discussed next.

8.5.1 Component construction-time

Each newly constructed component must be associated with its versioning metadata in terms of an annotation. In other words, component versioning information must be present ahead of time before use. The implementation code for the annotation holding version metadata for a newly created component is named `ComponentVersion` and is illustrated in Table 8.1. At constuction time the component version annotation must provide the following information:

1. Major version provides the major version of the component;
2. Minor version provides the minor version of the component;
3. Micro version provides the micro version of the component;
4. List of major versions with which the current component major version complies with;
5. List of minor versions with which the current component minor version complies with;

6. List of micro versions with which the current component micro version complies with.

The versioning metadata associated with a newly created component follows industrial conventions by identifying a component version as a sequence of three numbers (major, minor, and micro).

Table 8.1 Version annotation for newly created components

```
package org.component.version.annotation;


import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.ElementType;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface ComponentVersion {
int major ();
int minor();
int micro();
int [] complyWithMajors();
int [] complyWithMinors();
int [] complyWithMicros();
}
```

8.5.2 Application construction-time

At application construction time, the developer makes use of a set of components to construct his application. Obviously, container components (components declaring other components within their bodies) must specify the particular version of any component declared within their body. The implementation code for the annotation holding version metadata for a specific component declared within another container component is named `TargetComponentVersion` and is illustrated in Table 8.2.

Table 8.2 Version annotation for used components

```
package org.component.version.annotation;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;


@Retention(RetentionPolicy.RUNTIME)
@Target(    )
public @interface TargetComponentVersion {

int major ();
int minor();
int micro();

}
```

8.6 AOC component version mismatches detector tool

The prototype of the component version mismatch detector tool is obviously implemented in Java. The prototype implementation is simple and straightforward. As input it takes the directory of jar files used by a particular application and filters the class file stored inside the jar files. For each component found in the jar files, it validates whether the target specific version matches the specific version of the component loaded class. If the validation is negative, the version mismatch detector tool displays enough information so that the developer proceeds easily to remedy the problem. The displayed information lists the name of the jar file storing the container component, the full name of the container component and the full name of the declared component, i.e. the component whose version does not match the required version by the container component as shown in Table 8.3.

Table 8.3 Component version mismatch detector tool error message

```
Verifying version compliance of: C:\Users\Rowan\MyStuff\workspace\AC
component version mismatch detector\JarToBeValidated\compression_app.jar
```

Table 8.3 Component version mismatch detector tool error message (continued)

```
Component version mismatch...
Component name: world.component.compression.CompressingComponentDeclared
component name: world.component.compressors.TarCompressor
Required version 1.0.0
Found version 1.0.1
Compliance with majors[1], minors[0], micros[1]

Verifying version compliance of: C:\Users\Rowan\MyStuff\workspace\AC
component version mismatch
detector\JarToBeValidated\compressors.v.1.0.1.jar
```

The implementation of the component version mismatch detector is developed in Java and makes uses of two open source libraries: javassist and Apache common-io libraries. The javassist library provides an API for Java byte code introspection and manipulation while the Apache common-io library provides an API for Files management and filtering in (Apache, 2008). The complete listing of the mismatch detector tool is provided in Appendix II on a CDROM.

8.7 Usage scenario: Example

For better understanding of the AOC versioning scheme, a usage scenario is provided through a sample application. This application provides hypothetical compression service via two different types of compression algorithms. The application root component is named `CompressingComponent` and instantiates two compression components and readies them for user service. The implementation code for the root component is illustrated in Table 8.4. As shown in this table, the root component declares the `TarCompressor` and `ZipCompressor` components. Equally, each of those components is associated with a `TargetComponentAnnotation`. This annotation defines explicitly the required specific version of the component as required by its container component. In particular, for the `TarCompressor` component, the `TargetComponentAnnotation` specifies that the required major version must be equal to 1 and both the minor and micro versions must be both equal

to 0. A similar annotation pattern to what is shown for the `TarCompressor` component can be observed in the `ZipCompressor` component.

Table 8.4 Sample application

```
@ComponentVersion(major = 1, minor = 0, micro = 0,
    complyWithMajorVersions = {}, complyWithMinorVersions = {},
    complyWithMicroVersions = {})
public class CompressingComponent {

    @TargetComponentVersion(major = 1, minor = 0, micro = 0)
    public TarCompressor tarCompressor = new TarCompressor();

    @TargetComponentVersion(major = 1, minor = 0, micro = 1)
    public ZipCompressor zipCompressor = new ZipCompressor();

    public void listCompressionComponents() {
        zipCompressor.zip();
        tarCompressor.tar();
    }

    public static void main(String[] args) {
        new CompressingComponent().listCompressionComponents();
    }
}
```

Unlike the annotation used to specify the required version of a component inside its container component, the version of a component is specified using a different annotation, the `ComponentVersion` annotation. What is particular about this annotation is that it possesses lists of major, minor and micro versions which the component is capable of supporting apart from its own version. For example, the `TarCompressor` component `major`, `minor` and `micro` versions are respectively equal to 1, 0 and 0. Additionally, this component `compliesWithMajors`, `compliesWithMinors`, `compliesWithMicros` annotations are respectively empty meaning the component is compatible with no other version as shown in Table 8.5.

Table 8.5 Illustration of newly created component version annotation for the component TarCompressor

```
@ComponentVersion(major = 1, minor = 0, micro = 0,
   complyWithMajors = {} , complyWithMinors = {},
   complyWithMicros = {})
public class TarCompressor {

    public void tar() {
        System.out.println("Tar v.1.0.0 is in service...");
    }
}
```

Table 8.6 Illustration of newly created component version annotation for the component ZipCompressor

```
@ComponentVersion(major = 1, minor = 0, micro = 1
  complyWithMajors = (Interface21), complyWithMinors = {0},
complyWithMicros = {0})
public class ZipCompressor {

    public void zip() {
        System.out.println("Zip v.1.0.0 is in service...");
    }
}
```

While, the `ZipCompressor` component `major`, `minor` and `micro` versions are respectively equal to 1, 0 and 0, its versions compatibility lists (`compliesWithMajors`, `compliesWithMinors`, `compliesWithMicros`) have values respectively 1, 0 and 0 as shown in Table 8.6. This means that apart from its own version 1.0.1, this component is compatible with its pervious, i.e. version 1.0.0.

8.8 Advantages and limitations of the AOC versioning scheme

The version scheme of the AOC component model provides several advantages in comparision to current existing approaches.

1. Coexistence of multiple versions. The AOC versioning scheme allows two different versions of the same component to be used by a single application. Unfortunately, such a scenario is not realizable given the current Java specification and platform. To allow multiple version instances, the Java platform has to be modified since currently Java classes are distinguished mainly by their full name.
2. Software quality improvement. Improves the overall quality of a software component by reducing application faults and bugs related to version mismatched and due to version obsoleteness.
3. Improvement compared to existing approaches. For instance, Java classes have been augmented with a version attribute. Nevertheless, the AOC versioning scheme improves on that by providing compliance versioning metadata which enable components with compatible versions to replace each others. OSGi provides version information for bundles of components whereas the AOC versioning scheme targets fine-grained components, for instance ordinary Java classes.

On limitation of the AOC versioning scheme is that it requires more effort to be expended by the software constructor to provide the required versioning information as well as to specify the target version of the component as dictated by requirements. However, the extra effort is minimal and has to be incurred to obtain the benefits. Similirarily, existing approaches requires additional efforts to make use and take advantage of versioning information.

8.9 Conclusion

In this chapter, background information on component interfaces has been given. Additionaly, the AOC versioning scheme was presented as well as its implementation. The proposed versioning scheme makes use of the Java annotation feature to provide component versions as in terms of annotations. Additionally, a component version mismatch detector tool has also been presented along with a sample application illustrating how to use this versioning scheme. The advantage of using this versioning scheme and component version mismatch tool is an improvement in the overall quality of software applications.

CHAPTER 9

AOC COMPONENT COMPILER REALIZATION

9.1 Introduction

This chapter elaborates on the reference implementation of the AOC component compiler. The AOC compiler is specifically constructed to compile the composition descriptors of composite stateful AOC components. The compiler output is expressed in terms of Java source code. Precisely, the AOC component compiler acts as a source to source transformer since the composition descriptor content is also expressed in the same language of the generated code.

The next section presents an overview of the compilation and code generation processes. The following section presents the tools used in the implementation of this AOC compiler. The subsequent section presents the AOC compiler and discusses how it can be used. The last section identifies the limitation of the AOC compiler realization.

9.2 Overview of compilation and code generation

Among the expected deliverables of this research project is the implementation of an AOC compiler which is used to compile component's composition descriptors. The implemented compiler serves as a prototype to put into practice the software composition theoretical ideas tackled in this research work and to use those principles to leverage the software construction process. The AOC compiler takes as input the composition descriptors of an application's components and generates either source or binary code for those components in Java. To better understand the tasks done by the compilation and generation of composite components, a description of the compilation process with its underlying phases is presented next.

9.2.1 The process of compilation and code generation

The compilation process is considered as a process of program translation from one representation expressed in a specific language to a target language. The target language could be a programming language or machine code. Regardless of the passes and optimization done by a compiler, the basic tasks are common to all compilers. The process of compilation is broken down into two major parts: analysis and synthesis (A.W., 1997; Aho, Sethi and Ullman, 1986). The analysis part is further subdivided into three other parts (see next) which all contribute to the construction of an intermediate representation of the source program. The synthesis part uses the intermediate representation constructed previously during the analysis phase to construct or generate code in another intermediary representation or in a final target language.

Analysis phase. The analysis part consists of three sub-phases: lexical, syntactic and semantic. During the processing of those sub-phases, the operations of the program are determined and stored in a hierarchical structure called an AST (Abstract Syntax Tree). In addition, the token variables matched are stored in a data structure commonly known as the symbol table. This table contains important and necessary information (e.g. name, type of the symbol), which will be used by the various sub-phases of the compiler. Often in practice those sub-phases can be implemented separately or merged into one or more passes. Each of these passes is considered separately for the purpose of clarification.

1. **Lexical Analysis:** This sub-phase is responsible for scanning the source program's stream of characters, discarding all white spaces and comments and producing a stream of lexical tokens (e.g. identifiers, keywords, punctuation marks). The specification of lexical tokens is usually done using regular expressions. For example, an identifier might be specified as a collection of characters matched by the following regular expression:

```
[a-zA-Z] [a-zA-Z0-9]
```

This regular expression means that an identifier token begins with an alphabetical character and might be followed by zero or more alphanumerical characters. In addition, the lexical analyzer should also provide an error reporting mechanism which detects errors such as unrecognized tokens or misspelled keywords. There exist several lexical analyzer generator tools. The efficiency of those tools relies heavily on the power of regular expressions in the token specifications which therefore reduces the programming effort and the development time required to construct a compiler.

2. **Syntactic Analysis:** This sub-phase deals with the parsing of the program's tokens into grammatical phrases by applying the grammar rules of the source language. Context-free grammars, which are formal descriptions of recursive rules, are commonly used to guide the syntactic phase. Such a grammar is composed of terminal, nonterminal, start symbols and productions which are useful to describe the syntax of a programming language. The terminals represent the basic symbols of which a program is composed.

Table 9.1 Production rules

```
S → id = expression
expression → idenditfier
expression → number
experession → expression + expression
expersssion → expression * expression
```

For example, the tokens `number` and `identifier` present in the set of productions shown in Figure 9.1 are terminals. The nonterminals are syntactic variables which represent string sets and should appear on the left-hand side of one or more productions. For example, `expression` in Table 9.1 is a nonterminal which denotes the set of strings that form the language defined by the grammar. The production rules of a grammar determine the way in which terminal and nonterminal symbols are combined to form the program phrases.

The parsed phrases will be represented by what is known as a parse tree. For instance, the parse tree for the expression "`salary = basic + rate * 10`" is shown in Figure 9.1. In this tree, interior nodes are nonterminal symbols and leaf nodes are terminal symbols. However, a parse tree contains certain useless tokens such as punctuation, additional nonterminal symbols and extra productions rules added technically to help in the transformation of the grammar (factoring and elimination of ambiguities) (A.W., 1997). These extra details are useful to the parsing phase and are not of any help in the later phases. Consequently, a more condensed AST tree free from useless details is constructed from the parse tree in order to be used by the later compilation phases. The AST representation constructed from the parse tree for the above mentioned expression is shown in Figure 9.2. In an AST, operators and keywords are not leaves but interior nodes.

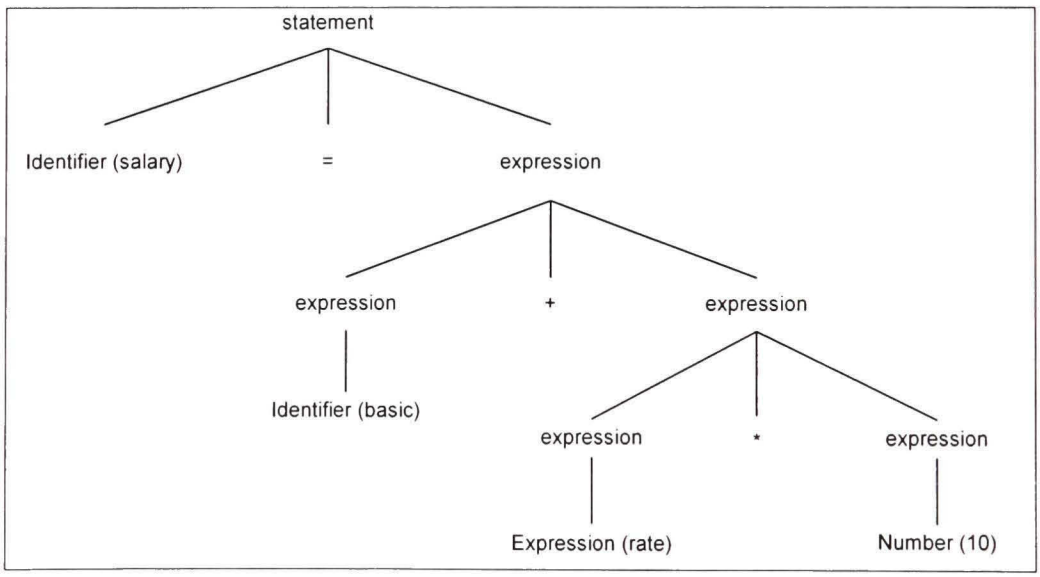

Figure 9.1 Parse tree representation.

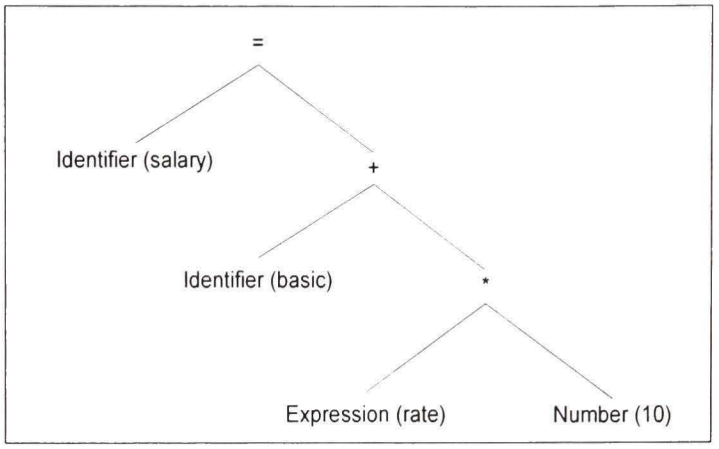

Figure 9.2 Abstract Tree Representation.

3. **Semantic Analysis:** In this sub-phase, the compiler traverses the syntax tree to collect type information and checks the source code against semantic errors such as illegal and undeclared types of expressions as well as illegal scope access to variables. It relies heavily on the information provided by the symbol table to accomplish its mission. As mentioned before, and in many cases, the lexical, syntactic and semantic analyses are done in one pass. The output of the semantic analysis can be considered as an annotated AST tree in addition to the content of the symbol table.

Synthesis Phase. The synthesis phase deals with different issues such as translation of the IR (intermediate representation) into another IR, control and data flow analysis depending on specific needs and purposes (e.g., various optimizations), and code generation. At this point of discussion, the translation or code generation phase is of importance to us. This phase can be done in two ways (Parr, 1996):

1. Syntax-directed translation. This method generates the target code progressively as the parsing proceeds. The input stream is parsed and translation actions are executed at the same time parsing is being done, so that at the end of parsing the output code has been generated. This method does not make use nor requires an IR of the source program. For this reason, it was used in old compilers when memory was not large enough to hold the entire abstract syntax tree of the program.

2. Translation through the use of an AST intermediate representation. In this method, the AST is traversed and translation actions are performed to generate the target code or to construct another IR which will be later wrapped into the target language. Generally, when a new IR is produced as an abstract syntax tree not only it conserves the semantic of the program and simplifies its representation, but also serves as a clean interface between the parser and the later phases of the compilation process. This method has some advantages over the other. For instance, the IR is an independent representation that could be mapped into any desired transitional intermediate or target languages without implementing a new parser or even altering the existing one, thus facilitating the portability of the compiler. Furthermore, it can be manipulated and optimized to suit specific needs. Usually and from a practical perspective IRs must have certain properties which make the manipulation of those properties simple and efficient. Some of these properties are:
   a. IR must be easily produced by the semantic analysis phase;
   b. IR must be convenient to translate into any language;
   c. IR should have individual components which describe extremely simple tasks such as a single fetch, store, add and move or jump operations.

In this synthesis phase, the source program AST representation is used to generate the program into the target language. Actually, the process of code generation takes place as the AST is traversed. Most of the time, every AST node contains enough information to generate the code. However, when additional information is needed the process becomes more difficult since additional secondary traversals have to be done in order to gather, from the other nodes, the necessary information to accomplish the translation of the specific construct. Consequently, tree traversals become more complicated. Fortunately, automatically generated tree traversal procedures can help do the task efficiently and easily. The traversal and translation of the tree is performed through the use of grammar rules and abstract syntax constructs associated with semantic actions. The AST nodes are tagged with tokens representing the terminal and non-terminal symbols. Those tags will be used by the code generation process.

9.3 Compiler construction tools

The evolution of compilation techniques and parsing methodologies has changed the way compilers are constructed. Compilers are often constructed manually. However, compiler parsers and AST traversal procedures for prototyping and some large grammars are constructed automatically using compiler constructor tools such as ANTLR (Another Tool for Language Recognition) (Parr, 2007), SableCC (Gagnon, 1998) and JavaCC (Sun, 2009).

To construct a compiler, compiler construction tools takes as input a file containing regular expression specifications of lexical tokens in addition to the rules of a specific programming language to generate as output the compiler API. Such API includes both lexical and syntax analyzers in addition to a parser for a specific programming language. Typically, the lexical analyzer is called by syntax analyzer to return the next token in the stream of characters representing the source program. In the same vein, the lexical analyzer is called by the parser to recognize the program instruction set in terms of valid programming language sentences. When a parser is called it usually constructs and returns an AST to the caller application.

To transform or to translate the source code into a different format or a different programming language, the information stored in an AST tree is used to generate the target source code. To traverse the AST tree, the tree walker (tree traversal API) is used. AST tree walkers can be constructed automatically using compiler construction tools.

To construct an AST tree walker, compiler construction tools take as input a grammar file containing a set of rules. Those rules describe the AST tree structure and are expressed in terms of regular expressions annotated with actions to direct the code generation process. This brief overview on compiler construction tool helps understand how the AOC compiler construction process works. The AOC compiler construction process is described in a subsequent section.

9.3.1 ANTLR a compiler construction tool

ANTLR is a compiler constructor tool widely used in both education and industrial sectors (Parr, 2007). ANTLR is an open source tool developed and maintained by Terence Parr to generate top-down "ll*" (leftmost derivation with unlimited look-ahead) parsers. ANTLR is distinguished by its ability to parse a grammar into AST tree so that later on it can be manipulated for optimization, transformation or translation purpose.

9.4 Purpose of the AOC compiler

As mentioned previously, Java has been chosen to be used as the programming language of experimentation in this research project. The AOC compiler takes as an input a composition descriptor which defines the sub-composites for a particular AOC stateful composite component. As output it produces either the source or the binary class file of that component according to the instrumentation supplied by the constructor. Since the content of a composition descriptor is expressed in terms of a subset of Java constructs, the AOC component compiler has therefore to parse those Java constructs and transform them accordingly into Java source code.

It might be confusing why a composition descriptor expressed in terms of Java constructs has to be translated into another text file containing the same Java source constructs. There are two main reasons behind this:

1. To constrain user from supplying arbitrary Java implementation code instructions inside the composition descriptor of an AOC stateful composite component.
2. To make sure that the user supplies the right syntactic constructs when defining the content of composition descriptors. In other words, it is to ensure that the definitions of a component's sub-composites constructs are valid and consequently this component can be composed at application construction with the desired set of sub-composites.

9.5 AOC compiler input and output

The AOC compiler receives a composition descriptor of an AOC composite component as input to produce out the source or the binary code of that component according to user instrumentation. However, from the time the input is read to the time the output is generated, there are several intermediate representations through which the input will pass.

9.5.1 Input representation

The input to the AOC compiler is a set of AOC component composition descriptors where each is associated with one and only one AOC composite stateful component. The composition descriptor consists of a textual file which is distinguished by its file name and extension. The composition descriptor must have the same name as the component it is associated with but a different file extension, i.e "comp". The role of this distinctive extension name is to make the composition descriptor recognizable by both the developer and more importantly, by the AOC compiler.

A composition descriptor holds a set of constructs which describe the structural component composition definitions required by the associated AOC composite component as shown in Table 9.2. The set of constructs used to describe the definition of composites are borrowed from the Java programming language. The rationale behind using Java constructs is purely driven by simplicity and ease of use. Since the target programming language of the composed component is also Java, it makes sense to express the composition relationship of a component in the same language of its implementation.

9.5.2 Output representation and generation

After the AOC compiler parses the composition descriptor of an AOC stateful composite component, it constructs an AST representation of the sub-composite component definition found in the composition descriptor. Next, the AOC compiler traverses the constructed AST

tree and at each node executes an associated action to finally generate the source code for the AOC composite component associated with the composition descriptor. Eventually, the source code is compiled using a java compiler to produce out the Java binary class file of the component.

Table 9.2 AOC composite component which can be used as a composition descriptor

```
public class Person {

    public  String firstName;
    public  String lastName;

    public  Displayer displayer;
}
```

Even though the definitions of component sub-composites are expressed in Java, they could have been equally expressed in another language such as XML or the like.

9.6 AOC compiler construction

ANTLR has been used as the automatic compiler construction tool to construct the AOC component compiler. The choice to use ANTLR is driven by the know-how to use it as well as by its wide acceptance as a compiler construction tool. The process of AOC compiler construction is illustrated in Figure 9.3.

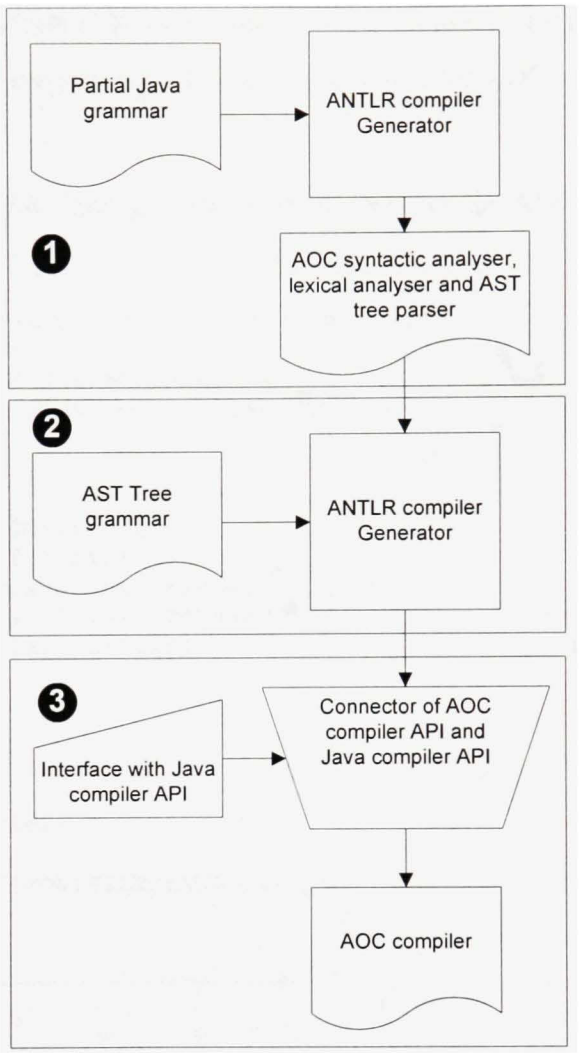

Figure 9.3 AOC compiler construction stages.

The construction of the AOC compiler is done into three stages as shown in Figure 9.3 (numbered in black circles):

1. Construction of AOC compiler lexical and syntactic analyzers as well as its parser. In this stage, ANTLR takes as input a grammar file of which a snippet is shown in Table 9.3 and the complete listing is shown in Appendix III. This grammar is an adaptation of the Java grammar which is available on ANTLR Web site (Habelitz, 2008). The grammar file provides ANTLR with the input to produce several modules of the AOC compiler. The

output produced by ANTLR is a Java API containing both the lexical and syntactic analyzers as well as the parser (AST tree constructor) for AOC composition descriptors.

Table 9.3 ANTLR input grammar rules to generate the AOC compiler parser

```
// Starting point for parsing a Java file.
componentSource
    :   classScopeDeclarations*
        -> ^(JAVA_SOURCE classScopeDeclarations*)
    ;


classScopeDeclarations
    :    modifierList
        (  type classFieldDeclaratorList SEMI
            -> ^(VAR_DECLARATION modifierList type
classFieldDeclaratorList)
        )
    |   SEMI!
    ;

modifierList
    :   modifier*

        -> ^(MODIFIER_LIST modifier*)
    ;
```

2. Construction of the AST tree walker and code generator. To construct the AST tree walker and the code generator, ANTLR takes as input another grammar file containing rules to traverse the AST tree. Furthermore, those rules are annotated with actions to direct the generation of the component source code. A snippet of the grammar rules used by ANTLR to generate the AST walker and code generator API is shown in Table 9.4. The complete listing of this grammar is shown in Appendix IV.
3. The last part of the AOC compiler is developed manually. This part is responsible to hook the different API parts of the AOC compiler together and to interface with the Java compiler API to allow direct compilation of AOC components into Java binary code.

Table 9.4 ANTLR grammar rules to generate the AOC walker and code generator

```
// Starting point for parsing a Java file.
componentSource [String componentName, String extendedComponentName]
    :
^(JAVA_SOURCE  (c+=classScopeDeclarations)* ) ->
component(componentName={$componentName},
extendedComponentName={$extendedComponentName}, composites={$c})

    ;

classScopeDeclarations
    :  ^(VAR_DECLARATION
    mL=modifierList
    t=type
        fl=classFieldDeclaratorList) ->
composite(modifiers={$mL.st}, type={$t.st}, fieldNameList={$fl.st})
    ;

modifierList
    :

     ^(MODIFIER_LIST  (mods+=modifier)* ) ->
modifier(modifier={$mods})
    ;

modifier
    :    PUBLIC     -> modifier(modifier={$PUBLIC.text})
    |    PROTECTED  -> modifier(modifier={$PROTECTED.text})
    |    PRIVATE    -> modifier(modifier={$PRIVATE.text})
    |    STATIC     -> modifier(modifier={$STATIC.text})
    |    ABSTRACT   -> modifier(modifier={$ABSTRACT.text})
    |    NATIVE     -> modifier(modifier={$NATIVE.text})
    |    SYNCHRONIZED  -> modifier(modifier={$SYNCHRONIZED.text})
    |    TRANSIENT  -> modifier(modifier={$TRANSIENT.text})
    |    VOLATILE   -> modifier(modifier={$VOLATILE.text})
    |    STRICTFP   -> modifier(modifier={$STRICTFP.text})
    |    FINAL  -> modifier(modifier={$FINAL.text})
    ;
```

The code of the AOC compiler is provided in Appendix V on a CDROM.

9.7 Conclusion

In this chapter, the AOC compiler construction process has been illustrated. At first, an overview of compilation and code generation has been given. Later, background information was given on compiler construction tools, in particular the ANTLR compiler generator. Additionally, the purpose of the AOC compiler was explained as well as a description of the input and output to the AOC compiler. Finally, the process of constructing the AOC compiler was presented.

CHAPTER 10

AOC MODEL APPLICATION STUDY CASE

10.1 Introduction

In this chapter, a case study is conducted to demonstrate the applicability of the AOC model as well as to demonstrate how the AOC model differs from the traditional OO software construction approach. The objectives of this case study are to demonstrate:

1. How software complexity is reduced;
2. How software reuse is increased;
3. How software construction, evolution and maintenance becomes easier;
4. How the AOC model promotes software construction to a higher level of software composition;
5. How AOC application can be constructed from existing applications.

10.2 Description of the application

The application used in this case study concerns a phone directory stored in a textual file. The main computational task the application performs is reading the phone directory entries from the file and displaying them on the console. The application is intentionally constructed with a minimal set of functionalities for the sake of clarity and simplicity.

10.3 Case study conduction Plan

The steps followed in the conduction of this cases study are:

1. A first prototype code of the phone directory application is presented. The code of this prototype is constructed using a rapid, but simple and naïve object oriented approach.
2. A second prototype of phone directory application is illustrated. However, this prototype code has been subject to restructuring and organization by a first round of refactoring.

3. A third prototype of the phone directory application is again restructured according to another round of refactoring.
4. A fourth and final prototype of the phone directory application is illustrated where the code of this prototype has been refactored to reach a level where all the code elements of the application have attained enhanced modularity.
5. The application is constructed following the AOC model approach.
6. The different application construction steps are analyzed, discussed, and compared. The output of this step is a comparison summary of the different approaches. The comparison is conducted according to the objectives set forth at the beginning of this chapter.

10.4 Construction of the application following an OO approach

10.4.1 First prototype

This first version of the phone directory application is an example of typically monolithic applications. The code of this application is contained into two components (classes) as shown in Table 10.1 and Table 10.2

Table 10.1 Phone directory first prototype – part 1

```
package org.ac.sample.phoneDirectory;

public class Entry {

private String lastName;
private String firstName;
private String internationalCode;
private String areaCode;
private String phone;

    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
```

Table 10.1 Phone directory first prototype – part1 (continued)

```
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getInternationalCode() {
        return internationalCode;
    }
    public void setInternationalCode(String internationalCode) {
        this.internationalCode = internationalCode;
    }
    public String getAreaCode() {
        return areaCode;
    }
    public void setAreaCode(String areaCode) {
        this.areaCode = areaCode;
    }
    public String getPhone() {
        return phone;
    }
    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

Table 10.2 Phone directory first prototype - part 2

```
package org.ac.sample.phoneDirectory;


import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class PhoneDirectoryApp{
```

Table 10.2 Phone directory first prototype - part 2 (continued)

```
public List<Entry> readEntries(File file) throws IOException {

    FileInputStream fileInputStream =  new FileInputStream(file);
    InputStreamReader inputStreamReader = new
            InputStreamReader(fileInputStream);
    BufferedReader bufferedReader = new
            BufferedReader(inputStreamReader);

    String line = null;
    List<Entry> entries = new ArrayList<Entry>();

    while((line = bufferedReader.readLine()) != null) {
        String[] values = line.split(",");
        Entry entry = new Entry();
        entry.setFirstName(values[0]);
        entry.setLastName(values[1]);
        entry.setInternationalCode(values[2]);
        entry.setAreaCode(values[3]);
        entry.setPhone(values[4]);
        entries.add(entry);
    }

    return entries;

}

public static void displayDirectory(List<Entry> entries) {
    for (Entry entry : entries) {
        System.out.println(entry.getFirstName() + " "
            + entry.getLastName()
            + ", " + entry.getInternationalCode()
            + "(" + entry.getAreaCode() + ")" + entry.getPhone());
    }
}

public static void main(String[] args) throws IOException {
    PhoneDirectoryApp app = new PhoneDirectoryApp();
    String filePath = "C://ProgramFiles//EclipseWorkspace//"
              + "ABC_Component_Model//phoneDirectory.txt";
    File file = new File(filePath);
    List<Entry> entries = app.readEntries(file);
        displayDirectory(entries);
    }
}
```

10.4.2 Second prototype

In the second prototype, a first round of refactoring has been performed. The result of this refactoring round is internal code reorganization of one of the two components of the application. It is worth mentioning that the refactoring done in this prototype did not add any new component as can be observed by looking at the code of
Table **10.3**.

Table 10.3 Phone directory application second prototype part-2

```
package org.ac.sample.phoneDirectory;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class PhoneDirectoryApp{


    private BufferedReader getBufferedReader(File file) throws
                                                    IOException {

        FileInputStream fileInputStream =  new FileInputStream(file);
        InputStreamReader inputStreamReader = new
                    InputStreamReader(fileInputStream);
        BufferedReader bufferedReader = new
                    BufferedReader(inputStreamReader);
        return bufferedReader;

    }


    private Entry bindValuesToEntry(String[] values) {

        Entry entry = new Entry();
        entry.setFirstName(values[0]);
        entry.setLastName(values[1]);
        entry.setInternationalCode(values[2]);
        entry.setAreaCode(values[3]);
        entry.setPhone(values[4]);
        return entry;
    }
```

Table 10.3 Phone directory application second prototype part-2 (continued)

```
    public List<Entry> readEntries(File file) throws IOException {

        BufferedReader bufferedReader = getBufferedReader(file);
        String line = null;
        List<Entry> entries = new ArrayList<Entry>();

        while((line = bufferedReader.readLine()) != null) {
            String[] values = line.split(",");
            entries.add(bindValuesToEntry(values));
        }

    return entries;

    }


    public static void displayDirectory(List<Entry> entries) {
       for (Entry entry : entries) {
           System.out.println(entry.getFirstName() + " "
                + entry.getLastName() + ", "
                + entry.getInternationalCode()
                + "(" + entry.getAreaCode()
                + ")" + entry.getPhone() );
       }
    }


    public static void main(String[] args) throws IOException {
        PhoneDirectoryApp app = new PhoneDirectoryApp();
        String filePath = "C://ProgramFiles//EclipseWorkspace//"
                  + "ABC_Component_Model//phoneDirectory.txt";
        File file = new File(filePath);
        List<Entry> entries = app.readEntries(file);
        displayDirectory(entries);
    }
}
```

10.4.3 Third refactored prototype

In this third prototype a considerable refactoring effort has been performed. The result of this second round of refactoring is the introduction of many new components. Those components have been identified from the application domain. The code of this application is illustrated

in Table 10.4, Table 10.5, Table 10.6, Table 10.7 and Table 10.8. The illustration of the complete code is essential to make the comparison between the different prototypes simple and straightforward.

Table 10.4 Phone directory application third prototype part-1

```
package org.ac.sample;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.IOException;
import java.io.InputStreamReader;

public class FileUtils {


    public static BufferedReader getBufferedReader(File file) throws
                                                        IOException {

        FileInputStream fileInputStream =  new FileInputStream(file);
        InputStreamReader inputStreamReader = new
                    InputStreamReader(fileInputStream);
        BufferedReader bufferedReader = new
                    BufferedReader(inputStreamReader);
        return bufferedReader;

    }

}
```

Table 10.5 Phone directory application third prototype part-2

```
package org.ac.sample;

public class Person {

private String firstName;
private String lastName;

    public Person(String firstName, String lastName) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
```

Table 10.5 Phone directory application third prototype part-2 (continued)

```
    public String getFirstName() {
        return firstName;
    }
    public void setFirstName(String firstName) {
        this.firstName = firstName;
    }
    public String getLastName() {
        return lastName;
    }
    public void setLastName(String lastName) {
        this.lastName = lastName;
    }
    public String toString() {
        return this.firstName + "," + this.lastName;
    }
}
```

Table 10.6 Phone directory application third prototype part-3

```
package org.ac.sample;

public class Phone {

public String internationalCode;
public String areaCode;
public String localNumber;

    public Phone(String internationalCode, String areaCode,
                    String localNumber) {

        this.internationalCode = internationalCode;
        this.areaCode = areaCode;
        this.localNumber = localNumber;
    }

    public String getInternationalCode() {
        return internationalCode;
    }

    public void setInternationalCode(String internationalCode) {
        this.internationalCode = internationalCode;
    }
```

Table 10.6 Phone directory application third prototype part-3 (continued)

```
    public String getAreaCode() {
         return areaCode;
    }

    public void setAreaCode(String areaCode) {
        this.areaCode = areaCode;
    }

    public String getLocalNumber() {
        return localNumber;
    }

    public void setLocalNumber(String localNumber) {
        this.localNumber = localNumber;
    }

    public String toString() {
        return this.internationalCode
                    + "(" + this.areaCode + ")"
                    + this.localNumber;
    }
}
```

Table 10.7 Phone directory application third prototype part-4

```
package org.ac.sample.phoneDirectory;

import org.ac.sample.Person;
import org.ac.sample.Phone;

public class Entry {


    private Person person;
    private Phone phone;

    public Entry(String[] values) {
        this.person = new Person(values[0], values[1]);
        this.phone = new Phone(values[2], values[3], values[4]);
    }

    public Person getPerson() {
        return person;
    }
```

Table 10.7 Phone directory application third prototype part-4 (continued)

```
    public void setPerson(Person person) {
        this.person = person;
    }

    public Phone getPhone() {
        return phone;
    }

    public void setPhone(Phone phone) {
        this.phone = phone;
    }

    public String toString() {
        return this.person  + " " +  this.phone;
    }
}
```

Table 10.8 Phone directory application third prototype part-5

```
package org.ac.sample.phoneDirectory;

import java.io.BufferedReader;
import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;

import org.ac.sample.FileUtils;

public class PhoneDirectoryApp{

    public List<Entry> readEntries(File file) throws IOException {
        BufferedReader bufferedReader =
                        FileUtils.getBufferedReader(file);
        String line = null;
        List<Entry> entries = new ArrayList<Entry>();

        while((line = bufferedReader.readLine()) != null) {
            String[] values = line.split(",");
            entries.add( new Entry(values));
        }

        return entries;
      }
```

Table 10.8 Phone directory application third prototype part-5 (continued)

```
    public static void displayDirectory(List<Entry> entries) {
        for (Entry entry : entries) {
            System.out.println(entry);
        }
    }


    public static void main(String[] args) throws IOException {
        PhoneDirectoryApp app = new PhoneDirectoryApp();
        String filePath = "C://ProgramFiles//EclipseWorkspace//"
                    + "ABC_Component_Model//phoneDirectory.txt";
        File file = new File(filePath);
        List<Entry> entries = app.readEntries(file);
        displayDirectory(entries);
    }
}
```

10.4.4 Fourth prototype constructed according to the AOC model

The construction of the phone directory application using the AOC uses the third prototype as its basis since it shares several characteristics with the AOC component model prototype. In particular, they both have small size components exhibiting enhanced modularity. The difference between the third prototype and the AOC (fourth) prototype is in the decomposition of the third prototype into composites that can be selected and hooked into the final component at application construction time. Obviously, the number of components has been increased significantly in the fourth prototype explicitly so that the fourth prototype complies with the AOC model as can be observed in the code illustrated from Table 10.9 to Table 10.28.

Table 10.9 AOC component 1

```
package org.aoc.sample.phone;

public class Phone {
    public Integer internationalCode;
    public Integer areaCode;
    public Integer localNumber;
}
```

Table 10.10 AOC component 1 composition descriptor

```
//Stateful components

public Integer internationalCode;
public Integer areaCode;
public Integer localNumber;

//  behavioral component
org.aoc.sample.phone.Displayer displayer
```

Table 10.11 AOC component 2

```
package org.aoc.sample.phone;
    public class Displayer {

    public static void display(Phone phone) {
        System.out.print(phone.internationalCode + "\t\t\t");
        System.out.print(phone.areaCode + "\t\t");
        System.out.print(phone.localNumber + "\t" + "\n");}
    }
}
```

Table 10.12 AOC component 3

```
package org.aoc.sample.phone;

public class PhoneFactory {
    public static Phone create(Integer internationalCode,
                        Integer areaCode, Integer localNumber) {
```

Table 10.12 AOC component 3 (continued)

```
        Phone phone = new Phone();
        phone.internationalCode = internationalCode;
        phone.areaCode = areaCode;
        phone.localNumber = localNumber;
        return phone;
    }
}
```

Table 10.13 AOC component 4

```
package org.aoc.sample.phone.directory.app;

import java.io.File;
import java.io.IOException;
import java.util.List;

import org.aoc.sample.phone.directory.entry.Entry;
import org.aoc.sample.phone.directory.entry.FileParser;
import org.aoc.sample.phone.directory.entry.ListDisplayer;


public class ApplicationLuncher {

    public static void main(String args[]) throws IOException {

        File file = new
                File("C://Users//rowan//MyStuff//EclipseWorkspace//" +
                "5_AtomizedComposed//phoneDirectory.txt");
        String splitToken = ",";
        List<Entry> entries = FileParser.parseEntries(file,
                splitToken);
        ListDisplayer.display(entries);

    }
}
```

Table 10.14 AOC component 5

```
package org.aoc.sample.phone.directory.entry;

public class Displayer {
```

Table 10.14 AOC component 5 (continued)

```
        public static void display(Entry entry) {
        entry.person.displayer.display(entry.person);
        entry.phone.displayer.display(entry.phone);
    }
}
```

Table 10.15 AOC component 6

```
package org.aoc.sample.phone.directory.entry;

public class Entry {
    public org.aoc.sample.phone.Phone phone;
    public org.aoc.sample.phone.directory.person.Person person;
}
```

Table 10.16 AOC component 6 composition descriptor

```
//Stateful  components

public org.aoc.sample.phone.Phone phone;
public org.aoc.sample.phone.directory.person.Person person;
//    Behavioral component
      public org.aoc.sample.phone.directory.entry.Displayer displayer;
```

Table 10.17 AOC component 7

```
package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.directory.person.Person;

public class Factory {

    public static Entry create(Person person, Phone phone) {
```

Table 10.17 AOC component 7 (continued)

```
        Entry entry = new Entry();
        entry.person = person;
        entry.phone = phone;
        return entry;
    }
}
```

Table 10.18 AOC component 8

```
package org.aoc.sample.phone.directory.entry;

import java.io.File;
import java.io.IOException;
import java.util.ArrayList;
import java.util.List;
import org.io.buffered.file.Reader;
import org.io.buffered.file.ReaderFactory;

public class FileParser {

    public static List<Entry> parseEntries(File file, String splitToken
                                          ) throws IOException {

        Reader  reader = ReaderFactory.create(file);
        String line = null;
        List<Entry> entries = new ArrayList<Entry>();

        while((line = reader.lineReader.readLine(
                      reader.bufferedReader)) != null) {
            entries.add(StringValuesToEntryBinder.
                bindStringValuesToEntry(line.split(splitToken)));
        }

        return entries;
    }
}
```

Table 10.19 AOC component 9

```
package org.aoc.sample.phone.directory.entry;

import java.util.List;
```

Table 10.19 AOC component 9 (continued)

```
public class ListDisplayer {

    public static void displayList(List<Entry> entries) {
        System.out.println("NameFamilyNameInternationalCode"
          +     "AreaCodeLocalCode");
        for (Entry entry : entries) {
            Displayer.display(entry);
        }
    }
}
```

Table 10.20 AOC component 10

```
package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.PhoneFactory;
import org.aoc.sample.phone.directory.person.Person;
import org.aoc.sample.phone.directory.person.PersonFactory;

public class StringValuesToEntryBinder {

    public static Entry bindStringValuesToEntry(String[] values) {

        Person person = PersonFactory.create(values[0].trim(),
                            values[1].trim());
        Phone phone = PhoneFactory.create(
                Integer.parseInt(values[2].trim()),
                Integer.parseInt(values[3].trim()),
                Integer.parseInt(values[4].trim()));
        Entry entry = Factory.create(person, phone);
        return entry;
    }
}
```

Table 10.21 AOC component 11

```
package org.aoc.sample.phone.directory.person;
public class Person {
    public String firstName;
    public String lastName;
    public Displayer displayer;
}
```

Table 10.22 AOC component 11 composition descriptor

```
// Stateful components

public String firstName;
public String lastName;


// Behavioral components
      public org.aoc.sample.phone.directory.person.Displayer displayer;
```

Table 10.23 AOC component 12

```
package org.aoc.sample.phone.directory.person;

public class Displayer {

    public static void display(Person person) {
        System.out.print(person.firstName + "\t");
        System.out.print(person.lastName + "\t\t");
    }
}
```

Table 10.24 AOC component 13

```
package org.aoc.sample.phone.directory.person;

public class PersonFactory {

    public static Person create(String firstName, String lastName) {
        Person person = new Person();
        person.firstName = firstName;
        person.lastName = lastName;
        return person;
    }
}
```

Table 10.25 AOC component 14

```
package org.io.buffered.file;

import java.io.BufferedReader;

public class Reader {
// Stateful components, because we need to hold this object for repeated
reads.
    public BufferedReader bufferedReader;

//behaviour components
    public org.io.buffered.file.line.Reader lineReader;
}
```

Table 10.26 AOC component 14 composition descriptor

```
//Single stateful component
public java.io.BufferedReader bufferedReader;

//Multipe behavioural components
public org.io.buffered.file.line.Reader lineReader;
```

Table 10.27 AOC component 15

```
package org.io.buffered.file;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStreamReader;
import org.io.buffered.file.Reader;

public class ReaderFactory {    public static Reader create(File file)
throws FileNotFoundException {
        org.io.buffered.file.Reader reader = new
                    org.io.buffered.file.Reader();
        FileInputStream fileInputStream =  new FileInputStream(file);
        InputStreamReader inputStreamReader = new
                    InputStreamReader(fileInputStream);
        reader.bufferedReader = new BufferedReader(inputStreamReader);
        return reader;
    }
}
```

Table 10.28 AOC component 16

```
package org.io.buffered.file.line;

import java.io.BufferedReader;
import java.io.IOException;

public class Reader {
    public static String readLine(BufferedReader bufferedReader)
                throws IOException {
        return bufferedReader.readLine();
    }
}
```

10.5 Analysis, discussion and comparison between the different versions

Each of the objectives set forth at the beginning of this case study is analyzed and discussed according to the variations found in the different prototypes of the same phone directory application.

10.5.1 How software complexity is reduced

Small size components especially behavioral components which provide algorithmic computation are less complex than coarse or large-grained components. Even though the third prototype has undergone significant refactoring effort, the AOC prototype has far more components (triple the count of components of the third prototype). Obviously, the increase in the number of components is the result of decomposing the components of the third prototype into many smaller components and consequently less complex components. For example the first prototype method `readEntries` has been decomposed into three methods in the second prototype. In the third prototype, the number of methods into which the `readEntries` has been decomposed stayed the same, however those methods have undergone some refactoring to make it easier to understand what those methods are intended to provide in terms of computational tasks. It is worth mentioning that obtaining code which

has undergone many refactoring passes is not often possible due to various constraints which have already been discussed in chapter 3.

In the AOC prototype, this `readEntries` method has been refactored to become the `parseEntries` method which is available through the `FileParser` AOC behavioral component, from a size viewpoint it has an equivalent size to the version of the third prototype. However, the methods it uses in the third prototype have been promoted to full fledged components to increase their reuse potential and to make their composition selective at application construction time in compliance with the AOC model.

Even though the `parseEntries` method has preserved an equivalent size to its `readEntries` counterpart in the third and AOC prototypes, from a complexity point of view it has isolated and packaged glue code of the third prototype version into structured and full fledged component code in the AOC prototype (i.e. in the `parseEntries` method). In other words, this componentized glue code is represented in an identifiable behavior which will be easier to understand, search or and consequently reuse. This easiness of handling and manipulating identifiable and componentized glue code is considered as a form of complexity reduction.

Table 10.29 Application prototypes structured code elements comparison

Prototype/comparison criteria	**Package count**	**Component count**	**Operation count**	**Reusable component count**	**Reusable stateful composite component count**
First prototype	1	2	3	1	-
Second prototype	1	2	5	1	-
Third prototype	2	5	8	4	-
AOC prototype	6	16	11	15	4

10.5.2 How software reuse is increased

Basically, the count of structured code elements is used as an indicator of increased reuse. In this case study, it is convened to not consider `set` and `get` operations as structured code elements since they are provided as ad-hoc operations to access properties and therefore can be removed in many cases. The bigger the count of structured code elements, the bigger is the reuse potential for those elements in various applications. Comparing the number of code elements shown in Table 10.29 indicates clearly that the AOC model approach has the highest rank when it comes to software reuse potential among the various prototypes. Furthermore, the AOC prototype has promoted many behaviors (methods) in the previous prototypes to full fledged components so that their potential of reuse is increased. For instance, the `readEntries` method of the third prototype belongs the `PhoneDirectoryApp` component. This component is root of the application: therefore, the reuse of its methods is not straightforward unless the software constructor has already developed this application and has deep knowledge of its internals and location. The name of the component does not provide any indication as to the computational role of its internal method. On the contrary, using the AOC model the method `readEntries` has been promoted to a full-fledged component having as name `org.aoc.sample.phone.directory.entry.FileParser`. In addition, the body of the `readEntries` method uses five different reusable AOC components:

1. `org.io.buffered.file.Reader`
2. `org.io.buffered.file.ReaderFactory`
3. `org.aoc.sample.phone.directory.entry.StringValuesToEntryBinder`
4. `org.io.buffered.file.line.Reader`
5. `org.aoc.sample.phone.directory.entry.Entry`

Among those five components, the first two components can be reused in any application which needs to manipulate files and therefore has higher reuse potential than the other components. The reuse potential of the orther components is limited to only applications which manipulate phone directories.

10.5.3 How software construction, evolution and maintenance becomes easier

Software construction, evolution and maintenance using the AOC model become easier for two main reasons:

1. **Small size components**: Breaking big issues into smaller ones is a common mechanism to solve complex issues. Hence, decomposition of software computational tasks into small sized code elements makes software construction easier. Constructing software according to the AOC model requires AOC components to have small size full fledged components. From a readability and comprehension viewpoint, constructing concise small functions is easier than constructing larger ones. For instance, the pool of variables which exist in the context a coarse or large size behavioral component is more often bigger than the one existing in the context of a small behavioral component. As a result, there is less chance to make inadvertent or erroneous manipulation of the variables which exist in the context of a small component. Furthermore, testing small size components is relatively easier than testing coarse or large grained components for the simple reason that there is less logic to test. The same argument holds for maintaining and evolving components with small sizes.
2. **Selective composition of sub-composites at application construction time:** Suppose a new requirement has emerged and requires the addition of a person's address to the phone directory. It is interesting to observe the impact of adding this requirement to both the second and fourth prototypes. The reason for choosing the second prototype is that, according to our experience, it represents fairly what a software construct might construct in an industrial setting.

   To illustrate how application construction, evolution, and maintenance become easier using the AOC model, it is interesting to consider the impact of introducing the new requirement from two viewpoints: addition of the requirement with no reuse and addition of the requirement via reuse of preexisting components.

With no reuse, the addition of the requirement to prototype 2 requires:

- Modification of the component `Entry` so that it has structures to hold the address details as emphasized in Table 10.30.
- Modification of the method `bindValuesToEntry` to bind the address from the corresponding values read from the file (see emphasized instructions of Table 10.31).
- Modification of the method `displayDirectory` to display the address with other information of the entry (see emphasized instructions of Table 10.31).

Table 10.30 Prototype 2 code elements part-1 modified according to the new requirement

```
package org.ac.sample.phoneDirectory;

public class Entry {

      ...
    private String streetAddress;
    private String postalCode;
    private String city;
    private String province;
    private String country;
      ...
}
```

Table 10.31 Prototype 2 code elements part-2 modified according to the new requirement

```
public class PhoneDirectoryApp{

...

    private Entry bindValuesToEntry(String[] values) {

        Entry entry = new Entry();
        entry.setFirstName(values[0]);
        entry.setLastName(values[1]);
        entry.setInternationalCode(values[2]);
        entry.setAreaCode(values[3]);
        entry.setPhone(values[4]);
        entry.setStreetNumber(new Integer(values[5]));
            entry.setStreetName(values[6]);
        entry.setPostalCode(values[7]);
        entry.setCity(values[8]);
```

Table 10.31 Prototype 2 code elements part-2 modified according to the new requirement (continued)

```
        entry.setProvince(values[9]);
        entry.setCountry(values[10]);
        return entry;
    }

      ...

    public static void displayDirectory(List<Entry> entries) {
        for (Entry entry : entries) {
            System.out.println(entry.getFirstName() + " "
              + entry.getLastName() + ", "
               + entry.getInternationalCode()
               + "(" + entry.getAreaCode() + ")"
              + entry.getPhone()
               + entry.getStreetNumber + " "
              + entry.getStreetName() + " "
              + entry.getPostalCode() + " "
              + entry.getCity() + " ".
              + entry.getProvince() + " "
              + entry.getCountry());
        }
    }
      ...
}
```

With no reuse, i.e. component derived from the new requirement do not exist and have to be constructed, the addition of the requirement to prototype 4 requires:

- Creation of the `Address` component, its composition descriptor and its related components (`Factory` and `Displayer`) in a new package **`org.aoc.sample.address`** as shown in Table 10.32.
- Modification of the `Entry` component composition descriptor and its related components: `Displayer`, `StringValuesToEntryBinder` and `Factory` as emphasized in Table 10.33.

Table 10.32 Prototype 4 new components as per the new requirement

```
package org.aoc.sample.address;

public class Address {

    public Integer streetNumber;
    public String streetName;
    public String postalCode;
    public String city;
    public String province;
    public String country;
    public Displayer displayer;
}


package org.aoc.sample.address;

public class Displayer {

    public static void display(Address address) {
        System.out.print(address.streetNumber + " ");
        System.out.print(address.streetName + " ");
        System.out.print(address.postalCode + " ");
        System.out.print(address.city + " ");
        System.out.print(address.province + " ");
        System.out.print(address.country + "\n");
    }

}


package org.aoc.sample.address;

public class Factory {

    public static Address create(Integer streetNumber,
            String streetName, String postalCode,
            String city, String province, String country) {

        Address address = new Address();
        address.streetName = streetName;
        address.streetNumber = streetNumber;
        address.postalCode = postalCode;
        address.city = city;
        address.province = province;
        address.country = country;
        return address;
     }

}
```

Table 10.33 Prototype 4 modified code elements according to the new requirement

```
package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.address.Address;
import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.directory.person.Person;
import org.aoc.sample.phone.directory.person.PersonFactory;

public class StringValuesToEntryBinder {
    public static Entry bindStringValuesToEntry(String[] values) {

        Person person = PersonFactory.create(values[0].trim(),
                values[1].trim());
        Phone phone = org.aoc.sample.phone.Factory.create(
                Integer.parseInt(values[2].trim()),
                Integer.parseInt(values[3].trim()),
                Integer.parseInt(values[4].trim()));
        Address address = org.aoc.sample.address.Factory.create(
                new Integer(values[5]), values[6], values[7], values[8],
                values[9], values[10]);
        Entry entry = Factory.create(person, phone, address);
        return entry;
     }
}

package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.address.Address;
import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.directory.person.Person;

public class Factory {

    public static Entry create(Person person, Phone phone,
            Address address) {
         Entry entry = new Entry();
         entry.person = person;
         entry.phone = phone;
         entry.address = address;
         return entry;
    }
}

package org.aoc.sample.phone.directory.entry;

public class Displayer {

    public static void display(Entry entry) {
        entry.person.displayer.display(entry.person);
        entry.phone.displayer.display(entry.phone);
        entry.address.displayer.display(entry.address);
    }
}
```

Table 10.33 Prototype 4 modified code elements according to the new requirement (continued)

```
package org.aoc.sample.phone.directory.entry;

public class Entry {
    public org.aoc.sample.phone.Phone phone;
    public org.aoc.sample.phone.directory.person.Person person;
    public org.aoc.sample.address.Address address;


    public org.aoc.sample.phone.directory.entry.Displayer displayer;
}

//  Compositiong descriptor for the Address component
// stateful components

public Integer streetNumber;
public String streetName;
public String postalCode;
public String city;
public String province;
public String country;

// behavioral components
public Displayer displayer
```

However with reuse, the addition of the requirement to prototype 2 requires the same modifications applied to the application as in the case of no reuse as emphasized in Table 10.30. The reason behind this is that the software constructor has to copy the reusable instructions from some other preexisting component and amalgamate them with application code.

With reuse, the addition of the requirement to prototype 2 requires only a subset of the modifications applied to application in the case of no reuse which can be summarized as:

- Modification of the `Entry` component composition descriptor and its related components: `Displayer`, `StringValuesToEntryBinder` and `Factory` as emphasized in Table 10.34.

Table 10.34 Modification with reuse according to the new requirement for prototype 4

```
package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.address.Address;
import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.directory.person.Person;
import org.aoc.sample.phone.directory.person.PersonFactory;

public class StringValuesToEntryBinder {

    public static Entry bindStringValuesToEntry(String[] values) {

        Person person = PersonFactory.create(values[0].trim(),
                values[1].trim());
        Phone phone = org.aoc.sample.phone.Factory.create(
                Integer.parseInt(values[2].trim()),
                Integer.parseInt(values[3].trim()),
                Integer.parseInt(values[4].trim()));
        Address address = org.aoc.sample.address.Factory.create(
                new Integer(values[5]), values[6], values[7],
                values[8], values[9], values[10]);
        Entry entry = Factory.create(person, phone, address);
        return entry;
    }
}



package org.aoc.sample.phone.directory.entry;

import org.aoc.sample.address.Address;
import org.aoc.sample.phone.Phone;
import org.aoc.sample.phone.directory.person.Person;


public class Factory {

    public static Entry create(Person person, Phone phone,
            Address address) {
        Entry entry = new Entry();
        entry.person = person;
        entry.phone = phone;
        entry.address = address;
        return entry;
    }
}



package org.aoc.sample.phone.directory.entry;

public class Displayer {
```

Table 10.34 Modification with reuse according to the new requirement for prototype 4 (continued)

```
 public static void display(Entry entry) {
        entry.person.displayer.display(entry.person);
        entry.phone.displayer.display(entry.phone);
        entry.address.displayer.display(entry.address);
    }
}


// Composition descriptor of the Entry component
// Stateful ocmponents
    public org.aoc.sample.phone.Phone phone;
    public org.aoc.sample.phone.directory.person.Person person;
    public org.aoc.sample.address.Address address;


// behavioral components
   public org.aoc.sample.phone.directory.entry.Displayer displayer;
```

Observing the modifications done on prototype 2 and prototype 4 of the phone directory application, a number of counting results has been obtained as shown in Table 10.35. Those counting results are used in discussing and demonstrating how the AOC model makes software construction, evolution and maintenance easier.

Analyzing the results illustrated in Table 10.35, it can be concluded that constructing application according to the AOC model without any reuse is costly in terms of development effort. In other words, a cost has to be paid upfront when constructing AOC model components. This cost is necessary to ready AOC components for reuse. So that later on AOC software application maintenance and evolution becomes easier and less costly in terms of development effort.

Table 10.35 Modification counting results as a result of introducing the new requirement

	With no reuse		With reuse	
	Prototype 2	**Prototype 4**	**Prototype 2**	**Prototype 4**
Count of added components	0	3	0	0
Count of modified components	2	4	2	4
Count of added instructions	11	26	11	5

For instance, modifying prototype 2 of the phone directory application to add the address requirement, without making any code reuse, necessitates the addition of 11 instructions as shown in Table 10.35. Similarly, modifying prototype 4 of the phone directory application to add the same requirement, without making any code reuse, necessitated 26 instructions. On the contrary, modifying prototype 2 of the phone directory application to add the address requirement, with code reuse, necessitates the addition of 11 instructions. However, modifying prototype 4 of the phone directory application to add the same requirement, without code reuse, necessitated 5 instructions.

In conclusion, when pools of reusable AOC components exist, software construction, maintenance and evolution based on that pool of components is significantly less costly than traditional software construction.

10.5.4 How the AOC model promotes software construction to a higher level of software composition

AOC model higher level of software composition is achieved via:

- Performing composition of sub-composites in a fluid and straightforward manner. For instance, the body of the composition descriptor of a component acts as a composition template in which are declared the sub-composite which satisfies the application requirements. Observing the content of composition descriptors illustrated in Table 10.10, Table 10.16 and Table 10.22 demonstrates how fluid and straightforward is the inclusion and exclusion of sub-composites. For example, to remove the `Displayer` component from any of those components, the line containing the declaration of that component has to be removed.
- Allowing the composition of reusable sub-composite components at application construction, thus giving the software constructor much flexibility as to the sub-composite components the host component composes or ignores.
- Usage of behavioral components as full fledged components. For example the `Displayer` components listed in the composition descriptor of Table 10.10, Table 10.16 and Table 10.22 are examples of such behavioral full fledged components.

10.5.5 How AOC applications can be constructed from existing applications

Prototype 3 forms the basis of any application which needs to be constructed according to the AOC model. This prototype has passed several refactoring steps so that its components and their methods exhibit characteristics similar to those exhibited by AOC components. To construct an AOC compliant application from prototype 3 of the phone directory application, the methods of prototype 3 components have to be promoted to full fledged components. Furthermore, composition descriptors have to be constructed as templates of the various sub-composites those components use as illustrated in prototype 4 of the phone directory application.

10.6 Conclusion

In this chapter, a case study was built to illustrate how software complexity is decreased, software reuse is increased, and how software construction, evolution, and maintenance

become easier using the AOC model. Furthermore, examples of higher level composition have been illustrated. Finally, a brief discussion on how AOC based applications can be constructed from a preexisting application was presented.

CHAPTER 11

VERIFICATION AND VALIDATION

11.1 INTRODUCTION

Conducting verification and validation is an essential step to verify objectively the solution approach. While verification deals with the accuracy of a theory or an approach, validation concerns the pertinence and meaningfulness of a theory or approach (Warrel, 2001). The validation process result provides an indicator to which level the new theory is applicable into practice. Validation of solutions or approaches can be performed by measuring and comparing the results derived from the application of those approaches and solutions against some reference results.

Solutions and approaches which lead to automated systems such as algorithms can be validated quantitatively with considerable precision. For instance, when measuring the performance of a computational algorithm, the time taken by the algorithm is compared against the time taken by alternative algorithms assuming the same input and output are consumed and produced by those algorithms. However, unautomated approaches and solutions such as software construction paradigms are often validated qualitatively since fewer quantitative measures can be applied in such cases.

11.2 Verification and validation criteria

The assessment of the research work and results presented in this report are performed according to the following aspects defined in (Olesen, 1992):

1. *Internal logic:* Deals with the assessment of the relevance and connection between the conceived theory and its relationships to existing theoretical work and the connection between the conceived theory and results.

2. *Truth*: Concern assessing whether the conceived theory and solution approaches can be used to describe real life issues.
3. *Acceptance*: Refers to assessing whether the solution approaches, results and tools produced by the research are accepted and used by other researchers and practitioners.
4. *Applicability*: Refers to the evaluation of the applicability of the conceived theory and research output in practice.
5. *Novelty*: Refers to evaluation of whether the solution approaches introduced new solution approaches to deal with the research problem.

11.3 Assessment

To perform the verification and validation assessment, the research issues as well as the objectives set forth in this research work provide a significant resource of information. The assessment results are described mostly in a qualitative manner with fewer results provided with quantitative measures.

11.3.1 Internal logic

This criterion is useful particularly in determining whether the research issues originate in the research literature and are aligned with the presented solutions.

In this respect, it is fairly easy to conduct the assessment. The research issues tackled in this research are: a) unwanted component's members, b) chaotic composition and amalgamation of code elements, c) suboptimal component reuse, and d) component version mismatches. Each of those research issues is identified after considerable effort of sources investigation, analysis and study of previous works originating from the literature on software components. For instance, the unwanted component's member issue is reported in (Al-Hatali and Walton, 2002) where compositional wrappers are suggested as a solution to hide unwanted members.

Even though this solution does hide unwanted members, it is has some limitation since those hidden members can be accessed and misused. Furthermore, the solution inflates the code size, a situation undesirable in both resources limited environments and heavily networked applications where code has to be transferred across the wires. The alternative solution suggested in this research provides an alternative solution by allowing selective composition of a component's members at application construction time, thus eliminating any bloating and risk of misuse caused by unwanted component members.

In conclusion, the research conducted in this thesis fulfills the criterion of internal logic.

11.3.2 Truth

This criterion assesses whether the issues addressed by the research project are real. Additionally, it assesses whether the suggested approach do p rovide a solution as it is claimed.

Do the issues addressed present a real research challenge? Assessing whether the research issues present real problems and challenges can be verified by looking at the goal of the research project. The goal of the research project is to provide a better approach to address software growing complexity, increase software reuse and promote software construction by composition of components. The answer is yes since the quest to achieve this goal has been tackled by researchers over the past few years with proposals of solutions having provided many enhancements (OO and component oriented paradigms), but none providing a satisfactory overall solution. Assessing whether the individual limitations tackled in this research work present a real research challenge can be verified:

1. By examining the presence of the issues in the research literature, i.e. the issues are tackled by other researchers.
2. Through demonstration of occurrence scenarios.

The assessment results of the limitation tackled in this research are provided in Table 11.1.

Table 11.1 Issues present real challenges-Literature support

Issue	Tackled by researchers	Occurrence Scenario
Unwanted component's members	Yes (Al-Hatali and Walton, 2002)	(section 3.3.1)
Chaotic composition and amalgamation of code elements	Yes (Humphrey, 2006)	(section 3.3.2)
Suboptimal component reuse	(Ferrett and Offutt, 2002) (Mili *et al.*, 1999a; Mili *et al.*, 1999b; Shiva and Shala, 2007)	(section 3.3.3)
Component version mismatches	(OSGi, 2009)	(section 3.3.4)

Do the suggested approach and tools provide the solution it is claimed? How can that be validated? Verification is done through demonstrations of usage scenarios where the solution approaches and tools provide remedies to those issues. Furthermore, a case study has been performed (see chapter 10) to demonstrate how the AOC model meets the goal of this resarch work.

Table **11.2** summarizes the issues, the solution approaches and tools provided by the research project, the method of verification and the verification outcome.

Table 11.2 Validation methods and outcome

Issues	**Approach, tool and solution**	**Truthfulness verification method**	**Verification outcomes**
Unwanted component's members	1. AOC model selective composition. 2. AOC compiler 3. Component's unused member measurement method (CUMM) 4. Automatic tool to measure the count and size of the unwanted members	1. Demonstration through implementation of sample application and compilation 2. Demonstration of how the CUMM method can be used. 3. Demonstration of the tool to measure component unused members	1. AOC compiler Works as specified 2. CUMM tool works with some restriction as explained in chapter 6
Chaotic composition and amalgamation of code elements	AOC model atomic and optimally modular components	Demonstration through implementation of a sample application	Works as specified
Suboptimal component reuse	AOC component model atomic and optimally modular components	Demonstration through implementation of a sample application	Works as specified

Table 11.2 Validation methods and outcome (continued)

Component version mismatches	1. Component version scheme 2. Component version mismatch detector	1. Demonstration through a sample application. 2. Demonstration of the version mismatch detector on a sample application	Works as specified

11.3.3 Acceptance

Assessing the acceptance of the approaches and tools proposed in this research by the research community is done by examining whether the approaches and tools have been presented in conference publications. The assessment result is positive. The first approach, the measurement method, the measurement tool, and the component versioning scheme suggested in this research project have been presented in international conferences as shown in Table 11.3. The subsequent work was only completed recently and still has to be submitted to refereed international conferences.

A suitable assessment of the acceptance of the approaches and tools provided in this research would probably measure the availability and how much widespread are the tools in the public domain. For the time being, the tools are not available publicly but such a move will be seriously considered in the future.

Table 11.3 Publications

Approach or Tool	Conferences
Component selective composition	Euromicro Conference, 2004 (Msheik, Abran and Lefebvre, 2004)
Component's unused member measurement method (CUMM)	14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon, 2004 (Msheik *et al.*, 2004)
Automatic tool to measure the count and size of the unwanted members	IEEE International Conference on Computer Systems and Applications, 2006. (Msheik *et al.*, 2006)
Component version mismatches	IEEE International Symposium on Industrial Electronics, 2006. (Msheik and Abran, 2006)

11.3.4 Applicability

Assessing the applicability of the approaches and the tools provided in this research would typically be performed by measuring how easy the approaches and tools can be applied to reap the benefits they provide. Table 11.4 suggests strategies for the assessment of the approaches and tools proposed in this research project.

Table 11.4 Applicability – assessment strategy

Proposed approach and/ or tool	Proposed assessment strategy
AOC model selective composition	Applying component selective composition in the construction of software application for newly created components and applications is straightforward provided that the components comply with the AOC

Table 11.4 Applicability – assessment strategy (continued)

	model. For each compositional component, the software constructor has to select the sub-composite components according to application requirements. For each unavailable stateful component the software constructor has to construct the structure of the component. For each unavailable behavioral component, the software constructor has to write the atomically or optimally modular code for that component. It is worth mentioning that writing atomically and optimally modular code is faster than writing traditional code since such code is small in size and thus much easier to comprehend, modify and test. Finally, the software constructor has to compile the application using the AOC compiler to produce out the application executable code. In conclusion, applying the AOC model to construct application can be considered as a more productive approach than traditional application construction. On the other hand, it should be mentioned that writing atomic and modular component might take longer time to produce than writing chaotically amalgamated code. Nevertheless, from a reuse perspective modular component is better than less reusable code. For already existing COTS (none AOC compliant) components, it is harder to construct applications using the AOC approach, since doing so requires the conversion of those COTS component libraries to AOC compliant component libraries. The conversion effort is considerable given the volume of existing industrial components.
Measurement of component unused	Applying the measurement tool requires little effort; the user has to run the CoMet tool on the application code. When the

Table 11.4 Applicability – assessment strategy (continued)

members using the CUMM	tool completes the measurement process it will generate a measurement report which the user can analyze and use in his decision making process.
Component version mismatch scheme and detector	Applying and making use of the component versioning scheme is straightforward and simple. The constructor is required to provide the complete version information (major, minor and micro) of each component used by the application. The versioning information has to be annotated to the declaration of the component. When the construction of the application is done, the software constructor has to run the component version mismatch detector on the constructed application code.

11.3.5 Novelty

Assessing the novelty elements provided by the approaches and tools proposed in this research project is illustrated in Table 11.5.

Table 11.5 Novelty elements

Approach and/ or tool	**Elements of novelty**
AOC model	1. Selective composition. Using selective composition, only the required components will be present in the deployed application. Furthermore, selective composition provides components with the ability to inherit or

Table 11.5 Novelty elements (continued)

	disinherit sub-composite components as dictated by application requirements. 2. Compositional software construction. Application construction is performed partially via component composition rather than instructional composition. In short, the AOC model provides a shift in the way software construction is performed. 3. Atomic and enhanced modularity components. AOC components are constructed to be small in size so that they exihibit atomic or enhanced modularity. 4. Better reuse. Better reuse is attained as a consequence of atomic and enhanced modularity. 5. Loosely coupled state and behavior. AOC stateful and behavioral components are defined each with its own structure, leading thus to loose coupling between state and behavior.
Component's unused member measurement method (CUMM)	1. Measurement of unused member method. CUMM is a measurement method developed using sound principles to measure the number of unused members of traditional OO software applications. 2. Automation of the measurement process. The measurement of component unwanted members is automated so that developers can rapidly obtain measurement results and act accordingly.
Component version scheme and component versions mismatch detector	1. Component version mismatches. The versioning scheme provides a way to detect version mismatches on the granular level of components. 2. Automatic detection of mismatches. An automatic mismatch detection tool is provided to help software developers detect version mismatches and then perform the right adjustment rapidly.

11.4 Conclusion

In this chapter, a verification and validation of the research project in terms of proposed approaches, method tools have been presented. The verification and validation process was performed according to the following aspects: internal logic, truth, acceptance, applicability, and novelty.

CONCLUSION

In an era marked by continuous technological evolution, the demand for software continues to grow in tandem with the complexity of software. On the bright side, the result is, an unprecedented evolution of virtually every aspect of the human life that touches and exploits the ingenious tool of software. On the somber side, from a software construction perspective, the result is an increasing demand for software, increasing complexity of software, monolithic amalgamation of software code which in turn leads to less modular, less comprehensible, less reusable and less maintainable code.

Over the past two decades, researchers have come up with new approaches, programming languages, constructs, design patterns, and best practices so that the complexity of software is reduced, its reuse is better exploited, its comprehension becomes easier and faster, its testing and maintenance become easier and less costly and finally its construction and time to market become much shorter. For instance, functional languages introduced the principle of software functional decomposition following the mathematical notion of functions. Equally, functional languages blurred the distinction between data and code. Both data and code are treated equally and interchangeably through an interesting and unique mechanism typical of functional languages. This mechanism is highly appreciated due to its practical usefulness in the construction of software. Additionally, software constructed using functional languages tend to be less buggy since functions are usually side effect-free. In spite of these advantages provided by functional languages, they have less industrial adoption primarily due to performance inefficiencies compared to other types of programming languages.

By contrast, procedural languages which promote software structural decomposition have better performance than functional programming languages. However, procedural languages are less efficient when it comes to capturing real life abstraction compared to the way OOP does. It is this inefficiency of procedural languages which primarily has paved the way to the succeeding programming language type, notably the OOP language type.

An OOP language uses the metaphor of real word objects to decompose the different interacting parts of software. By and large, the characteristic of capturing a real word object in term of an equivalent software object is behind the wide industrial adoption of OOP languages. As their predecessors, OOP languages have their own limitations. While objects provide an interesting mechanism to increase static software reuse (code reuse) they do not easily support dynamic reuse in terms of objects distribution across heterogeneous environments and platforms.

Recently, CBSC which can be considered as an extension to OOP has been widely adopted to address the issues of object composition and distribution (CORBA, COM/DCOM/.NET, EJB, Java beans).

Undeniably, significant progress can be observed, using the above mentioned programming languages, in particular OOP and CBSC. However, BCSC suffers from several issues. Among those issues and limitations, the following ones are addressed in this research project.

Tackled research issues and limitations

During software construction, code composition plays a key role in the manifestation of the limitations tackled in this research project and which are:

1. Unwanted component's members. During the construction of different applications, various subsets of a component's members are used according to the context and requirements of each application. Unfortunately, a number of unwanted members might be present in the code of the application though they are not needed.
2. Chaotic composition and amalgamation of code elements. Even though OOP languages, and by extension CBSC, rely considerably on the real word object metaphor to perform software construction using an object and component decomposition approach, the presence of chaotic amalgamation of code chunks is inevitable and leads to less modular and reusable code.

3. Suboptimal component reuse. The inherent structural nature of objects and components is rigid enough to prevent the reuse of a component or object behavioral members without reusing the whole object or component. In other words, behavioral members are subordinate and cannot be reused on their own and consequently leading to suboptimal software reuse.
4. Component versions mismatch. Component evolution leads to the construction of different component versions due to legacy dependencies or simply due to implementation variations. As a result component versions mismatch occur and can lead to faulty behaviors.

Proposed theory

To address the above listed limitation, the AOC model is proposed as a remedy to the first three limitations listed above. Equally, a component versioning scheme is also proposed to remedy the fourth and last of those limitations.

The AOC model is designed as an overall and straightforward solution based on atomic and optimally modular component composition. In this respect, the idea behind proposing the AOC model is to go back to the basics and reshape the amalgamation of code in the form of atomically and optimal modular components to ultimately promote software construction to a higher level relying increasingly on software components composition rather than on code chunk amalgamation. Making use of elementary structures in the form of atomic and optimally modular components will potentially result in optimal software reuse.

The AOC model borrows from other approaches and programming languages. For instance, it promotes and uses behaviors as full fledged components on their own and therefore allows for better component composition and software reuse. Additionally, AOC components are basically objects and thus bear objects capabilities such as inheritance, polymorphism, and encapsulation.

Research deliverables

The objectives set forth and realized in this research project are:

1. Development of a proposition and implementation of the AOC model.
2. Design of a measurement method to measure component unwanted members.
3. Development of a measurement tool prototype to measure component unused members.
4. Development of a component versioning scheme to remedy for component version mismatches.
5. Development of a prototype implementation of the component version mismatch detector.

Research Limitations

This research project aimed to improve the current software construction approaches and methods by addressing and providing solutions to a number of limitations. In particular, the AOC model seeks to improve the efficiency of organizing code chunks into atomic and optimally modular components and leveraging those components to attain ultimately optimal software reuse.

It is obvious, that adoption of the AOC model by software constructors is crucial to the adoption and deployment of the AOC model as well to the exploitation and capitalization on the benefits and advantages provided by this component model.

Similarly, constructing atomic and optimally modular components is a synonym of fine grained AOC components. Constructing AOC model compliant components results in significantly higher cardinality of components in comparison to construction components according to traditional software construction models. Additionally, fine grained components would translate in further development effort and costs and potential less efficient code in terms of performance.

Even though, increased efforts and costs are most likely to occur in the short term, the long term benefits and advantages outweigh the initial costs simply because reuse, comprehension, testing and maintenance are easier to do. In the same vein, atomic and optimally modular components provide an interesting trend and incitation towards standard components.

As to the less efficient code in terms of performance, optimization techniques such as code in-lining can be used and can significantly reduce such performance inefficiencies.

Future work

Since fine-grained and components with enhanced modularity will most likely result in considerably high cardinality of components; such proliferation and abundance of components requires therefore management for development, search, identification, testing, maintenance and reuse in newer applications. Obviously, those issues provide a potential for additional research work.

Another interesting research effort would target the evolution of components over time. For instance, what might be considered an application level component today may becomes part of component library in the coming years and decades. In this sense, the issues which need further investigation and study concern the organization of components in AOC component spaces and how components would evolve in terms of versions, names and identifiers.

APPENDIX I

COMET: CUMM MEASUREMENT TOOL PROTOYPE CODE

The content of this appendix is provided on a CDROM in a directory called Appendix I.

APPENDIX II

COMPONENT VERSIONS MISMATCH DETECTOR TOOL CODE

The content of this appendix is provided on a CDROM in a directory called Appendix II.

APPENDIX III

GRAMMAR TO GENERATE THE AOC COMPILER PARSER

```
/**
 * The grammar found in this file represent a subset of the original file.
 * It had been adapted by Hamdan Msheik to construct the the AOC compiler.
 *
 * An ANTLRv3 capable Java 1.5 grammar for building ASTs.
 *
 * Note that there's also the tree grammar 'JavaTreeParser.g' that can be
fed
 * with this grammer's output.
 *
 *
 * Please report any detected errors or even suggestions regarding this
grammar
 * to
 *
 *          dieter [D O T] habelitz [A T] habelitz [D O T] com
 *
 *      with the subject
 *
 *          jsom grammar: [your subject note]
 *
 * To generate a parser based on this grammar you'll need ANTLRv3, which
you can
 * get from 'http://www.antlr.org'.
 *
 *
 * This grammar is published under the ...
 *
 * BSD licence
 *
 * Copyright (c) 2007-2008 by HABELITZ Software Developments
 *
 * All rights reserved.
 *
 * http://www.habelitz.com
 *
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *  1. Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *  2. Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the
distribution.
```

```
 *  3. The name of the author may not be used to endorse or promote
products
 *     derived from this software without specific prior written
permission.
 *
 * THIS SOFTWARE IS PROVIDED BY HABELITZ SOFTWARE DEVELOPMENTS ('HSD')
``AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL 'HSD' BE LIABLE FOR ANY DIRECT,
INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA,
 * OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */

grammar Java_composites;

options {
    backtrack = true;
    memoize = true;
    output = AST;
    ASTLabelType = CommonTree;
}

tokens {

    // operators and other special chars

    AND                     = '&'       ;
    AND_ASSIGN              = '&='      ;
    ASSIGN                  = '='       ;
    AT                      = '@'       ;
    BIT_SHIFT_RIGHT         = '>>>'     ;
    BIT_SHIFT_RIGHT_ASSIGN  = '>>>='    ;
    COLON                   = ':'       ;
    COMMA                   = ','       ;
    DEC                     = '--'      ;
    DIV                     = '/'       ;
    DIV_ASSIGN              = '/='      ;
    DOT                     = '.'       ;
    DOTSTAR                 = '.*'      ;
    ELLIPSIS                = '...'     ;
    EQUAL                   = '=='      ;
    GREATER_OR_EQUAL        = '>='      ;
```

```
GREATER_THAN            = '>'               ;
INC                     = '++'              ;
LBRACK                  = '['               ;
LCURLY                  = '{'               ;
LESS_OR_EQUAL           = '<='              ;
LESS_THAN               = '<'               ;
LOGICAL_AND             = '&&'              ;
LOGICAL_NOT             = '!'               ;
LOGICAL_OR              = '||'              ;
LPAREN                  = '('               ;
MINUS                   = '-'               ;
MINUS_ASSIGN            = '-='              ;
MOD                     = '%'               ;
MOD_ASSIGN              = '%='              ;
NOT                     = '~'               ;
NOT_EQUAL               = '!='              ;
OR                      = '|'               ;
OR_ASSIGN               = '|='              ;
PLUS                    = '+'               ;
PLUS_ASSIGN             = '+='              ;
QUESTION                = '?'               ;
RBRACK                  = ']'               ;
RCURLY                  = '}'               ;
RPAREN                  = ')'               ;
SEMI                    = ';'               ;
SHIFT_LEFT              = '<<'              ;
SHIFT_LEFT_ASSIGN       = '<<='             ;
SHIFT_RIGHT             = '>>'              ;
SHIFT_RIGHT_ASSIGN      = '>>='             ;
STAR                    = '*'               ;
STAR_ASSIGN             = '*='              ;
XOR                     = '^'               ;
XOR_ASSIGN              = '^='              ;

// keywords

ABSTRACT                = 'abstract'        ;
ASSERT                  = 'assert'          ;
BOOLEAN                 = 'boolean'         ;
BREAK                   = 'break'           ;
BYTE                    = 'byte'            ;
CASE                    = 'case'            ;
CATCH                   = 'catch'           ;
CHAR                    = 'char'            ;
CLASS                   = 'class'           ;
CONTINUE                = 'continue'        ;
DEFAULT                 = 'default'         ;
DO                      = 'do'              ;
DOUBLE                  = 'double'          ;
ELSE                    = 'else'            ;
ENUM                    = 'enum'            ;
EXTENDS                 = 'extends'         ;
FALSE                   = 'false'           ;
FINAL                   = 'final'           ;
FINALLY                 = 'finally'         ;
```

```
FLOAT                   = 'float'           ;
FOR                     = 'for'             ;
IF                      = 'if'              ;
IMPLEMENTS              = 'implements'      ;
INSTANCEOF              = 'instanceof'      ;
INTERFACE               = 'interface'       ;
IMPORT                  = 'import'          ;
INT                     = 'int'             ;
LONG                    = 'long'            ;
NATIVE                  = 'native'          ;
NEW                     = 'new'             ;
NULL                    = 'null'            ;
PACKAGE                 = 'package'         ;
PRIVATE                 = 'private'         ;
PROTECTED               = 'protected'       ;
PUBLIC                  = 'public'          ;
RETURN                  = 'return'          ;
SHORT                   = 'short'           ;
STATIC                  = 'static'          ;
STRICTFP                = 'strictfp'        ;
SUPER                   = 'super'           ;
SWITCH                  = 'switch'          ;
SYNCHRONIZED            = 'synchronized'    ;
THIS                    = 'this'            ;
THROW                   = 'throw'           ;
THROWS                  = 'throws'          ;
TRANSIENT               = 'transient'       ;
TRUE                    = 'true'            ;
TRY                     = 'try'             ;
VOID                    = 'void'            ;
VOLATILE                = 'volatile'        ;
WHILE                   = 'while'           ;

// tokens for imaginary nodes

ANNOTATION_INIT_ARRAY_ELEMENT;
ANNOTATION_INIT_BLOCK;
ANNOTATION_INIT_DEFAULT_KEY;
ANNOTATION_INIT_KEY_LIST;
ANNOTATION_LIST;
ANNOTATION_METHOD_DECL;
ANNOTATION_SCOPE;
ANNOTATION_TOP_LEVEL_SCOPE;
ARGUMENT_LIST;
ARRAY_DECLARATOR;
ARRAY_DECLARATOR_LIST;
ARRAY_ELEMENT_ACCESS;
ARRAY_INITIALIZER;
BLOCK_SCOPE;
CAST_EXPR;
CATCH_CLAUSE_LIST;
CLASS_CONSTRUCTOR_CALL;
CLASS_INSTANCE_INITIALIZER;
CLASS_STATIC_INITIALIZER;
CLASS_TOP_LEVEL_SCOPE;
```

```
    CONSTRUCTOR_DECL;
    ENUM_TOP_LEVEL_SCOPE;
    EXPR;
    EXTENDS_BOUND_LIST;
    EXTENDS_CLAUSE;
    FOR_CONDITION;
    FOR_EACH;
    FOR_INIT;
    FOR_UPDATE;
    FORMAL_PARAM_LIST;
    FORMAL_PARAM_STD_DECL;
    FORMAL_PARAM_VARARG_DECL;
    FUNCTION_METHOD_DECL;
    GENERIC_TYPE_ARG_LIST;
    GENERIC_TYPE_PARAM_LIST;
    INTERFACE_TOP_LEVEL_SCOPE;
    IMPLEMENTS_CLAUSE;
    LABELED_STATEMENT;
    LOCAL_MODIFIER_LIST;
    JAVA_SOURCE;
    METHOD_CALL;
    MODIFIER_LIST;
    PARENTESIZED_EXPR;
    POST_DEC;
    POST_INC;
    PRE_DEC;
    PRE_INC;
    QUALIFIED_TYPE_IDENT;
    STATIC_ARRAY_CREATOR;
    SUPER_CONSTRUCTOR_CALL;
    SWITCH_BLOCK_LABEL_LIST;
    THIS_CONSTRUCTOR_CALL;
    THROWS_CLAUSE;
    TYPE;
    UNARY_MINUS;
    UNARY_PLUS;
    VAR_DECLARATION;
    VAR_DECLARATOR;
    VAR_DECLARATOR_LIST;
    VOID_METHOD_DECL;
}

@header {
package ac.compiler;
}

@members {

    private boolean mMessageCollectionEnabled = false;
    private boolean mHasErrors = false;
    private List<String> mMessages;

    /**
     *  Switches error message collection on or of.
     *
```

```
     *  The standard destination for parser error messages is
<code>System.err</code>.
     *  However, if <code>true</code> gets passed to this method this
default
     *  behaviour will be switched off and all error messages will be
collected
     *  instead of written to anywhere.
     *
     *  The default value is <code>false</code>.
     *
     *  @param pNewState  <code>true</code> if error messages should be
collected.
     */
    public void enableErrorMessageCollection(boolean pNewState) {
        mMessageCollectionEnabled = pNewState;
        if (mMessages == null && mMessageCollectionEnabled) {
            mMessages = new ArrayList<String>();
        }
    }

    /**
     *  Collects an error message or passes the error message to <code>
     *  super.emitErrorMessage(...)</code>.
     *
     *  The actual behaviour depends on whether collecting error messages
     *  has been enabled or not.
     *
     *  @param pMessage  The error message.
     */
     @Override
    public void emitErrorMessage(String pMessage) {
        if (mMessageCollectionEnabled) {
            mMessages.add(pMessage);
        } else {
            super.emitErrorMessage(pMessage);
        }
    }

    /**
     *  Returns collected error messages.
     *
     *  @return  A list holding collected error messages or
<code>null</code> if
     *           collecting error messages hasn't been enabled. Of course,
this
     *           list may be empty if no error message has been emited.
     */
    public List<String> getMessages() {
        return mMessages;
    }

    /**
     *  Tells if parsing a Java source has caused any error messages.
     *
```

```
     * @return <code>true</code> if parsing a Java source has caused at
least one error message.
     */
    public boolean hasErrors() {
        return mHasErrors;
    }
}

@lexer::header {
package ac.compiler;
}

@lexer::members {
/**
 * Determines if whitespaces and comments should be preserved or thrown
away.
 *
 * If <code>true</code> whitespaces and comments will be preserved within
the
 * hidden channel, otherwise the appropriate tokens will be skiped. This
is
 * a 'little bit' expensive, of course. If only one of the two behaviours
is
 * needed forever the lexer part of the grammar should be changed by
replacing
 * the 'if-else' stuff within the approprate lexer grammar actions.
 */
public boolean preserveWhitespacesAndComments = false;
}

// Starting point for parsing a Java file.
componentSource
    :   classScopeDeclarations*
        -> ^(JAVA_SOURCE classScopeDeclarations*)
    ;


classScopeDeclarations
    :    modifierList
        (  type classFieldDeclaratorList SEMI
            -> ^(VAR_DECLARATION modifierList type
classFieldDeclaratorList)
        )
    |   SEMI!
    ;

modifierList
    :   modifier*

        -> ^(MODIFIER_LIST modifier*)
    ;

modifier
    :   PUBLIC
    |   PROTECTED
```

```
    |   PRIVATE
    |   STATIC
    |   ABSTRACT
    |   NATIVE
    |   SYNCHRONIZED
    |   TRANSIENT
    |   VOLATILE
    |   STRICTFP
    |   FINAL
    ;

type
    :   simpleType
    |   objectType
    ;

simpleType // including static arrays of simple type elements
    :   primitiveType arrayDeclaratorList?
        ->  ^(TYPE primitiveType arrayDeclaratorList?)
    ;

objectType // including static arrays of object type reference elements
    :   qualifiedTypeIdent arrayDeclaratorList?

        ->  ^(TYPE qualifiedTypeIdent arrayDeclaratorList?)
    ;

primitiveType
    :   BOOLEAN
    |   CHAR
    |   BYTE
    |   SHORT
    |   INT
    |   LONG
    |   FLOAT
    |   DOUBLE
    ;

qualifiedTypeIdent
    :   typeIdent (DOT typeIdent)*
        ->  ^(QUALIFIED_TYPE_IDENT typeIdent+)
    ;

typeIdent
    :   IDENT^ genericTypeArgumentList?

    ;

genericTypeArgumentList
    :   LESS_THAN genericTypeArgument (COMMA genericTypeArgument)*
genericTypeListClosing
        ->  ^(GENERIC_TYPE_ARG_LIST[$LESS_THAN, "GENERIC_TYPE_ARG_LIST"]
genericTypeArgument+)
    ;
```

```
genericTypeArgument
    :   type
    |   QUESTION genericWildcardBoundType?
        ->  ^(QUESTION genericWildcardBoundType?)
    ;

genericWildcardBoundType
    :   (EXTENDS | SUPER)^ type
    ;

genericTypeListClosing  // This 'trick' is fairly dirty - if there's some
time a better solution should
                        // be found to resolve the problem with nested
generic type parameter lists
                        // (i.e. <T1 extends AnyType<T2>> for generic type
parameters or <T1<T2>> for
                        // generic type arguments etc).
    :   GREATER_THAN
    |   SHIFT_RIGHT
    |   BIT_SHIFT_RIGHT
    |   // nothing
    ;


classFieldDeclaratorList
    :   classFieldDeclarator (COMMA classFieldDeclarator)*
        ->  ^(VAR_DECLARATOR_LIST classFieldDeclarator+)
    ;

classFieldDeclarator
    :   variableDeclaratorId
        ->  ^(VAR_DECLARATOR variableDeclaratorId)
    ;

variableDeclaratorId
    :   IDENT^ arrayDeclaratorList?
    ;

arrayDeclarator
    :   LBRACK RBRACK
        ->  ^(ARRAY_DECLARATOR)
    ;

arrayDeclaratorList
    :   arrayDeclarator+
        ->  ^(ARRAY_DECLARATOR_LIST arrayDeclarator+)
    ;



WS  :  (' '|'\r'|'\t'|'\u000C'|'\n')
    {
        if (!preserveWhitespacesAndComments) {
            skip();
        } else {
```

```
            $channel = HIDDEN;
        }
    }
    ;


COMMENT
    :   '/*' ( options {greedy=false;} : . )* '*/'
    {
        if (!preserveWhitespacesAndComments) {
            skip();
        } else {
            $channel = HIDDEN;
        }
    }
    ;

LINE_COMMENT
    : '//' ~('\n'|'\r')* '\r'? '\n'
    {
        if (!preserveWhitespacesAndComments) {
            skip();
        } else {
            $channel = HIDDEN;
        }
    }
    ;

IDENT
    :   JAVA_ID_START (JAVA_ID_PART)*
    ;

fragment
JAVA_ID_START
    :  '\u0024'
    |  '\u0041'..'\u005a'
    |  '\u005f'
    |  '\u0061'..'\u007a'
    |  '\u00c0'..'\u00d6'
    |  '\u00d8'..'\u00f6'
    |  '\u00f8'..'\u00ff'
    |  '\u0100'..'\u1fff'
    |  '\u3040'..'\u318f'
    |  '\u3300'..'\u337f'
    |  '\u3400'..'\u3d2d'
    |  '\u4e00'..'\u9fff'
    |  '\uf900'..'\ufaff'
    ;

fragment
JAVA_ID_PART
    :  JAVA_ID_START
    |  '\u0030'..'\u0039'
    ;
```

APPENDIX IV

GRAMMAR TO GENERATE THE AOC COMPILER AST WALKER

```
/**
 * The grammar found in this file represent a subset of the original file.
 * It had been adapted by Hamdan Msheik to construct the the AOC compiler.
 *
 * For more information see the head comment within the 'java.g' grammar
file
 * that defines the input for this tree grammar.
 *
 * BSD licence
 *
 * Copyright (c) 2007-2008 by HABELITZ Software Developments
 *
 * All rights reserved.
 *
 * http://www.habelitz.com
 *
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 *
 *  1. Redistributions of source code must retain the above copyright
 *     notice, this list of conditions and the following disclaimer.
 *  2. Redistributions in binary form must reproduce the above copyright
 *     notice, this list of conditions and the following disclaimer in the
 *     documentation and/or other materials provided with the
distribution.
 *  3. The name of the author may not be used to endorse or promote
products
 *     derived from this software without specific prior written
permission.
 *
 * THIS SOFTWARE IS PROVIDED BY HABELITZ SOFTWARE DEVELOPMENTS ('HSD')
``AS IS''
 * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO,
THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL 'HSD' BE LIABLE FOR ANY DIRECT,
INDIRECT,
 * INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING,
BUT NOT
 * LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE,
DATA,
 * OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY
OF
 * LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING
```

```
 * NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
SOFTWARE,
 * EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 *
 */
tree grammar Java_compositesTreeParser;

options {
    backtrack = true;
    memoize = true;
    tokenVocab = Java_composites;
    ASTLabelType = CommonTree;
    output = template;
}


@treeparser::header {
package ac.compiler;
}

@treeparser::members {

    boolean mMessageCollectionEnabled = false;
    private boolean mHasErrors = false;
    List<String> mMessages;

    /**
     *  Switches error message collection on or of.
     *
     *  The standard destination for parser error messages is
<code>System.err</code>.
     *  However, if <code>true</code> gets passed to this method this
default
     *  behaviour will be switched off and all error messages will be
collected
     *  instead of written to anywhere.
     *
     *  The default value is <code>false</code>.
     *
     *  @param pNewState  <code>true</code> if error messages should be
collected.
     */
    public void enableErrorMessageCollection(boolean pNewState) {
        mMessageCollectionEnabled = pNewState;
        if (mMessages == null && mMessageCollectionEnabled) {
            mMessages = new ArrayList<String>();
        }
    }

    /**
     *  Collects an error message or passes the error message to <code>
     *  super.emitErrorMessage(...)</code>.
     *
     *  The actual behaviour depends on whether collecting error messages
     *  has been enabled or not.
```

```
     *
     *  @param pMessage  The error message.
     */
     @Override
    public void emitErrorMessage(String pMessage) {
        if (mMessageCollectionEnabled) {
            mMessages.add(pMessage);
        } else {
            super.emitErrorMessage(pMessage);
        }
    }

    /**
     *  Returns collected error messages.
     *
     *  @return  A list holding collected error messages or
<code>null</code> if
     *           collecting error messages hasn't been enabled. Of course,
this
     *           list may be empty if no error message has been emited.
     */
    public List<String> getMessages() {
        return mMessages;
    }

    /**
     *  Tells if parsing a Java source has caused any error messages.
     *
     *  @return  <code>true</code> if parsing a Java source has caused at
least one error message.
     */
    public boolean hasErrors() {
        return mHasErrors;
    }


    public void logLn(String string) {
       System.out.println(string);
    }


    public void log(String string) {
       System.out.print(string);
    }


}


// Starting point for parsing a Java file.
componentSource [String componentName, String extendedComponentName]
    :
^(JAVA_SOURCE  (c+=classScopeDeclarations)* ) ->
component(componentName={$componentName},
extendedComponentName={$extendedComponentName}, composites={$c})
```

```
    ;

classScopeDeclarations
    :   ^(VAR_DECLARATION
    mL=modifierList
    t=type
        fl=classFieldDeclaratorList) -> composite(modifiers={$mL.st},
type={$t.st}, fieldNameList={$fl.st})
    ;

modifierList
    :

     ^(MODIFIER_LIST  (mods+=modifier)* ) -> modifier(modifier={$mods})
    ;

modifier
    :    PUBLIC     -> modifier(modifier={$PUBLIC.text})
    |    PROTECTED  -> modifier(modifier={$PROTECTED.text})
    |    PRIVATE    -> modifier(modifier={$PRIVATE.text})
    |    STATIC     -> modifier(modifier={$STATIC.text})
    |    ABSTRACT   -> modifier(modifier={$ABSTRACT.text})
    |    NATIVE     -> modifier(modifier={$NATIVE.text})
    |    SYNCHRONIZED  -> modifier(modifier={$SYNCHRONIZED.text})
    |    TRANSIENT  -> modifier(modifier={$TRANSIENT.text})
    |    VOLATILE   -> modifier(modifier={$VOLATILE.text})
    |    STRICTFP   -> modifier(modifier={$STRICTFP.text})
    |    FINAL  -> modifier(modifier={$FINAL.text})
    ;

type
    :   st=simpleType -> type(type={$st.st})
    |   ct=objectType  -> type(type={$ct.st})
    ;

simpleType // including static arrays of simple type elements
    :   ^(TYPE p=primitiveType   (ad=arrayDeclaratorList)?) ->
simpleType(primitiveType={$p.st}, arrayDec={$ad.st})
    ;

objectType // including static arrays of object type reference elements
    :   ^(TYPE qti=qualifiedTypeIdent
    (ad=arrayDeclaratorList)?) -> objectType(typeIdent={$qti.st},
arrayDec={$ad.st})
    ;

primitiveType
    :   BOOLEAN -> primitiveType(type={$BOOLEAN.text})
    |   CHAR -> primitiveType(type={$CHAR.text})
    |   BYTE -> primitiveType(type={$BYTE.text})
    |   SHORT -> primitiveType(type={$SHORT.text})
    |   INT -> primitiveType(type={$INT.text})
    |   LONG -> primitiveType(type={$LONG.text})
    |   FLOAT -> primitiveType(type={$FLOAT.text})
```

```
    |   DOUBLE -> primitiveType(type={$DOUBLE.text})
    ;

qualifiedTypeIdent
    :   ^(QUALIFIED_TYPE_IDENT (id+=typeIdent)+) ->
qualifiedTypeIdentifier(identifier={$id})
    ;

typeIdent
    :   ^(id=IDENT  (gtal=genericTypeArgumentList)?) ->
typeIdentifier(identifierName={$id.text},
genericTypeArgumentList={$gtal.st})
    ;

genericTypeArgumentList
    :   ^(GENERIC_TYPE_ARG_LIST (gta+=genericTypeArgument)+) ->
genericTypeArgumentList(genericTypeArgument={$gta})
    ;

genericTypeArgument
    :   t=type -> type(type={$t.st})
    |  ^(QUESTION  (gwbt=genericWildcardBoundType)?) ->
genericTypeArgument(genericWildcardBoundType={$gwbt.st})
    ;

genericWildcardBoundType
    :   ^(EXTENDS   t=type) ->
extendsGenericWildcardBoundType(type={$t.st})
     |  ^(SUPER  t=type)    -> superGenericWildcardBoundType(type={$t.st})
    ;

classFieldDeclaratorList
    :   ^(VAR_DECLARATOR_LIST (cfd+=classFieldDeclarator)+) ->
classFieldDeclaratorList(classFieldDeclarator={$cfd})

    ;

classFieldDeclarator
    :   ^(VAR_DECLARATOR vdid=variableDeclaratorId) ->
classFieldDeclarator(variableDeclaratorId={$vdid.st})
    ;

variableDeclaratorId
    :   ^(id=IDENT (adl=arrayDeclaratorList)?) ->
variableDeclaratorId(identifier={$id}, arrayDecList={$adl.st})
    ;

arrayDeclarator
    :   lb=LBRACK  rb=RBRACK -> arrayDeclartor(leftBracket={$lb.text},
rightBracket={$rb.text})
    ;

arrayDeclaratorList

    :  { $st = %arrayDeclartorList(); }
```

```
    ^(ARRAY_DECLARATOR_LIST
      (ARRAY_DECLARATOR
      { if($ARRAY_DECLARATOR != null) {
$st.setAttribute("arrayDecList", "[]");
        }
      }
      )*
    )
    ;
```

APPENDIX V

AOC COMPILER CODE

The content of this appendix is provided on a CDROM in a directory called Appendix V.

BIBLIOGRAPHY

A.W., Appel. 1997. *Modern Compiler Implmentation in Java*. Cambridge University Press, 548 p.

Abran, A., J. Moore, P. Bourque and R. Dupuis. 2004. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE-Computer Society Press.

Achermann, Franz. 2002. ""Forms, Agents and Channels"". University of Bern, 235 p.

Achermann, Franz, Markus Lumpe, Jean-Guy Schneider and Oscar Nierstrasz. 2001. "Piccola - a Small Composition Language". In *Formal Methods for Distributed Processing - A Survey of Object-Oriented Approaches*. Cambridge Univesity Press.

Achermann, Franz, and Oscar Nierstrasz. 2002. *Software Architectures and Component Technology*, 648 vol.: Springer International.

Aho, A.V., R. Sethi and J.D. Ullman. 1986. *Compilers - Principles, Techniques and Tools*. Addison-Wesley Plublishing Co.

Al-Hatali, M.S., and H.G. Walton. 2002. "Smart Features for Compositional Wrappers". In *Workshop on Component-based Software Development Processes (ICSR7 2002)*. Austin, Texas.

Apache. 2001. "Byte Code Engineering Library". Online. <http://jakarta.apache.org/site/downloads/downloads_bcel.cgi>. Retrieved May 5, 2005.

Apache. 2008. "Apache common-io library". Online. <http://jakarta.apache.org >. Retrieved August 5, 2009.

Birngruber, Dietrich, and Johannes Kepler. 2001a. "CoML: Yet Another, But simple Component Composition Language". In *Workshop on Composition Languages*. Vienna, Austria.

Birngruber, Dietrich, and Johannes Kepler. 2001b. "A Software Composition Language and Its Implementation". In *Perspectives of System Informatics (PSI 2001)*. p. 519-529. Novosibirsk, Russia: LNCS 2244, Springer.

Brooks, F. P., Jr. 1987. "No Silver Bullet Essence and Accidents of Software Engineering". *Computer,* vol. 20, n° 4, p. 10-19.

C.Schmidt, D., D. L. Levine and S. Mungee. 1998. "The Design of the TAO Real-Time Object Request Broker". *IEEE Computer Communications Journal,* vol. 21, n° 4, p. 294-324.

Cai, Yuangfang, and Sunny Huynh. 2007 "An Evolution Model for Software Modularity Assessment
http://dx.doi.org/10.1109/WOSQ.2007.2 ". In *Proceedings of the 5th International Workshop on Software Quality* p. 3 IEEE Computer Society.

Clements, C. Paul. 1996. "From Subroutines to Subsystems: Component-Based Software Development". In *Component-Based Software Engineering: Selected Papers from the Software Engineering Institute*. Los Alamitos, CA: IEEE Computer Society Press.

Coker, L., and R. R. Hayes. 1997 Monterey, California. "Services First, Components Second!". In *6th International OMG-DARPA Workshop on Compositional Software Architectures*.

Coplien, James. 1992. *Advanced C++ Programming Styles and Idioms*. Hamburg: Addison-Wesley.

Dinakar, Karthik. 2009. "Agile development: overcoming a short-term focus in implementing best practices". In *Proceeding of the 24th ACM SIGPLAN conference companion on Object oriented programming systems languages and applications*. p. 579-588. Orlando, Florida, USA: ACM.

Durham, J. 2001. "History-making components". Online. <http://www.ibm.com/developerworks/webservices/library/co-tmline/>. Retrieved December 02, 2009.

Eclipse Foundation. 2008. "Eclipse IDE". Online. <http://www.eclipse.org/>. Retrieved February 02, 2009.

Exton, C. 1997. "Distributed fault tolerance specification through the use of interface definitions". In *Proceedings of the Conference on Technology of Object-Oriented Languages (TOOLS 24)*. p. 254-259.

Exton, C., D. Watkins and D. Thompson. 1997. "Comparisons between CORBA IDL & COM/DCOM MIDL: interfaces for distributed computing". In *Proceedings of the Conference on Technology of Object-Oriented Languages and Systems (TOOLS 25)*. p. 15-32.

Ferrett, L.K., and J. Offutt. 2002. "An empirical comparison of modularity of procedural and object-oriented software". In *Proceedings. Eighth IEEE International Conference on Engineering of Complex Computer Systems.* p. 173-182. Greenbelt, Maryland, USA.

Fowler, M., K. Beck, J. Brant, W Opdyke and D Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional.

Fraser, S., and D. Mancl. 2008. "No Silver Bullet: Software Engineering Reloaded". *Software, IEEE,* vol. 25, n° 1, p. 91-94.

Gagnon, E. 1998. "Sablecc an Object-Oriented Compiler Framework". McGill University.

Gamma, E., R. Helm, R. Johnson and J. Vlissides. 1994. *Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Ghosh, Sudipto. 2002. "Improving Current Component Development Techniques for Successful Component-Based Software Development". In *Workshop on Component-based Software Development Processes (ICSR7 2002)*.

Google. 2007. "Guice container". Online. <http://code.google.com/p/google-guice/>. Retrieved October 4, 2009.

Goplan, R.S. 1998. "A detailed comparaision of CORBA, DCOM and Java/RMI". <http://www.execpc.com/~gopalan/misc/compare.html>. Retrieved March 15, 2003.

Habelitz, Dieter. 2008. "A Java 1.5 grammar". Online. <http://www.antlr.org/grammar/1207932239307/Java1_5Grammars>. Retrieved November 17, 2009.

Habermann, A. N. 1988. "Programming environments for reusability". In *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences. Vol.II. Software Track.* Vol. 2, p. 1-10.

Heinemann, T., G., and T. Council, W. 2001. *Component-Based Software Engineering*. Addison-Wesely.

Hitz, M., and B. Montazeri. 1995. "Measuring Coupling and Cohesion In Object-Oriented Systems". In *Int. Symposium on Applied Corporate Computing*. Monterry, Mexico.

Hudak, P. 1989. "Conception, Evolution, and Application of Functional Programming Languages". In *ACM Computing Surveys.* Vol. Volume 21, p. 359-411.

Hughes, J. 1990. "Why Functional Programming Matters". *The Computer Journal,* vol. 32, n° 2.

Hull, M.E.C., P.N. Nicholl and Y. Bi. 2001. "Approaches to component technologies for software reuse of legacy systems". *IEEE Computing and Control Engineering Journal*, p. 281-287.

Humphrey, Watts. 2006. "A CAI State of the Practice Interview". Online. <http://www.itmpi.org/default.aspx?pageid=266.>. Retrieved May 10, 2009.

Humphrey, Watts S. 2001. "The Future of Software Engineering: II". Online. <http://www.sei.cmu.edu/library/abstracts/news-at-sei/wattsnew2q01.cfm>. Retrieved May 10, 2009.

IBM. 1998. "Bean Markup Language". Online. <http://www.alphaworks.ibm.com/tech/bml>. Retrieved January 4, 2004.

Institute of Electrical and Electronics Engineers. 1990. *IEEE standard glossary of software engineering terminology*. IEEE Std 610.12-1990. New York, N.Y.: Institute of Electrical and Electronics Engineers.

Interface21. 2003. "Spring Framework Lightweight container". Online. <http://www.springframework.org/about>. Retrieved August 5, 2009.

ISO/IEC. 2001. *Software engineering - Product quality - Part1: Quality model*. 9126-1:2001. ISO/IEC.

Jacobsen, H.-A., and B.J. Kramer. 2000. "Modeling interface definition language extensions". In *Proceedings of the 37th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS-Pacific 2000)*. p. 242-252.

Jacobson, I. 1998. "Component-based Development with UML". Online. <http://www.ibm.com/developerworks/webservices/library/co-tmline/>. Retrieved April 25, 2004.

Jacquet, J.-P., and A. Abran. 1997. "From Software Metrics to Software Measurement Methods: A Process Model". In *Third International Symposium and Forum on Software Engineering Standards (ISESS'97)* (2-6 June). p. 128-135. Walnut Creek, CA: IEEE Computer Society.

Johnson, Mark. 1999. "Bean Markup Language". Online. <http://www.javaworld.com/javaworld/jw-08-1999/jw-08-beans.html>. Retrieved December, 1 2009.

Johnson, Rod. 2002. *Expert One-on-One J2EE Design and Developement*. Wrox Press.

Kozaczynski, W., and G. Booch. 1998. "Component-based software engineering". *IEEE Software*, p. 34-36.

Krishnamurthy, B. 1994. "Software architecture and reuse-an inherent conflict?". In *Proceedings of the Third International Conference onSoftware Reuse: Advances in Software Reusability*. p. 215.

Kuhne, Thomas. 1999. *A Functional Pattern System for Object-Oriented Design*. Hamburg: Verlag Dr. Kovac.

Laplante, Philp A. 2007. *What Every Engineer Should Know about Software Engineering*. Florida: CRC Press Taylor & Francis Group.

Larman, G. 2004. *Applying UML and Patterns*. Prentice Hall.

Ledbetter, L., and B. Cox. 1985. "Software-ICs". *BYTE Magazine*, (June), p. 307-315.

Li, J. 1999. *A Survey on Microsoft Component-based Programming Technologies*. Montreal: Concordia University.

Li, L., and R. Abraham. 2002. "History of COBOL". Online. <http://www.csee.umbc.edu/courses/graduate/631/Fall2002/COBOL.pdf>. Retrieved August 5, 2009.

Lowy, Juval. 2003. *Programming .NET Components*. O'Reilly.

Maurer, Peter. 2003. *Component-Level Programming*. Pearson Education Inc.

Mcdirmid, S., M. Flatt and C. H. Wilson. 2001. "Jiazzi: New-Age Components for Old-Fashioned Java.". In *International Conference on Object Oriented Programming, Systems, Languages and Applications*.

McIlroy, M.D. 1968. "Mass Produced Software Components". In *NATO Software Engineering Conference*. Scientific Affairs Division.

Mens, T., and T. Tourwe. 2004. "A survey of software refactoring". *IEEE Transactions on Software Engineering*, vol. 30, n° 2, p. 126-139.

Microsoft Corporation, Digitial Equipement Corporation. 1995. "The Component Object Model Specification". Online. <http://www.microsoft.com/com/resources/specs.asp>. Retrieved January 10, 2005.

Mili, A., S. Yacoub, E. Addy and H. Mili. 1999a. "Toward an engineering discipline of software reuse". *Software, IEEE*, vol. 16, n° 5, p. 22-31.

Mili, H., J. Dargham, A. Mili, O. Cherkaoui and R. Godin. 1999b. "View programming for decentralized development of OO programs". In *Proceedings on Technology of Object-Oriented Languages and Systems, 1999*. p. 210-221.

Mili, H., F. Mili and A. Mili. 1995. "Reusing software: issues and research directions". *IEEE Transactions on Software Engineering*, vol. 21, n° 6, p. 528-562.

Milner, Robin. 1993. *Logic of Algebra and Specifications*. Springer-Verlag.

Mowbray, T.J., and w.A. Ruh. 1997. *Inside Corba: Distributed Object Standards and Applications*. Addison Wesly.

Msheik, H., and A. Abran. 2006. "Extending CSCM to support Interface Versioning". In *IEEE International Symposium on Industrial Electronics*. Vol. 4, p. 3311-3315.

Msheik, H., A. Abran and E. Lefebvre. 2004. "Compositional structured component model: handling selective functional composition". In *Proceedings of the 30th Euromicro Conference*. p. 74-81. Rennes, France.

Msheik, H., A. Abran, H. McHeick, D. Touloumis and A. Khelifi. 2006. "CoMet: A Tool Using CUMM to Measure Unused Component Members". In *IEEE International Conference on Computer Systems and Applications*. p. 697-703.

Msheik, H., A. Abran, H. Mcheik and P. Bourque. 2004. "Measuring Components Unused Functions". In *14th International Workshop on Software Measurement (IWSM) IWSM-Metrikon 2004, Konigs Wusterhausen* (2-5 November, 2004). p. 367-380. Magdeburg, Germany: Springer-Verlag.

Nada, N., L. Luqi, M. Shing, D. Rine, E. Damiani and S. Tuwaim. 2000. "Software reuse technology practices and assessment tool-kit". In *Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 34)*. p. 307-316.

Nierstrasz, O., and D. Tsichritzis. 1995. *Object-Oriented Software Composition*. Prentice Hall International (UK) Ltd.

Odersky, Martin. 2009. "The Scala Language Specification". Online. <http://www.scala-lang.org/node/198>. Retrieved May 10, 2009.

Olesen, J. 1992. "Concurrent Development in Manufacturing – Based on Dispositional Mechanisms". Technical University of Denmark.

OMG. 2001. "Introduction to OMG Specification". Online. <http://www.omg.org/gettingstarted/specintro.htm#OMA>. Retrieved January 10, 2005.

OMG. 2002. "Object Management Architecture". Online. <http://www.omg.org/oma/>. Retrieved January 10, 2005.

OMG. 2006. *CORBA Component Model Specification*. <http://www.omg.org/technology/documents/corba_spec_catalog.htm>.

Ongg, Jay. 1997. *An Architectural Comparison of Distributed Object Technologies*. MIT. <http://scanner-group.mit.edu/PDFS/OnggJ.pdf>.

OSGi, Alliance. 2009. "OSGi Service Platform Core Specification". Online. <http://www.osgi.org/Specifications/HomePage>. Retrieved May 10, 2009.

Parnas, D. L. 1972. "On the Criteria to be Used in Decomposing System into Modules". *Communications of the ACM*, vol. 15, n° 12, p. 1053-1058.

Parr, Terrence. 1996. *Language Translation Using PCCTS and C++*. Automata Plublishing Company.

Parr, Terrence. 2007. *The Definitive ANTLR Reference*. Pragmatic Bookshelf.

PicoContainer. 2008. "Pico Container Introduction". Online. <http://www.picocontainer.org/introduction.html>. Retrieved December 1, 2009.

Ramnivas, Laddad. 2003. *AspectJ in Action*. Manning Publications Co.

Rashid, A. 2001. "Aspect-oriented and component-based software engineering". *IEE Proceedings -Software*, vol. 148, n° 3, p. 87-88.

Rosemary, R. 1998. *DCOM Explained*. Digital Press.

Ruh, W., T. Herron and P. Klinker. 1999. *IIOP Complete: Understanding CORBA and Middleware Interoperability*. Addison Wesley.

Ryder, G. R., M.L. Soffa and M. Burnett. 2005. "The Impact of Software Engineering Research on Modern Progamming Languages". *ACM Transactions on Software Engineering and Methodology*, vol. 14, n° 4, p. 431-477.

SEI. 2003. "CBS Overview". Online. Software Engineering Institute. <http://www.sei.cmu.edu/cbs/index.html>. Retrieved January 10, 2005.

Sherlock, T.P., and G. Cronin. 2000. *COM Beyond Microsoft: Designing and Implementing COM Servers on Compaq Platfroms*. Digital Press.

Shiva, Sajjan G., and Lubna Abou Shala. 2007. "Software Reuse: Research and Practice". In *Fourth International Conference on Information Technology, ITNG '07*. p. 603-609.

Siegel, J. 2001. "What's Coming in CORBA 3". Online. Object Management Group, Inc. <http://www.omg.org/technology/corba/corba3releaseinfo.htm>. Retrieved January 10, 2005.

Sullivan, J. K. , G. W. Griswold, C. Yuanfang and Hallen B. 2001. "The structure and value of modularity in software design". In *Symposium on the Foundations of Software Engineering*. Vienna.

Sun. 1997. "JavaBeans". Online. <http://java.sun.com/products/javabeans/docs/beans.101.pdf>. Retrieved Decembre 10, 2005.

Sun. 1999. "Java 2 Platform, Entreprise Edition". Online. <http://java.sun.com/j2ee/sdk_1.2.1/techdocs/guides/j2eeoverview/Introduction.fm.html#7916>. Retrieved January 10, 2005.

Sun. 2002. "The J2EE Tutorial". Online. <http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Overview2.html#64857>. Retrieved January 10, 2005.

Sun. 2006. "JSR-000220 Enterprise JavaBeans v.3.0". Online. SUN MICROSYSTEMS, INC. <http://jcp.org/aboutJava/communityprocess/pfd/jsr220/index.html>. Retrieved December 1, 2009.

Sun. 2009. "JavaCC". Online. <https://javacc.dev.java.net/doc/docindex.html >. Retrieved Octobre 10, 2009.

Szyperski, C. 1998. *Component Software: Beyond Object-Oriented Programming*. New York: ACM Press and Addison-Wesley.

Tari, Z., and O. Buhkres. 2001. *Fundamentals of distributed object systems : The CORBA Prespective*. John Wiley & Sons, Inc.

Trifu, Adrian, and Urs Reupke. 2007. "Towards Automated Restructuring of Object Oriented Systems". In *11th European Conference on Software Maintenance and Reengineering (CSMR '07)*. p. 39-48.

Vanhelsuwe, L. 1997. *Mastering JavaBeans*. Sybex Inc.

Waldo, J. 1998. "The solution to the re-use problem". In *Proceedings of the Fifth International Conference on Software Reuse*. p. 369-370.

Warrel, A. 2001. "Design Syntactics: A Functional Approach to Visual Product Form – Theory, Models and Methods". Göteborg, Sweden, Chalmers University of Technology.

Watkins, D., and D. Thompson. 1998. "Adding semantics to interface definition languages". In *Proceedings of the Australian Software Engineering Conference*. p. 66-78.

Weerawarana, Sanjva, Francisco Curbera and Mathew J. Duftler. 2001. "A Composition Language for JavaBeans Components". In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems*. San Antanio, Texas.

Yu, Y., J. Mylopoulos, E. Yu, J. Cesar, L. Linda and L. Liu. 1998. " Software refactoring guided by multiple soft-goals". In *36th Annual Meeting of the Association for Computational Linguistics and 17th International Conference on Computational Linguistics*.

Zhao, J., and B. Xu. 2004. "Measuring Aspect Cohesion". In *Proc. Fundamental Approaches to Software Engineering*. Barcelona, Spain.

Zuse, H. 1997. *A Framework of Software Measurement*. Berlin: Walter de Gruyter.