BUSINESS PROCESS AUTOMATION WITH MODEL DRIVEN DEVELOPMENT

A Thesis Submitted to the
Graduate School of Natural and Applied Sciences of Dokuz Eylül University
In Partial Fulfillment of the Requirements for the Master of Science in
Computer Engineering

by
Emrecan SEZEN

September, 2010
İZMİR

M.Sc THESIS EXAMINATION RESULT FORM

We have read the thesis entitled **“BUSINESS PROCESS AUTOMATION WITH MODEL DRIVEN DEVELOPMENT”** completed by **EMRECAN SEZEN** under supervision of **ASST. PROF. DR. DERYA BİRANT** and we certify that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

……………………………………………………………

Asst. Prof. Dr. Derya Birant

Supervisor

…………………………………………………………… ……………………………………………………………

Prof. Dr. Alp Kut

(Jury Member)

Asst. Prof. Dr. Reyat Yılmaz

(Jury Member)

Prof.Dr. Mustafa SABUNCU
Director
Graduate School of Natural and Applied Sciences

ACKNOWLEDGMENTS

I would like to thank to my supervisor, Asst. Prof. Dr. Derya Birant, for her support, supervision and useful suggestions throughout this study. I am also highly thankful to Dr. Kökten Ulaş Birant for his valuable suggestions throughout study.

I owe my deepest gratitude to my family. This thesis would not have been possible unless their unflagging love and support.

Special thanks are directed to TUBITAK for their financial support throughout two years. Thanks to TUBITAK BIDEB, I could attain necessary hardware and software equipments and I could allocate more time to my thesis.

Emrecan SEZEN

BUSINESS PROCESS AUTOMATION WITH MODEL DRIVEN DEVELOPMENT

ABSTRACT

Model is an abstraction of a system or a part of it and provide us a way to control and manage systems from higher abstraction levels. Model Driven Architecture (MDA) is a framework defined by Object Management Group (OMG) to use visual models as a single resource for software developers. Abstraction levels has reached the highest point in software industry with the evolution of MDA framework. MDA defines some abstraction levels in models where each level corresponds to a different concern. Abstraction levels from the highest to the lower can be listed as follows: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM).

Aside from modeling, there are still programming problems, which are named cross-cutting concerns, that can not be solved clearly using traditional programming techniques such as Object Oriented Programming (OOP). Aspect Oriented Programming (AOP) approach provides some mechanisms to solve these cross-cutting concerns in a more effective and modular way. Aspect-oriented constructs: join point, pointcut, advice, aspect, aspect weaver make it possible to better program cross-cutting concerns. AspectJ is the most popular general purpose AOP implemention which offers a great deal of power and improved modularity.

Because the most attention is being made on the programming languages level (AspectJ, Hyper/J ... etc) in AOP environment, this has been an hindrance on the evolution of Aspect Oriented Modeling (AOM) which aims to make modeling of aspects in an easy and effective way. AOM is still in process and there are works for AOM which focus on techniques for the identification, analyses, management and representation of cross-cutting concerns in the modeling phase. Because solutions from different tool vendors are different than each other, this hinders the full adoption of Aspect Oriented Modeling to the software modeling area.

Keywords: Model Driven Architecture, Model Driven Development, Aspect Oriented Software Development, Aspect Oriented Modeling, Software Engineering

İŞ SÜREÇLERİNİN MODEL YÖNELİMLİ YAZILIM GELİŞTİRME METODOLOJİSİ İLE OTOMASYONU

ÖZ

Model bir sistemin ya da sistem parçasının soyut halidir ve modeller aracılığı ile sistemlerin daha yüksek soyutlama seviyelerinden kontrolleri ve yönetimleri sağlanır. Model Yönelimli Mimari (MDA) yazılım süreçlerindeki tüm aşamaları tek bir görsel model üzerinden yürütmeyi amaçlayan ve standartları Object Management Group (OMG) tarafından belirlenen bir mimaridir. MDA ile birlikte yazılımdaki soyutlama seviyeleri tarihindeki en yüksek noktaya gelmiş bulunmaktadır. MDA yazılımdaki soyutlama seviyelerini birden fazla seviyeye bölerek, her bir seviyenin kendine has problemler ile ilgilenmesi fikrini getirmiştir. Bu soyutlama seviyeleri en soyuttan en az soyuta olacak şekilde şöyle sıralanabilir: bilgisayar bağımsız model (CIM), platform bağımsız model (PIM), platform bağımlı model (PSM).

Görsel modellemedeki gelişmeler bir yana, şu aşamada halen mevcut programlama yaklaşımları ile modüler biçimde çözülemeyen problemler bulunmaktadır. Bu problemlerin mevcut programlama yaklaşımları ile çözümünde, mevcut uygulama tekrarlayan kod bloklarından dolayı karışık ve anlaşılması zor bir hal almaktadır. Bu problemlere enine kesen (cross-cutting) problemler denmektedir. İlgi Yönelimli Programlama (AOP), bu türden problemlerin yazılım seviyesinde modüler biçimde çözümü için etkili mekanizmalar içermektedir. Bu mekanizmalara birleşim noktaları, birleşim nokta kümeleri, tavsiyeler, ilgiler, ilgi dokuyucuları örnek olarak gösterilebilir. AOP yaklaşımını gerçekleştiren en yaygın ve hakim programlama dili AspectJ dilidir. Bu dil aracılığı ile, mevcut uygulamayı enine kesen problemlerin çözümü kodda karışıklık sağlamayacak şekilde yapılabilmektedir.

İlgi yönelimli programlama ile ilgili çalışmaların sadece programlama dilleri seviyesinde kalmasından dolayı, programlama dillerine göre bir üst soyutlama seviyesinde yer alacak olan, ilgilerin görsel olarak modellenmesi konusuna yeterli ilgi gösterilmemiştir. Bu konu ile ilgili çeşitli çalışmalar bulunmakla birlikte, standartlaşmanın henüz olmamasından dolayı, bir üründe üretilen bir modelin başka bir üründe kullanılmasında zorluklar yaşanmaktadır. Bu durum da, ilgilere yönelik modellemenin ilerlemesine engel oluşturmaktadır.

Anahtar sözcükler: Model Güdümlü Mimari, Model Güdümlü Geliştirme, Cepheye Yönelik Yazılım Geliştirme, Cepheye Yönelik Modelleme, Yazılım Mühendisliği

CONTENTS

Page

CHAPTER ONE
INTRODUCTION

1.1 Introduction

"Model is an abstraction of a system or a part of it." (Gally, 2007). Models provide us a way to control and manage systems from higher abstraction levels. The history of modeling is as long as the history of programming.

Today, abstraction levels in software projects are at the highest point in the history. Programmers do not program anymore, instead they use visual models to design their projects.

Model Driven Development (MDD) is an approach to software development to use visual models as a single resource for software developers. All the task is done on the visual model and other software related outputs (source code, documentation, test code ... etc.) are automatically generated from the source model. This dramatically increases software product quality and software developer's productivity. Model Driven Architecture (MDA) defines standarts for Model Driven Development.

Model Driven Architecture (MDA) process defines some abstraction levels in models where each level corresponds to a different concern. Abstraction levels from the highest to the lower can be listed as follows: Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM).

Aside from modeling, there are still programming problems, which are named cross-cutting concerns, that can not be solved clearly using traditional programming techniques such as Object Oriented Programming (OOP). Aspect Oriented Programming technique provides some mechanisms to solve cross-cutting concerns in a more effective and modular way. Some of these mechanisms are such that: join point, pointcut, advice, aspect, aspect weaver, inter-tpe declaration, context exposing ...etc. These mechanisms make it possible to better program cross-cutting concerns by seperately specifying these concerns and then weave or compose them together into a coherent implementation.

Aspect Oriented Programming is growing rapidly and it is used in many areas, such as middle-ware, security, fault tolerance, quality of service, and operating systems ...etc. AOP is not yet a fully mature discipline and needs to be used in more applications to be improved.

Gregor Kiczales and colleagues at Xerox PARC developed AspectJ as the most popular general purpose AOP implementation and made it available in 2001. Aspect Oriented Programming in language AspectJ offers a great deal of power and improved modularity.

For an experienced Java developer to become familiar with AspectJ language syntax is so simple because the AspectJ language uses Java programming language as base. When AspectJ language specific constructs are learned, writing the whole aspect code is the composition of the Java code and AspectJ language constructs.

Because the most attention is being made on the programming languages level (AspectJ, Hyper/J ... etc) in AOP environment, this has been an hindrance on the evolution of Aspect Oriented Modeling (AOM) which aims to make modeling of aspects in an easy and effective way. There are works in AOM which focus on techniques for the identification, analyses, management and representation of cross-cutting concerns in the modeling phase by using UML extension mechanisms (UML Profiles). Becasue UML does not support Aspect Oriented Modeling in its standard specification, UML profiles generated by different tool vendors are different than each other. This hinders the full adoption of Aspect Oriented Modeling to the software modeling area. As a result, model transformation languages lack stability and maturity to deal with UML Profiles.

We have designed and developed an aspect modeler tool named AspectModeler that allows the definition of aspects in a practical and efficient way. This tool eliminates some steps that are necessary in other approaches. AspectModeler has a number of interfaces; base model and aspect model representation, pointcut modeling, pointcut list, advice modeling, advice list, aspect definition where each of the interfaces plays a critical role to make aspect modeling easier. The output of AspectModeler is an aspect file that is a valid AspectJ language code. To run the existing software system with modelled aspects is simply as putting the generated aspect file into the existing project's design environment.

This thesis is divided into 5 chapters. In Chapter 1, the aim of the thesis is introduced. In Chapter 2, MDA and MDA related standards are discussed detailed. In Chapter 3, AOP approach and AspectJ as an AOP implementation are discussed detailed. In Chapter 4, Aspect Oriented Modeling (AOM), related works for AOM and our aspect modeler tool AspectModeler are discussed detailed. Last chapter presents conclusion.

CHAPTER TWO

MDA

2.1 MDA Definition

“Model is an abstraction of a system or a part of it.” (Gally, 2007). Models provide us a way to control and manage systems from higher abstraction levels. The history of modeling is as long as the history of programming.

First programmers coded instructions to the computer in 1s and 0s which is named machine code. Machine code only consist of 1s and 0s and different bit patterns correspond to different CPU (Central Processing Unit) instructions. Programmers had to know very good knowledge about the CPU to correctly program it. Because first programs were small and CPU capabilities were limited, it was possible to program machine by using machine code.

An important software innovation called assembly language made programmers job easy to program. Assembly language had an higher abstraction level than machine language. It was more human readable than machine language. With the rise of assembly language programmers were able to use simple mnemonics that the computer understands such *MOV AX, DX* to move data from the D register to the A register. A tool named assembler converted these simple mnemonics to 1s and 0s instead of programmers. Assembly language also made programs less prune to errors. Becase programs were less prune to errors, the quality level increased. Assembly language made writing more comprehensive programs such as payroll, billing ... etc. in shorter times. The maintenance of the programs were also easy becasue the language itself.

With the advent of third-generation languages(3 GLs) abstraction levels dramatically increased in programming. These languages, such as FORTRAN and COBOL, were more powerful, more abstract, more human readable. Like assemblers, these programming languages had language compilers to translate higher-level instructions to machine codes. Instructions in these languages were abstract and understandable but this was not the same as these intructions' machine code equivalents. A simple instruction could be represented with tens or hundreds lines of machine codes.

Programmers productivity increased with the increase of abstraction levels in programming. Becase programming languages were more abstract, they were more closer to the human thinking. For example, *PRINT* instruction in FORTRAN describes the functionality of itself. But this is not the same as *MOV AX, DX* instruction in assembly language if you don't know the meanings of registers. Programmers started to spent less time to find the correct instruction set with the increase of abstraction levels. So the productivity also increased.

Today, abstraction levels in software projects are at the highest point in the history. Programmers do not program anymore, instead they use visual models to design their projects. "Visual models are simple sketches that don't contain any relevant information of the way the project is to be implemented." (Igor & Jadranka, 2007).

In todays competitive business environment, software products must easily adopt to new changes to be successful. Software firms that do not continuously improve the operation, products, and services of their business do not have any chance to play a role in the industry. With visual modeling approach, software products can easily adopt new changes without any consistency problem. Because software developers see the big picture with the help of visual models, consistency of software behaviour is under control.

When working in a software project, we can draw pictures to visually model the behaviour of a system or a part of it. Models are just pictures in this scenario and we can not refer any semantic information by using these pictures. In this condition, program codes are manually written by programmers with a programming language. In this scenario, pictures can only be used for documentation to help other software related business people to understand software behavour. But this removes the automation of some task in the project lifecycle. If there would be a visual model as a single resource for software developer to automate some task in the project lifecycle (source code generation, documentation, ... etc.), it would certainly be better.

Model Driven Development (MDD) is an approach to software development to use visual models as a single resource for software developers. All the task is done on the visual model and other software related outputs (source code, documentation, test code ... etc.) are automatically generated from the source model. This dramatically increases software product quality and software developer's productivity. Model Driven Architecture (MDA) defines standarts for Model Driven Development.

Each abstraction level in programming history can be summarized as in Figure 2.1.

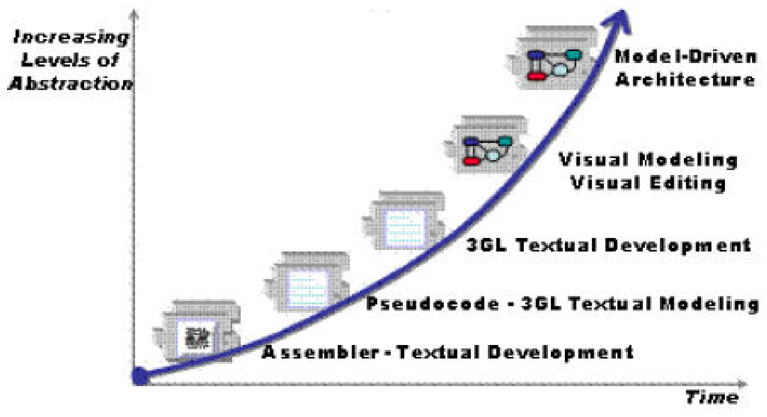

Increasing Levels of Abstraction
Model-Driven Architecture
Visual Modeling Visual Editing
3GL Textual Development
Pseudocode - 3GL Textual Modeling
Assembler - Textual Development
Time

Figure 2.1 Abstraction levels in programming history

Model Driven Development (MDD) keeps you away from coding the program manually. The generation of program code is done later in the software development process by software tools that support MDD aproach. By the use of MDD approach, it is possible to concentrate on the problem domain rather than implementation details. Because the software industry produced powerful tools that support MDD approach, a big portion of the program code can automatically be generated easily.

A case study on the use of MDA at Deutsche Bank Bauspar shows how MDD approach is reducing the software ownership costs at major international corporations (Watson, (n.d.)). This project involved a maintenance upgrade to an eighties-vintage COBOL back-office mainframe running CICS & DB2, marrying it to a Web–based front-end to give users sitting at 30.000 client machines in 1250 Deutsche Bank offices across Germany access to details of their savings-and-loan accounts. A dozen or so different kinds of software artifacts, including COBOL modules, Oracle database schemas, DB2 database schemas, EJBs, XML-RPC interface definitions and Junit test classes were all created using MDA techniques, with around 60% of the new business logic and 90% of the database-related code being created this way. Figure 2.2 shows code generation ratios at different layers of the project.

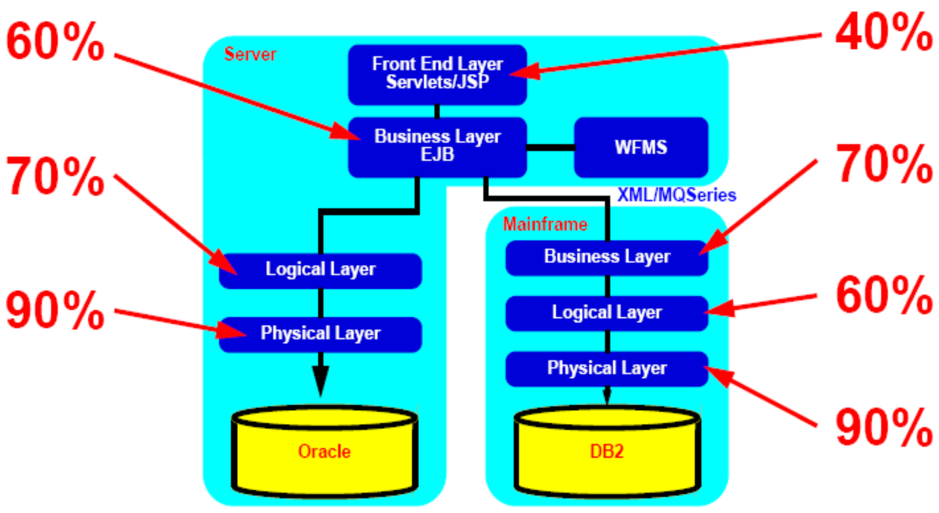


Figure 2.2 Code generation ratios in Deutsche Bank’s MDA application

The MDA technique seperates the system functionality from the implementation details. MDA is a framework for Model Driven Development (MDD) defined by Object Management Group (OMG). The focus is on the modeling task in MDA. You create model first that is not understandable by the computers which is named Computation Independent Model (CIM). After CIM you create model that can be processed by computers which is named Platform Independent Model (PIM), this model does not contain any platform specific information. After PIM you are ready to create a model that has platform specific details which is named Platform Specific Model (PSM). The conversion from PIM to PSM is done automatically by tools that support MDA. When PSM is created, code generation can also automaticaly be done without any human intervention. Software developers focus on the domain model rather than writing thousand lines of program codes.

2.2 Advantages of MDA

Advantages of MDA can be summarized as follows:

- MDA-enabled tools automate the task of generating PSM from PIM and then generating program code from PSM. This means that software developers focus more on the domain model design and less on the implementation details. This approch makes producing new applications faster, better and cheaper.
- Not only business requirements transformed into the final implementation (i.e. program code), but also non-business functional requirements such as security, scalability issues are also transformed into the final implementation.
- Code generated by MDA-enabled tools is undoubtedly has a very high quality. Code generated by MDA-enabled tools derives from libraries based on patterns designed by the industry's best developers. Meaning that

implementation has a very qualified architecture to do the best for performance, scalability, security, ...etc.

- MDA applications are portable because of the platform independent model feature. Platform independent models designed by MDA-enabled tools can easily be converted to platform specific representation that consist of the current platform's technological details.
- "OMG has 15 years' experience creating modeling standards, and includes all the major modeling tools vendors and a host of end-user organisations in diverse domains from manufacturing to medicine. " (Watson, (n.d.)). This feature of OMG makes MDA future-proof. When new platforms are introduces, OMG will add mappings for these new platforms into the MDA framework. And then tool ventors will implement these specifications in their MDA enabled tools. So, MDA framework and MDA enabled tools will continue to stay uptodate in future. Tool vendors have a critical role for this issue.
- In an MDA development project, attention focuses first on the application's business functionality and behaviour. This property of MDA makes software products more value-added for the customer. Value-added product makes the customer ready to become a bigger player in the industry. In MDA technique, implementation (technical) details of the software project are also critical but secondary to business functions. A big portion of these implementation details can be generated automatically by MDA enabled tools. Software developer sometimes may hand-code some technical details that MDA-enabled tools can not generate automatically.
- MDA application can interoperate. Because MDA is a standart of specifications, MDA enabled tools follow these standarts to become compatible with MDA framework. This feature makes MDA applications interoperable, meaning that any MDA application can be moved from one MDA enabled platform to another without any problem.

- MDA solves the documentation issue in a practical way. Keeping the documents up to date with code is tedious and time-consuming. But with MDA the models, code, and documentation are always in synchronization. As an example The *PathMATE Documentation Map* (Duby, 2003) generates custom documentation containing the models and their associated descriptions.
- MDA allows the participation of system analysts, stakeholders, also customers in a part of software development process. For example, Computation Independent Model (CIM) of the business functionality can be modelled, reviewed, refined with the contributions of system analysts, stakeholders, customers. With this approach, some design errors can be detected early in the design phase.

2.3 Disadvantages of MDA

- An MDD supporting infrastructure must clearly define how the model's elements represent real-world elements and software artifacts. Increased complexity of the modeling languges can be a disadvantage for MDA. Tool vendors must continuously take user feedbacks about which symbols are easy to understand and which are not to make modeling easy.
- Generation of PSM from PIM, and generation of program code from PSM is automated or semi-automated by MDA-enabled tools. Software developer sometimes may hand-code some technical details that MDA-enabled tools can not generate automatically. In these situations, manually editing the program code cannot be easy for the developer. Because the code generated by MDA-enabled tools derives from libraries based on patterns designed by the industry's best developers, sometimes it may be difficult to understand the program code before editing. Meaning that, higher conversion rates makes

MDA-enabled tools more successful from the software developer's perspective.

- MDA is standart framework defined by Object Management Group (OMG). OMG has a concortium that includes all the major modeling tools vendors and a host of end-user organisations in diverse domains. If tool vendors does not exactly conform to standards established by OMG, this will be a disadvantage for MDA applications' interoperating feature.
- OMG continues to work about standardization issues. These issues makes tool vendors' implementations special to their products only. This is also a disadvantage for MDA applications' interoperating feature.

2.4 MDA Process

In the MDA approach there are four major steps:

- Generation of a Computation Independent Model (CIM)
- Building a Platform Independent Model (PIM)
- Transforming the PIM into a Platform Specific Model (PSM)
- Generate the code out of the PSM

A CIM shows exactly what the system is expected to do. It hides all the information technology details to remain independent of how the system will be implemented. CIM viewpoint has the highest abstraction level because it presents only what the system expected to do. This feature of CIM makes it possible to bridge the gap between domain experts and information technologists responsible for implementing the system. In an MDA application, CIM should have enough information in it to be able to build an exact PIM. Figure 2.3 shows a CIM example that presents the overall sale process from marketing to delivery.

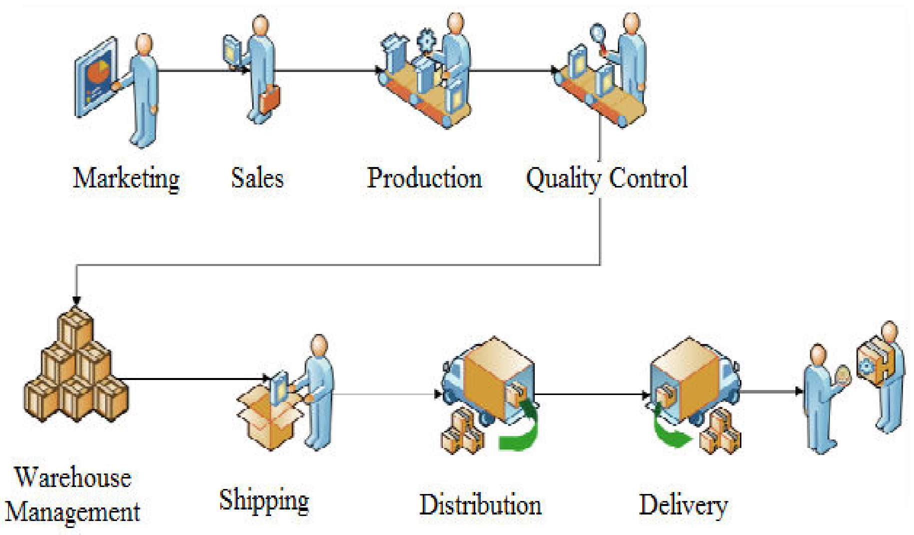


Figure 2.3 CIM example

A PIM is independent of any implementation technology and has a high level of abstraction level from the viewpoint of a software developer. In an MDA application, Unified Modeling Language (UML) is used as modeling language. UML is also a standard defined by OMG (www.omg.org). A PIM includes functionality and behaviour of the system meaning that it presents how the system will be implemented. Several PSMs can be generated from a PIM so that, a PIM must be extremely detailed to generate exact PSMs from it. At this abstraction level of MDA process, arhitects and designers play a critical role.

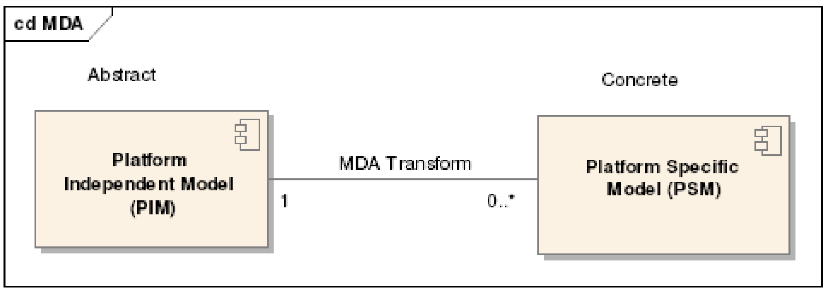

cd MDA
Abstract
Concrete
Platform Independent Model (PIM)
MDA Transform
1
0..*
Platform Specific Model (PSM)

Figure 2.4 Several PSMs from a single PIM

Figure 2.5 shows an example PIM with three UML classes, showing the abstract properties and relationships between an Author, Book and Publisher entities. It has no platform specific details (for example a specific programming language construct) and has a clear abstract presentation. It includes all the necessary information such as class properties, association rules between classes ... etc to create an exact PSM.

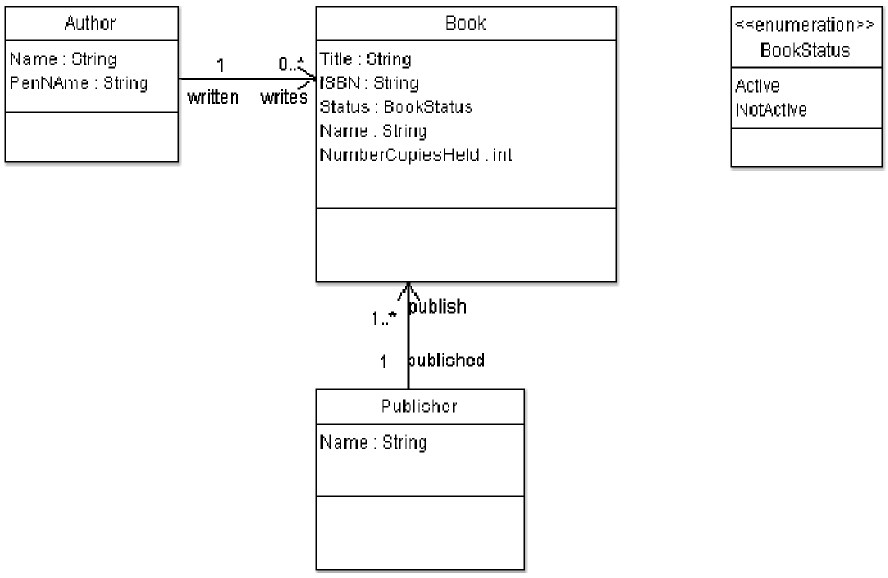


Figure 2.5 PIM example

Class diagram shown in Figure 2.5 is created with ArgoUML tool v0.28.1 (http://argouml.tigris.org).

A PSM combines the specifications in the PIM with the details of a particular type of platform. Because the final implementation code will be generated from the PSM, the PSM must include all the necessary information in it. Otherwise the generated code will require the software programmer to make final modifications on it.

There are 4 levels of possible automations to build a PSM from a PIM:

- Transformation is completely done by software developer.
- Software developer uses some established patterns for transformation.
- MDA-enabled tool generates some parts of PSM from the PIM, and then the other parts are manually transformed by software developer.
- MDA-enabled tool generates all parts of PSM from the PIM. This is the ideal solution that must exists in MDA approach.

The complete PIM is stored in Meta Object Facility (MOF) which is the input of the mapping step. After the mapping step applied to the PIM, the result PSM is produced. Figure 2.6 shows the generation process of a PSM from a PIM.

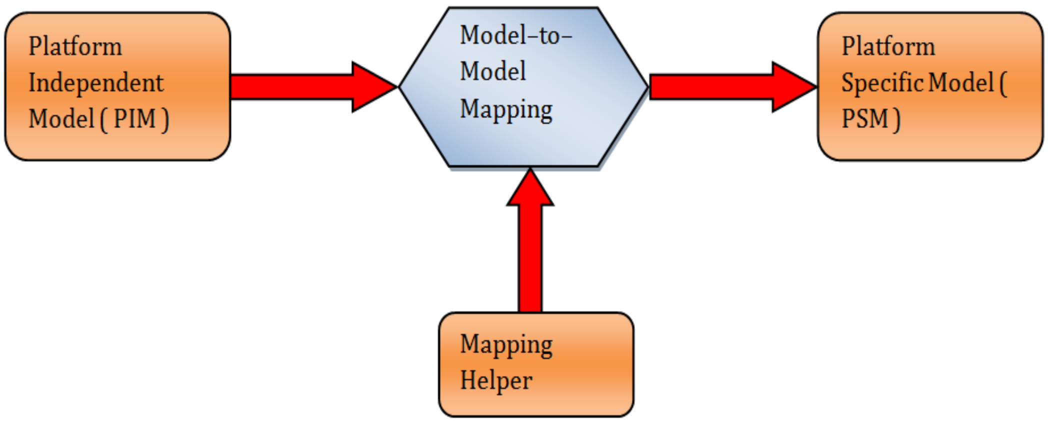
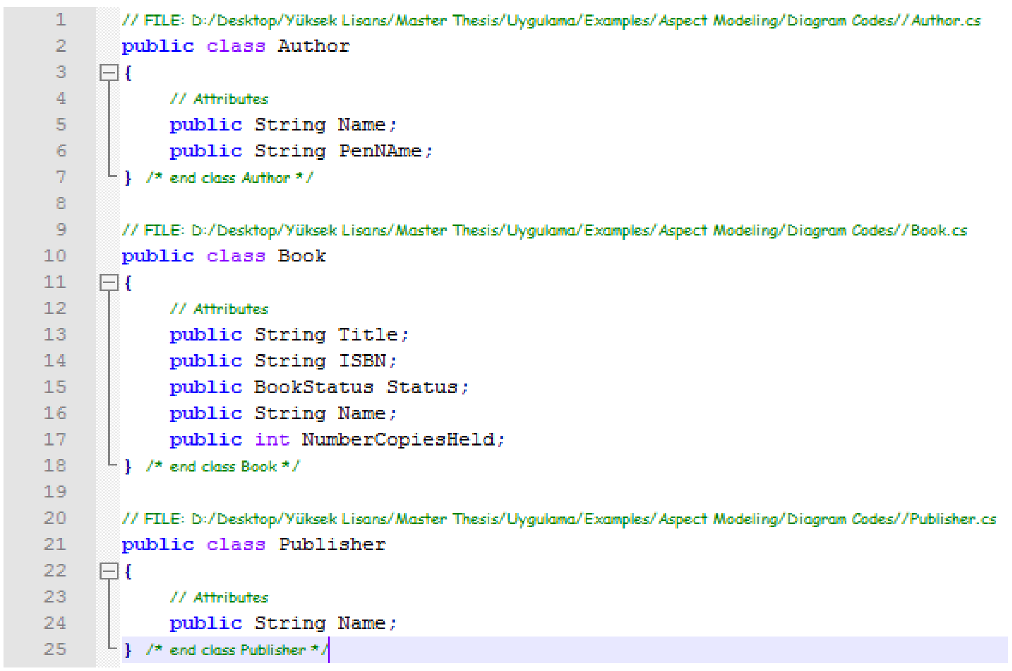

Platform Independent Model (PIM)
Model-to-Model Mapping
Platform Specific Model (PSM)
Mapping Helper

Figure 2.6 PIM to PSM generation

Code generation step from the PSM produces the implementation of the system for the particular platform. In this step, the PSM is taken as an input and then automation tool produces the program code. MDA development tools, available now from many vendors, can convert a PSM into a working implementation on virtually any middleware platform: Web Services, XML/SOAP, EJB, C#/.NET, OMG’s own CORBA, or others.

Figure 2.7 shows the generated program code of the PIM that is shown in Figure 2.5. C# programming language is used as the target platform. Code generation is also done with ArgoUML tool v0.28.1.

In this MDA process, we have not realized the step of generation PSM from the PIM. Because our MDA-enabled tool generated all parts of the PSM from the PIM and then generated implementation code automatically. We only selected the target platform (C# programming language in this example) before generating code from the PIM.



```
// FILE: D:/Desktop/Yüksek Lisans/Master Thesis/Uygulama/Examples/ Aspect Modeling/ Diagram Codes//Author.cs
public class Author
{
    // Attributes
    public String Name;
    public String PenNAme;
} /* end class Author */

// FILE: D:/Desktop/Yüksek Lisans/Master Thesis/Uygulama/Examples/ Aspect Modeling/ Diagram Codes//Book.cs
public class Book
{
    // Attributes
    public String Title;
    public String ISBN;
    public BookStatus Status;
    public String Name;
    public int NumberCopiesHeld;
} /* end class Book */

// FILE: D:/Desktop/Yüksek Lisans/Master Thesis/Uygulama/Examples/ Aspect Modeling/ Diagram Codes//Publisher.cs
public class Publisher
{
    // Attributes
    public String Name;
} /* end class Publisher */
```

Figure 2.7 Code generated from the PIM

Figure 2.8 (Sims, 2002) summarizes the overall MDA process.

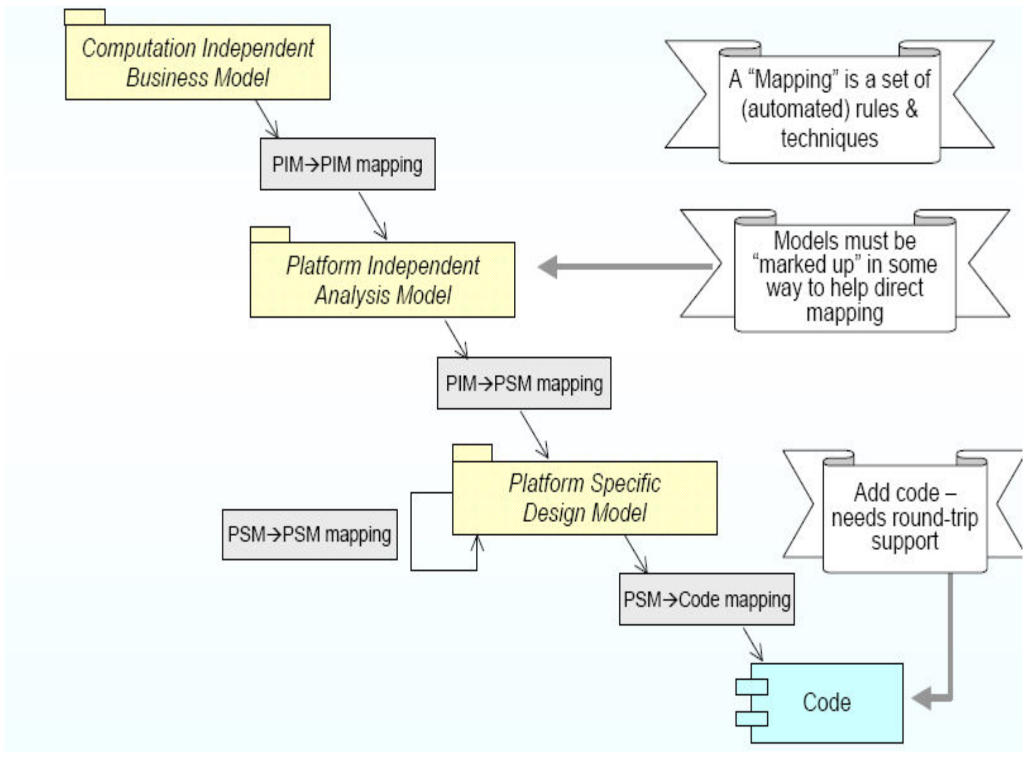

Computation Independent Business Model
PIM→PIM mapping
Platform Independent Analysis Model
PIM→PSM mapping
Platform Specific Design Model
PSM→PSM mapping
PSM→Code mapping
Code
A "Mapping" is a set of (automated) rules & techniques
Models must be "marked up" in some way to help direct mapping
Add code – needs round-trip support

Figure 2.8 Overall MDA process

2.5 Key Standards for MDA

The core standards of MDA are:

- Meta Object Facility (MOF)
- XML Metadata Interchange (XMI)
- Common Warehouse Metamodel (CWM)
- Unified Modeling Language (UML)

These standards are also defined by OMG as MDA framework. These standards define the core infrastructure of the MDA and greatly contribute to the current state of system modeling. These standards form the basis for publishing and managing models within MDA. These standards together with MDA contribute to developing large software systems to solve industry problems. With modeling tools that are built on top these standards, a large variety of applications from finance to e-commerce, from telecom to healthcare … etc can be built with more productivity and less costs. Figure 2.9 illustrates the idea.

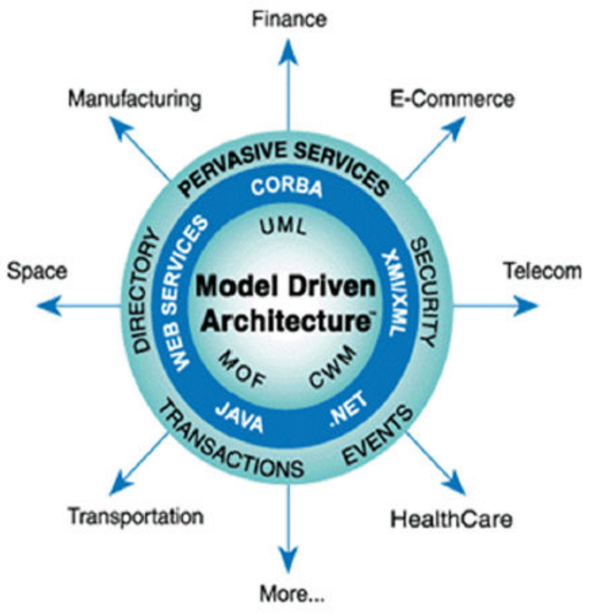


Figure 2.9 OMG's Model Driven Architecture model

2.5.1 Meta Object Facility (MOF)

The Meta Object Facility (MOF) is an OMG standard and defines a common, abstract language for defining new metamodels. MOF is a meta-metamodel or the model of a metamodel.

OMG ratified MOF in 1997 (Frankel, 2003). MOF has the basic premise that there will be more than one modeling languages to solve different problems in different domains, so that there must be a standard while defining new modeling languages.

We use different set of modeling constructs for different functions. The set of modeling constructs we use for relational data modeling includes table, column, key, …etc. The set modeling constructs we use for class modeling includes class, attribute, operation, association, …etc. MOF architects had understood the critical point that these different modeling constructs in different domains has some common behaviour. These common behaviours play the critical role on the evaluation of MOF.

MOF is the universal way of describing modeling constructs. MOF has been used to describe the modeling constructs in relational data modeling. MOF has been used to describe the modeling constructs in UML class modeling. And MOF can be used to describe new modeling constructs for new modeling languages.

In the MDA, models are integrated into the development process through the chain of model transformations from PIM to PSM, and then from PSM to application code. But to enable this chain of model transformations, MDA requires models to be expressed in MOF-based language. This guarantees that the models can be managed (storing, parsing, transforming of models) with no problem between different MDA-enabled tools. OMG's other standards as modeling languages; Unified Modeling Language (UML) and Common Warehouse Metamodel (CWM) are good examples of MOF-based languages. Code generation of an example UML class diagram that is built in an MDA-enabled tool can easily be done in another MDA-enabled tool with no problem.

The MOF Model (the MOF's core meta-metamodel) is object oriented. MOF borrows this object oriented class modeling constructs from UML and presents them as the common means for describing the syntax of modeling constructs. For example, definition of a table construct in relational data modeling language can be made as similar to a class construct in UML, definition of a column construct in relational data modeling language can be made as similar to an attribute construct in UML. Because the MOF Model is object oriented at the root level, other models along the MOF metamodeling hierarchy is also object oriented. This is also the nature of the inheritance property.

The current version of MOF is 1.4 that specification file can be downloaded from http://www.omg.org/technology/documents/formal/mof.htm. The details of the specification can be found here.

2.5.2 XML Metadata Interchange (XMI)

XML Metadata Interchange (XMI) is a framework for defining, interchanging, manipulating, integrating XML data and objects. XMI can also be used to automatically produce XML DTDs and XML Schemas from UML or other MOF metamodels. It means that XMI defines mapping from UML to XML, from other MOF-enabled modeling languages to XML.

XMI is an important OMG standard due to the prominence of XML in today's distributed system. After XML became popular, MOF architects began to study for representing metadata as XML documents. As a result of this study, in the late 1998, the OMG adopted a MOF-XML mapping which named as XML Metadata Interchange (XMI).

In 2001 the W3C (World Wide Web Consortium), owner of the XML specification, approved XML Schema as the successor to XML DTDs. After the approval of XML Schema as the successor of XML DTDs, OMG also approved the new XMI specification which defines a mapping from MOF to XML Schema. Table 2.1 shows the table of corresponding MOF and XMI versions:

Table 2.1 Corresponding MOF and XMI versions

MOF Version	**XMI Version**
MOF 1.3	XMI 1.1
MOF 1.4 (current)	XMI 1.2
MOF 1.4 (current)	XMI 1.3 (add XML Schema support)
MOF 1.4 (current)	XMI 2.0 (current, new format)
MOF 2.0 (in progress)	XMI 2.1 (in progress)

There is a misconception about the scope of the XMI standard. Because the UML is the most well-known MOF metamodel and XML Schemas or XML DTDs can be automatically generated from UML metamodel, it is believed that XMI is only for UML metamodels. This is not true for XMI. XMI can also act as a generator to produce an XML DTD or XML Schema from arbitrary MOF metamodel's abstract syntax.

Today, some MDA-enabled tools take XMI documents as input. These XMI documents may store some UML models or some other MOF-enabled modeling languages' models. Because XMI standard has XML DTDs or XML Schemas of the input document, the input XMI documents are validated against XML DTDs or XML Schemas first and then, these XMI documents can be converted to any MOF-enabled modeling languages' models. XMI as an input mechanism can also be defined for XMI as an output mechanism. Built model (UML model or any other MOF-enabled model) can be exported to XMI because XMI standard has the mapping MOF-XML.

Figure 2.10 illustrates the XMI process.

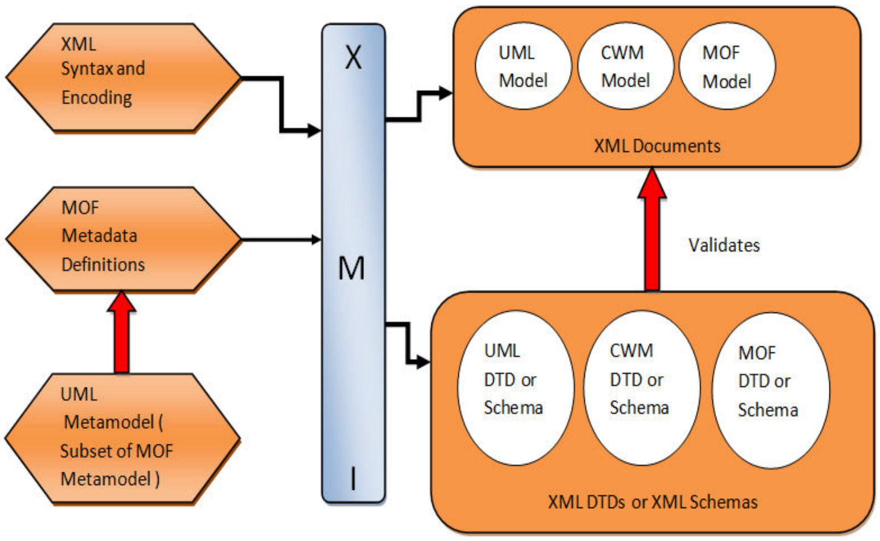
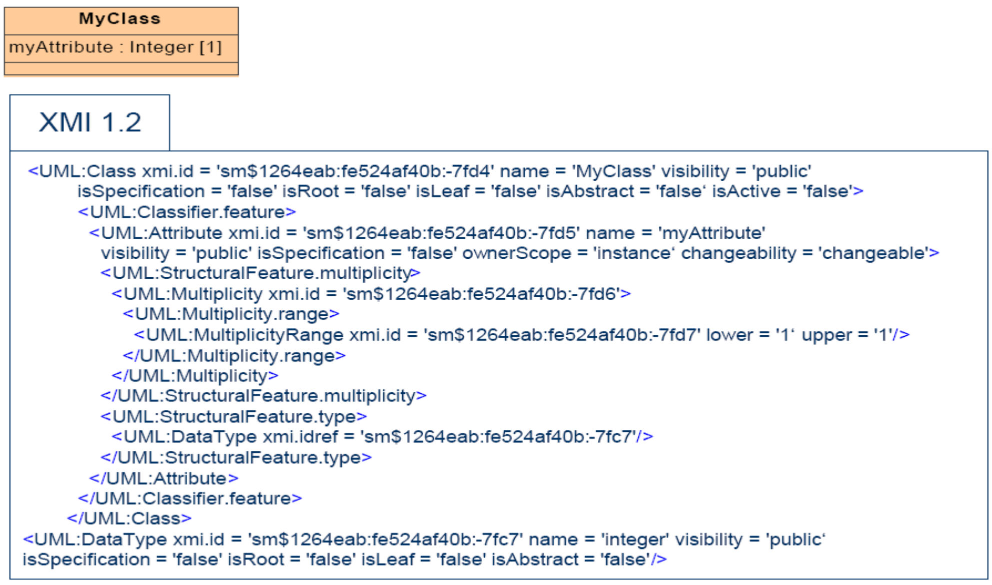


Figure 2.10 XMI Process

Figure 2.11 shows an example XML document in XMI 1.2 format. This XMI document is the result of a class named *"MyClass"* with has only one attribute named *"myAttribute"* type integer.



```
<UML:Class xmi.id = 'sm$1264eab:fe524af40b:-7fd4' name = 'MyClass' visibility = 'public'
        isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
        <UML:Classifier.feature>
          <UML:Attribute xmi.id = 'sm$1264eab:fe524af40b:-7fd5' name = 'myAttribute'
           visibility = 'public' isSpecification = 'false' ownerScope = 'instance' changeability = 'changeable'>
           <UML:StructuralFeature.multiplicity>
             <UML:Multiplicity xmi.id = 'sm$1264eab:fe524af40b:-7fd6'>
               <UML:Multiplicity.range>
                 <UML:MultiplicityRange xmi.id = 'sm$1264eab:fe524af40b:-7fd7' lower = '1' upper = '1'/>
               </UML:Multiplicity.range>
             </UML:Multiplicity>
           </UML:StructuralFeature.multiplicity>
           <UML:StructuralFeature.type>
             <UML:DataType xmi.idref = 'sm$1264eab:fe524af40b:-7fc7'/>
           </UML:StructuralFeature.type>
          </UML:Attribute>
        </UML:Classifier.feature>
      </UML:Class>
<UML:DataType xmi.id = 'sm$1264eab:fe524af40b:-7fc7' name = 'integer' visibility = 'public'
isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'/>
```

Figure 2.11 An example XMI document

2.5.3 Common Warehouse Metamodel (CWM)

The Common Warehouse Metamodel (CWM) standardizes a complete, comprehensive metamodel that enables data modeling, data warehousing, data transformation, data analysis. Specifications of this modeling language have been defined by OMG. This standard is a product of cooperative effort between OMG and the Meta-Data Coalition (MDC). CWM is also a MOF-based modeling language like UML. The difference between the UML and CWM is; UML is used for application modeling and CWM is used for data modeling. Meaning that CWM is used for modeling relational data. It is a good example of applying the MDA paradigm to an application area.

Another similar definition of CWM can be given as the following: "CWM is a standard set of interfaces that can be used to enable easy interchange of warehouse and business intelligence metadata between warehouse tools, warehouse platforms and warehouse metadata repositories in distributed heterogeneous environments" (Gally, 2007).

CWM uses XMI as its interchange mechanism. By this way, CWM benefits the full power and flexibility of XMI to interchange both warehouse metadata and CWM metamodel itself. To interchange warehouse metadata or CWM metamodel itself, the CWM uses the original specifications of XMI meaning that it does not require any other extensions to XMI.

XMI can act as a generator to produce an XML DTD or XML Schema from arbitrary MOF metamodel's abstract syntax. By this way, a standard XML DTD or XML Schema is generated for CWM metamodel using XMI's DTD or Schema production rules. After that, the entire warehouse metadata can be represented as an XML document using XMI's Document Production Rules.

CWM can be used by six categories of users:

- Warehouse platform and tool vendors
- Professional service providers
- Warehouse developers
- Warehouse administrators
- End users
- Information technology managers

Current version of CWM is 1.1 that specification file can be downloaded from http://www.omg.org/technology/documents/formal/cwm.htm. The details of the specifications can be found here.

Figure 2.12 shows the fragment of relational data metamodel of the Common Warehouse Metamodel. As seen from Figure 2.12, CWM metamodel is like UML metamodel. The *table* modeling construct used in CWM is like *class* modeling construct used in UML. This is because the MOF borrows object oriented class modeling constructs from UML and presents them as the common means for describing the syntax of modeling constructs. Because the MOF model is object oriented at the root level, CWM model that inherits from MOF is also object oriented.

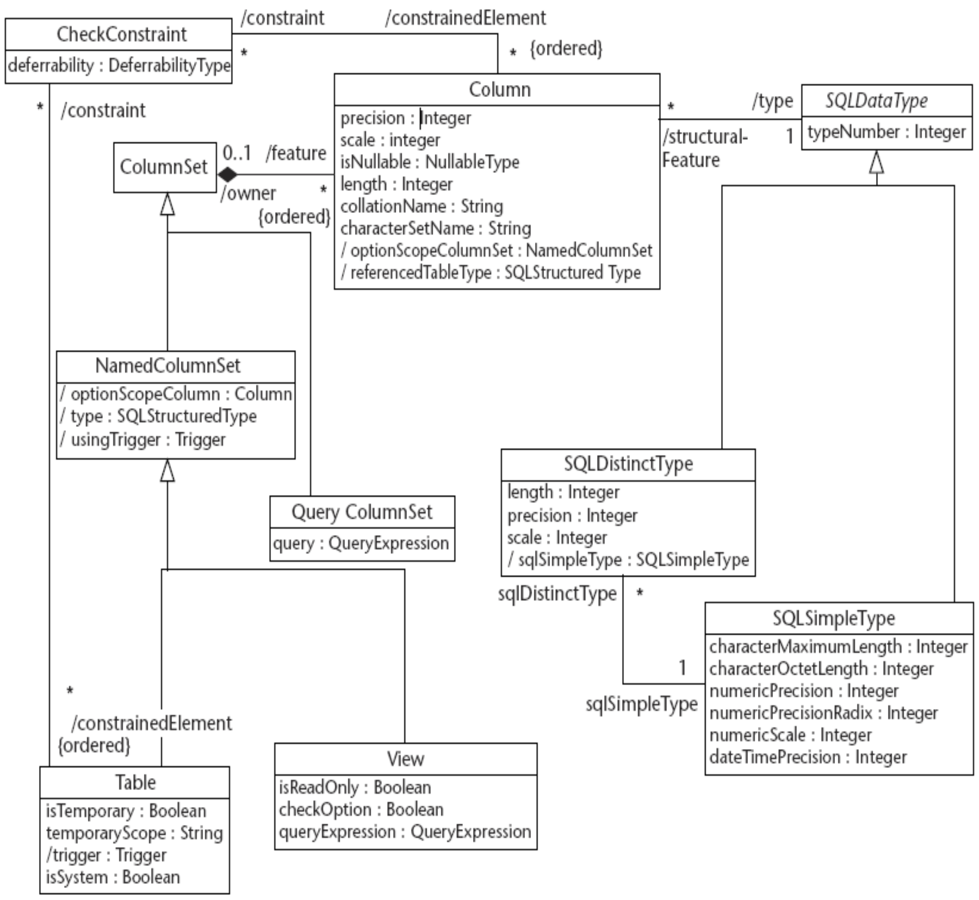


Figure 2.12 A fragment of the CWM relational data metamodel

2.5.4 Unified Modeling Language (UML)

The Unified Modeling Language (UML) is OMG's most-used specification and the way the software developers model application structure, behaviour, architecture, business process and data structure.

In the MDA approach there are four major steps: generation of a Computation Independent Model (CIM), building a Platform Independent Model (PIM), transforming PIM into a Platform Specific Model (PSM) and then generating the code out of the PSM. Two of these four steps, PIM and PSM are defined in UML in many specifications making OMG's standard modeling language a key foundation of the MDA. Note that usage of UML in PIM and PSM is not a requirement; usage of a modeling language that is MOF-enabled is the key solution for the applications that is MDA-enabled.

As object-oriented analysis and design techniques spread during early 1990s, the OOAD industry divided into three camps, corresponding to the followers of Grady Booch, Ivar Jacobson, and Jim Rumbaugh. These three developers had their own notation, methodological approaches and tools. In the late 1990s, The Rational Software Corporation brought Ivar Jacobson and Jim Rumbaugh into the company to join Grady Booch, three developers wrote the first informal UML specification. Then they sent this first informal specification to OMG for standardization.

Before the first Unified Modeling Language (UML) standards were published, visual software modeling tools had different notations created by different gurus. This led to incompatibility of models between different modeling tools in the industry. The absence of a standardized notation deterred potential users and as an inevitable result the modeling tool market was tiny and fragmented. Beyond the standardization issues, many of the modeling tools only allowed sketching of diagrams just a picture. These modeling tools lacked the ability to derive meanings from sketched diagrams and they could not check overall consistency between model elements. Sketched models were rarely integrated into the software development lifecycle.

The UML standard has changed the way of modeling and triggered the dramatic growth in visual modeling that has led to its worldwide use not only is software design, but also in non-software disciplines such as systems engineering and in the business domain.

As tool vendors in OMG community started to implement UML standard specifications in their modeling tools, continuous feedback from these tool vendors and user communities that use modeling tools received. These feedbacks helped the UML standard to evolve and mature. The original UML 1 standard of 1997 was backed by 21 OMG member companies and feedbacks from these companies helped OMG to refine UML specification. After that UML 2 in 2005 was published by OMG. This revision to UML standard contained all the resolved issues reported by tool vendors and user communities. Beside that UML 2 standard had also some new improvements in its underlying structure. UML 2 had the infrastructure specified using OMG's Meta Object Facility (MOF) framework. This means that UML 2 is more than just a pretty picture. Because UML is an MOF-based modeling language since UML 2 standard specification, it has all the benefits of a MOF-based modeling language. A MOF-based modeling language can capture the meaning of model elements and can capture relationships between model elements. A MOF-based modeling language can automatically generate application code from models. A MOF-based modeling language can automatically generate application documents from models … etc. Shortly a MOF-based modeling language can be included into the MDA process.

These developments in the world of visual modeling have led to establishment of OMG's Model Driven Architecture (MDA).

This section briefly have discussed UML standard with its importance in visual modeling industry and UML standard evolution. In the following sections more detailed information about the usage of UML standard in the MDA process will be given.

Figure 2.13 shows a fragment of UML metamodel for class modeling.

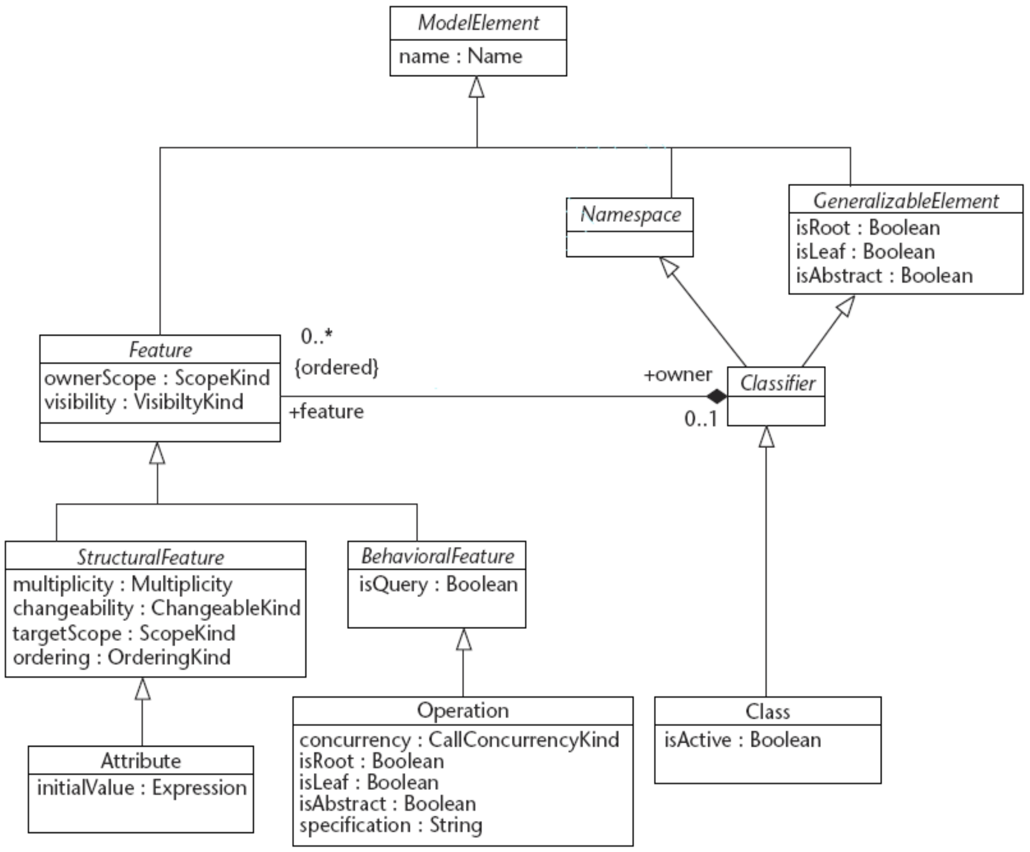


Figure 2.13 A fragment of UML metamodel for class modeling

2.6 MOF Metalevels

MOF architecture defines four “metalevels” named M3, M2, M1, and M0. The definitions of these metalevels are important in order to be MDA literate.

Level M3 is the MOF itself whose elements are the constructs to define metamodels. This is the root and most abstract metalevel. MOF elements include Class, Attribute, Association, and so on. These MOF constructs are used to define new modeling languages. For a modeling language to be MOF-based, this modeling language’s modeling constructs must be instances of MOF modeling constructs. M3 level is meta-metamodel and M3 level is self describing. This level is the end of the line.

Level M2 has the metamodels that are defined via MOF constructs. There are a number of good examples of MOF-based metamodels such as Unified Modeling Language (UML), Common Warehouse Metamodel (CWM) … etc. These metamodels' constructs are defined using the MOF Class, MOF Attribute, MOF Association, and so on. Figure 2.14 shows an example metamodel with some modeling constructs that are instances of MOF constructs.

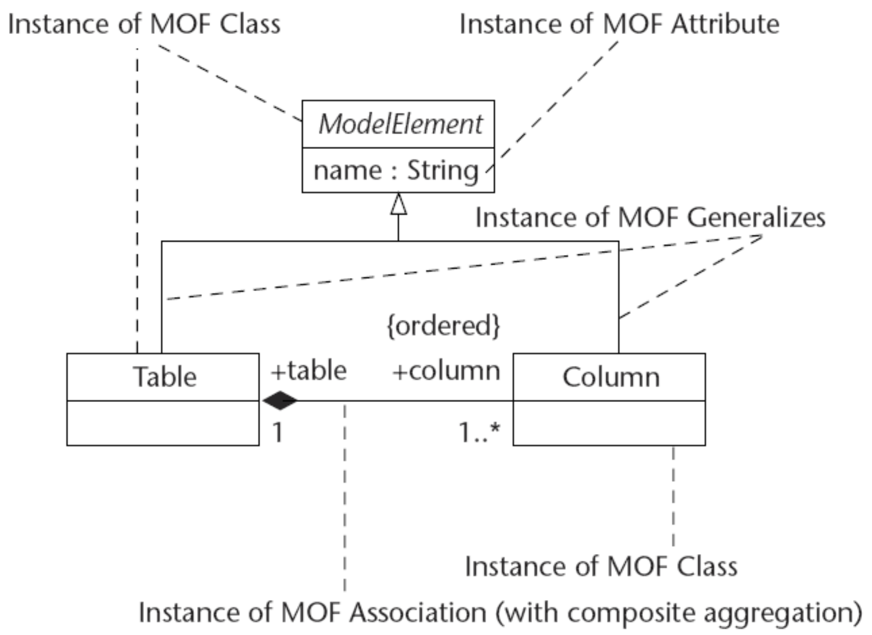

Instance of MOF Class
Instance of MOF Attribute
ModelElement
name : String
Instance of MOF Generalizes
{ordered}
Table
+table
+column
Column
1
1..*
Instance of MOF Class
Instance of MOF Association (with composite aggregation)

Figure 2.14 Example M2 level metamodel constructs

Level M1 has the model elements that are instances of M2 level metamodel constructs. A UML class diagram can be a good example for the M1 level model. Figure 2.14 shows an example metamodel that has Table, Column …etc constructs. As an example M1 level model that is instance of the metamodel shown in Figure 2.14, we can figure the following:

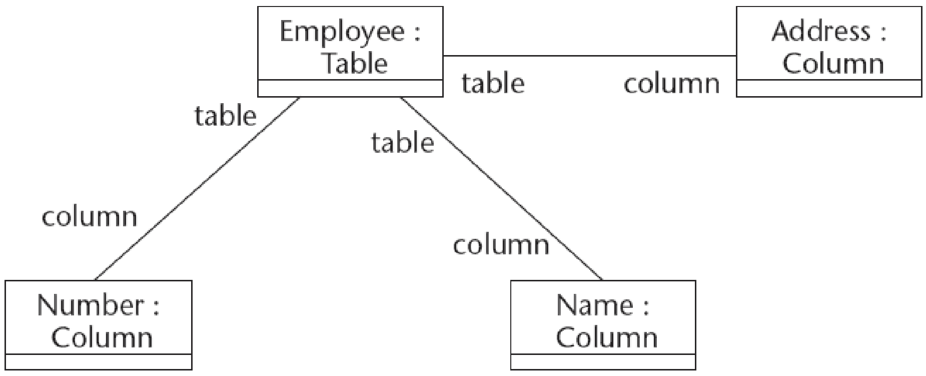


Figure 2.15 Example M1 level model

Employee is an instance of table construct. *Employee* table has some instances of column constructs that are named as *Address*, *Number*, and *Name.*

Level M0 has objects and data that are instances of M1 level model elements. This is the leaf level of the metalevel hierarchy. Elements at this level represent real life entities. As an example M0 level model that have instances of the model elements shown in Figure 2.15, we can consider an employee with Name: “Ahmet Ersin”, Number: 123456 and Address: “Esentepe Mah. Gül Sok. Yağmur Apt. No: 31 Kat: 2 Daire: 3”.

To put it briefly, M2 level elements are instances of M3 level elements. M1 level elements are instances of M2 level elements. M0 level elements are instances of M1 level elements. M3 level is meta-metamodel level and is the end of line. M3 level is the self describing level.

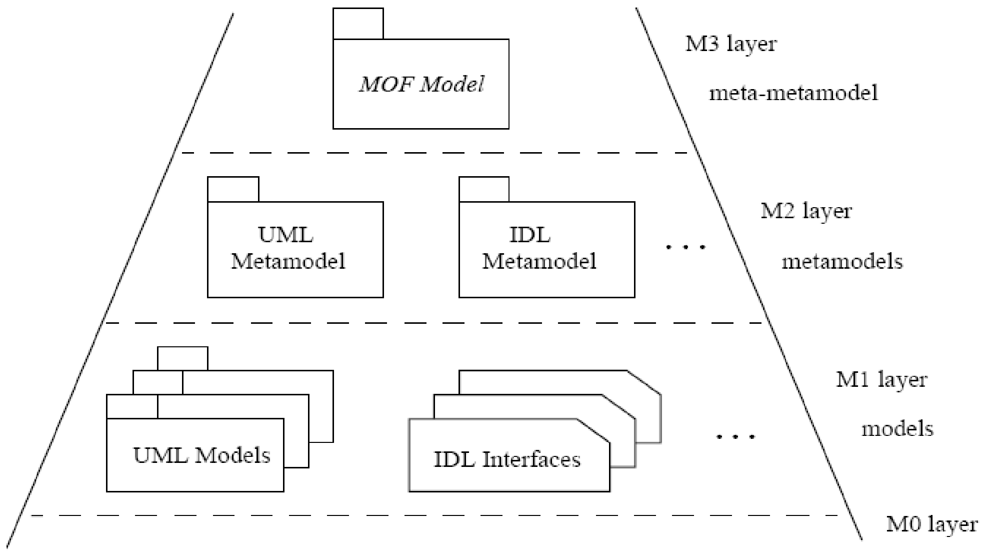


Figure 2.16 MOF Metalevels

2.7 More UML

As said before, the UML standard has changed the way of modeling and triggered the dramatic growth in visual modeling that has led to its worldwide use not only in software design, but also in non-software disciplines such as systems engineering and in the business domain.

This section provides details about the most important UML diagrams used in the visual modeling of computing programs.

UML 2.0 defines thirteen types of diagrams, divided into three categories: six diagram types represent static application structure; three types represent general types of behaviour; and four types represent different aspects of interactions.

Structure Diagrams include the Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram and Deployment Diagram.

Behaviour Diagrams include the Use Case Diagram, Activity Diagram, and State Machine Diagram.

Interaction Diagrams derive from the Behaviour Diagram. Interaction Diagrams include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

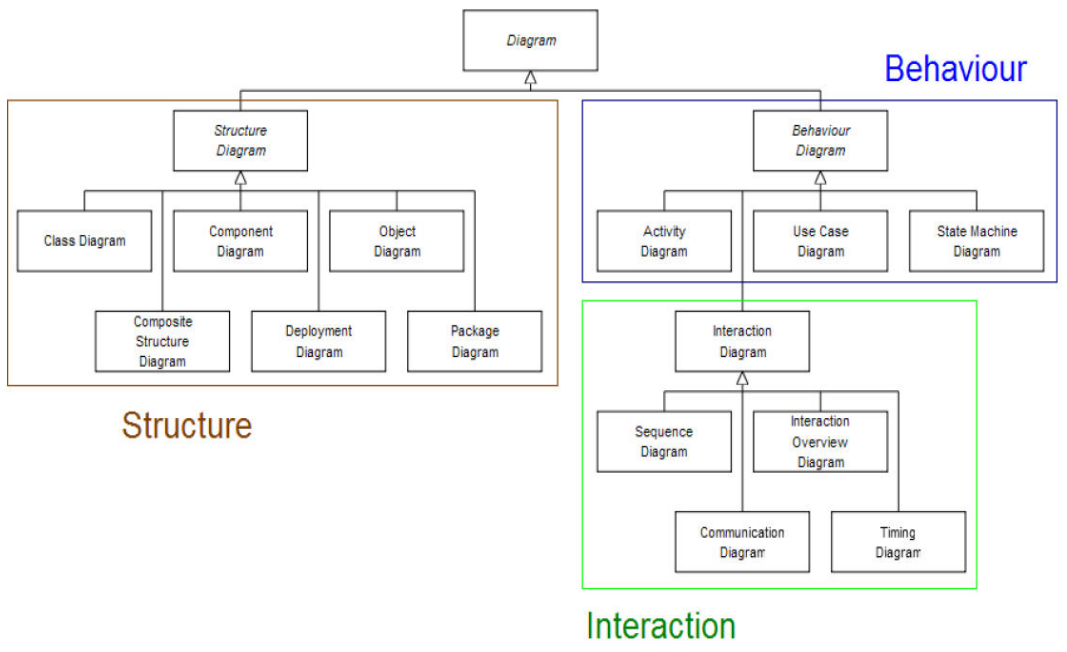


Figure 2.17 UML 2.0 Diagrams

The most useful, standard UML diagrams (Bell, 2003); use case diagram, class diagram, sequence diagram, state machine diagram, activity diagram, component diagram, and deployment diagram will be explained in this section.

Class Diagram is in the Structure Diagrams category because it does not describe the time-dependent behaviour of the system. The main elements of a class diagram are class, association, generalization, realization, dependency, aggregation, composition … etc.

Classes are drawn as rectangles. List of attributes and operations are shown in separate compartments. Relationships among classes are drawn as paths connecting class rectangles. The different kinds of relationships are distinguished by different kinds of line styles.

Figure 2.18 shows an example class diagram from the box office application. This class diagrams states the structure of the system as the following. Customers may have many reservations, but each reservation is made by only one customer. There are two kinds of reservations: subscription series and individual reservations. Subscription series can reserve many tickets and individual reservations can only reserve one ticket. A ticket can only be reserved at most by one reservation. Every performance has many tickets available and each performance can be identified by a show, date and time attributes. A show can be shown by one or more performance.

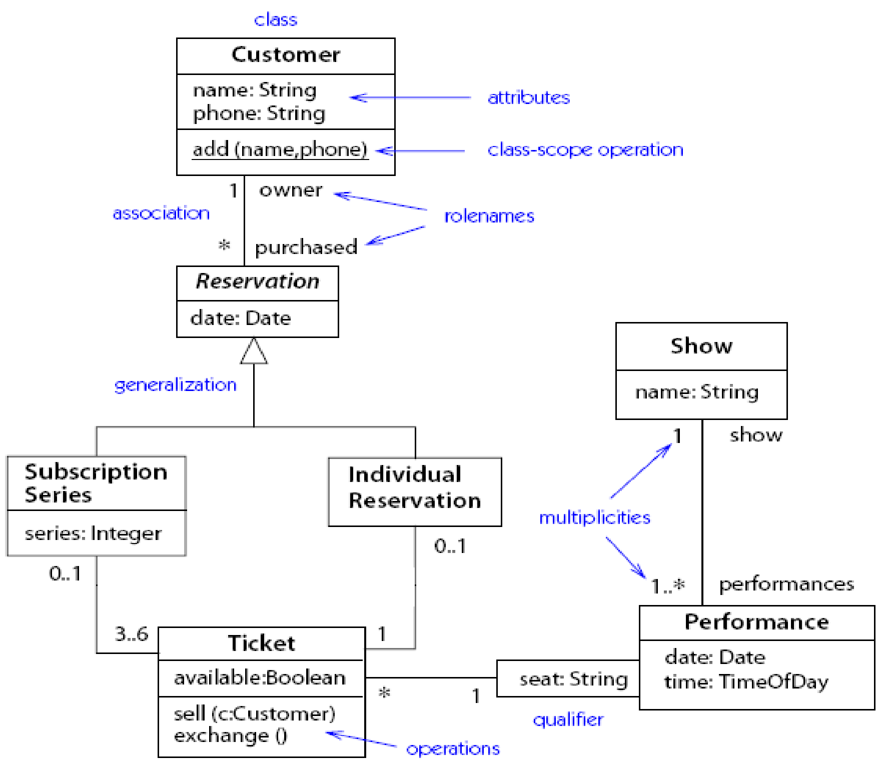


Figure 2.18 Class diagram of a box office application

An **association** describes connections among objects in a system. The most common kind of association is binary association between a pair of classes. Association symbol is line and drawn from one class to another. Associations carry information that shows relationships among objects in a system. Without associations, the designed system has only isolated classes that do not work together.

Each connection of an association to a class is called an association end. Association ends can have names and the most important property of association ends are multiplicity. Multiplicity shows that how many instances of a class can be related to one instance of the other class.

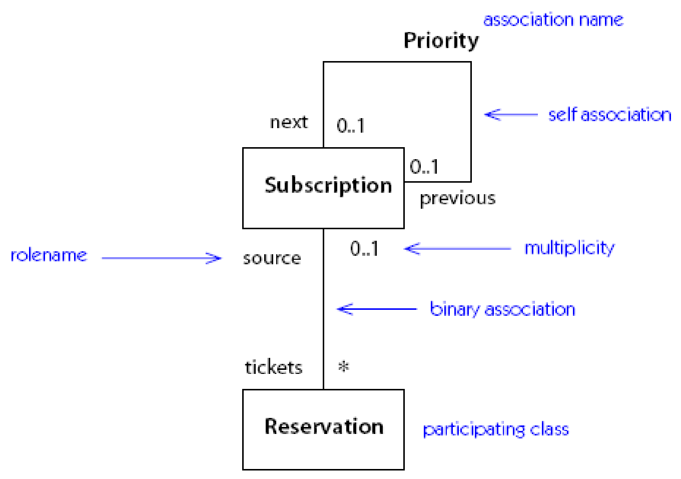


Figure 2.19 Association notation

An **aggregation** is also an association with a special meaning that depicts part – whole relationship. It is shown by a hollow-diamond adornment on the end of the path to the aggregate class.

A **composition** is a stronger form of association in which the composite object has the responsibility for managing its parts, such as their allocation and deallocation. It is shown by a filled-diamond adornment on the end of the path to the composite class.

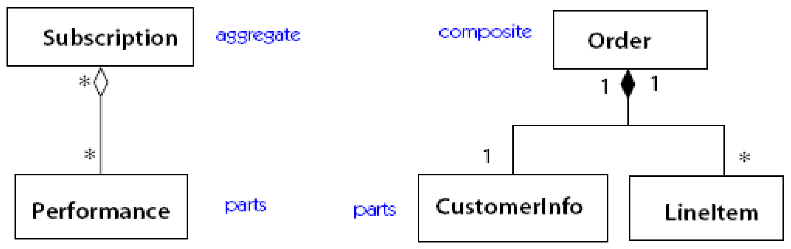


Figure 2.20 Aggregation and composition notations

A **generalization** is a relationship between a more general description and a more specific description that builds on it and extends it. The more specific description inherits all the properties, members, relationships of the more general description and may contain additional information. The more general description is called as parent and the more specific description is called as child.

A generalization is drawn as an arrow from child to the parent, with a large hollow triangle on the end connected to the parent.

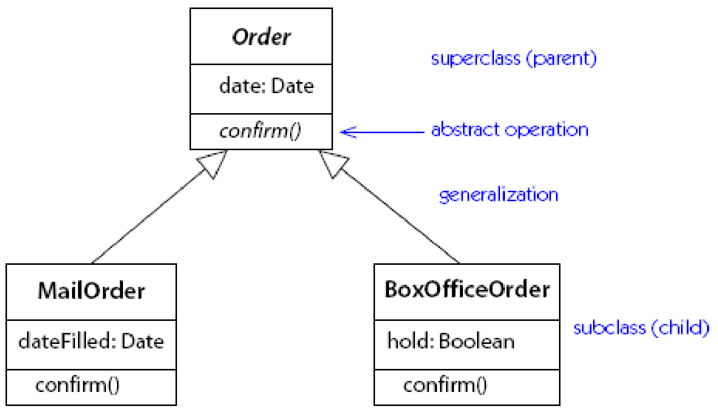


Figure 2.21 Generalization notation

A **realization** is a relationship that connects a model element, such as a class, to another model element, such as an interface. Interface does not include implementation, shows only behavioral specification. Class that realizes the interface must support all the operations that the supplier has.

Realization is displayed as a dashed arrow with a closed hollow arrowhead. It is similar to generalization notation except the dashed arrow line style.

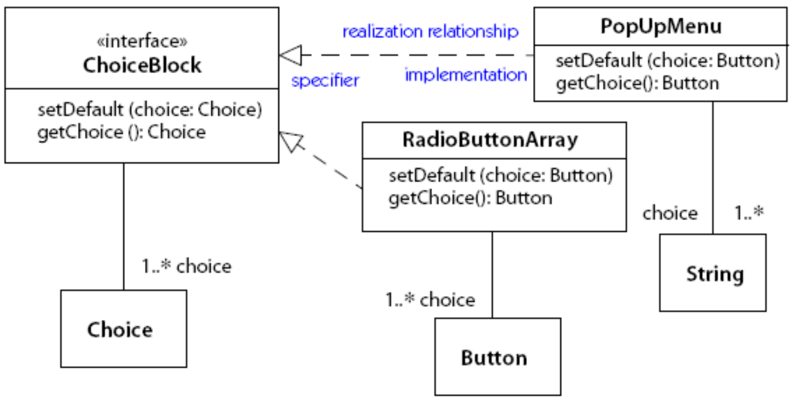


Figure 2.22 Realization notation

A **dependency** is also a relationship between two or more model elements. It indicates a situation which a change to the supplier element may require a change to the client element in dependency. A dependency is drawn as a dashed arrow from the client to the supplier.

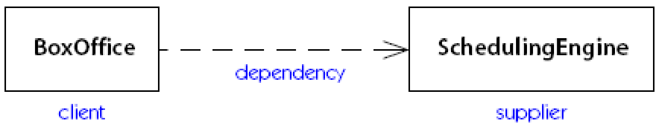


Figure 2.23 Dependency notation

Component Diagram is in the Structure Diagrams category. Component Diagram shows the physical packaging of the reusable pieces of the system into substitutable units called as components. Components are the high level reusable pieces. A component diagram shows dependencies among components and each component realizes some interfaces and use others.

A components is a physical unit of implementation with well defined interfaces that is intended to be used as a replaceable part of a system. Each component includes the implementation of certain classes from the system design. Beside included implementation, components may need some extra implementation that it does not include. In these situations, components depend on the other components that support the proper interfaces for the required implementation.

A component is drawn as a rectangle with two small rectangles on its side. It may be attached to another components' interfaces by solid lines.

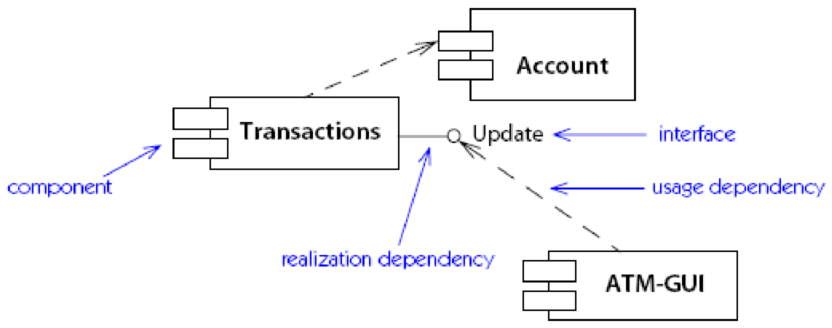


Figure 2.24 Component Diagram

Deployment Diagram is in the Structure Diagrams category. Deployment diagram shows the physical arrangement of runtime computational resources, such as computers and their interconnections. Each computation resource in deployment diagram is called as a node. At runtime, nodes can contain components and objects.

A node is shown as a three dimensional cube with the name of the node at the top of the cube. Association between nodes represent communication paths. The associations can have stereotypes to make difference of different kinds of paths.

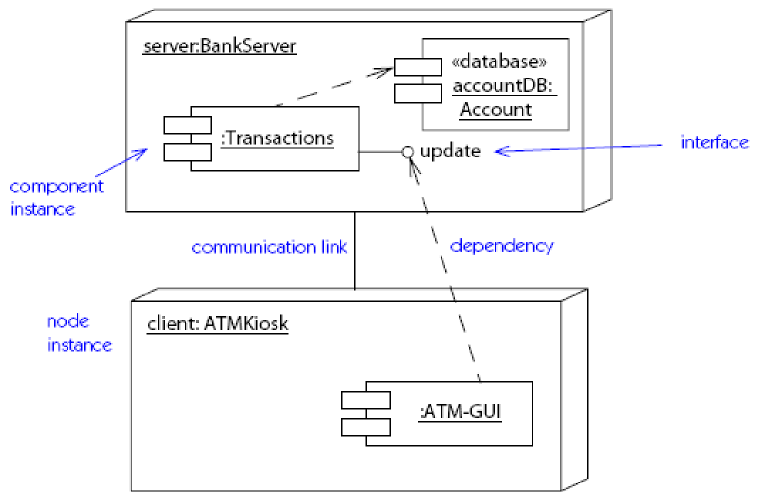


Figure 2.25 Deployment diagram

Use Case Diagram is in the Behaviour Diagrams category because it illustrates a unit of functionality provided by the system as percieved by outside users, called actors. A use case is a functionality of a system that is expressed as a list of relations among actors and the sytem. The main purpose of the use case diagram is to help development teams to visualize the functional requirements of a system, including the relationship of actors to essential processes.

To show a use case on the use case diagram, an oval (with the name written) is put in the middle of the diagram. To draw an actor on a use case diagram, a stick person is put to the left or right of the use case. Relationships between use cases and actors is represented as lines drawn from an actor to the use case.

In the example Telephone Catalog use case diagram shown in Figure 2.26, there are four actors; customer, salesperson, shipping clerk and supervisor. There are four use cases; check status, place order, fill orders, establish credit. A customer can use check status, place order and establish credit processes. A salesperson can use check status, place order processes. A shipping clerk can use fill orders process and a supervisor can use establish credit process.

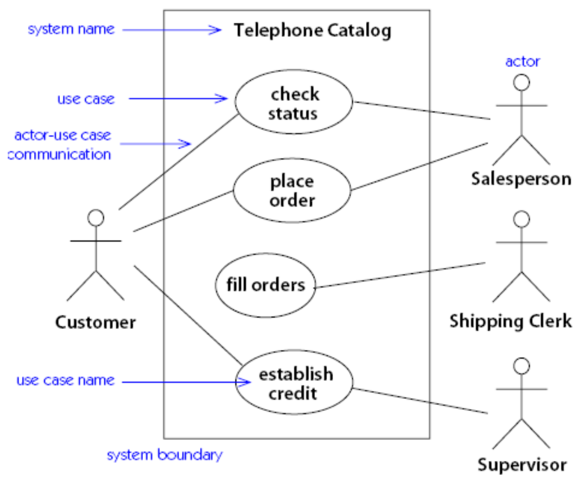


Figure 2.26 Use case diagram

State Machine Diagram is in the Behaviour Diagrams category. State Machine diagrams model the possible states of an object of a class. States in the diagram are connected by transitions. Each state models a period of time during the life of an object during which it satisfies certain conditions. An event signalled may cause the firing of a transition that takes the object from one state to the new state. The notation of the state machine diagrams has five basic elements: the initial starting point which is drawn as a solid circle, a transition between states which is represented as a line with an open arrowhead, a state which is represented as a rectangle with rounded corners, a decision point which is represented as an open circle, one or more termination points which are represented as a circle with a solid circle inside it. Figure 2.27 shows an example state machine for ticket selling machine.

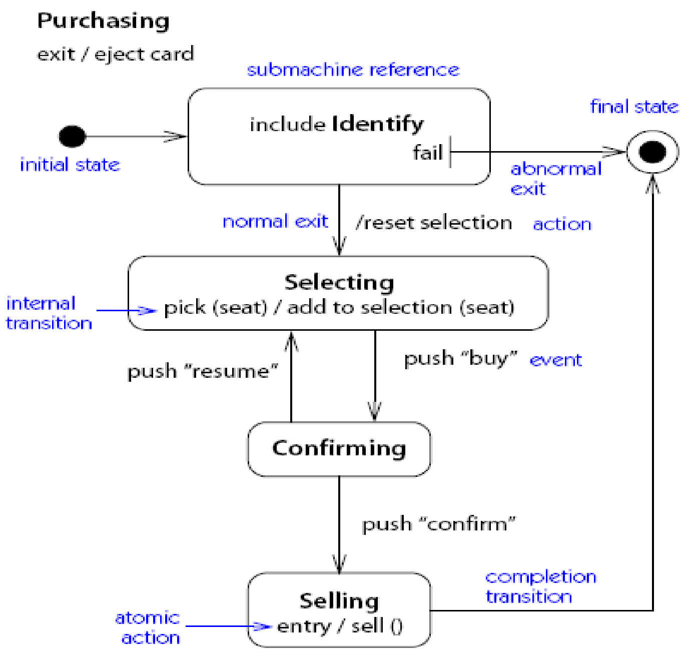


Figure 2.27 Example state machine diagram

Activity Diagram is in the Behaviour Diagrams category. Activity diagrams show the procedural flow of control between one or more objects while processing an activity. Activity diagrams can be used both to model high level business processes and to model low levels internal class actions. Because activity diagrams are less technical in appearence, compared to the sequence diagrams, to use activity diagrams for modeling high level business processes may be a better experience.

An activity diagram's notation is similar to a state machine diagram's notation. Like a state machine diagram, an activity diagram also starts with a solid circle that represents the initial activity. An activity, as a state in the state machine diagrams, is represented as a rectangle with rounded corners with the activity's name enclosed. Activities can be connected to other activities through transition lines or activities can be connected to decision points that connect to different activities guarded by the conditions of the decision point. Activities end with termination point that is represented as a circle with a solid circle inside it (just as in state machine diagram). Activities can also be grouped into swimlanes to indicate the object that actually performs the activity. Figure 2.28 shows an example activity diagram.

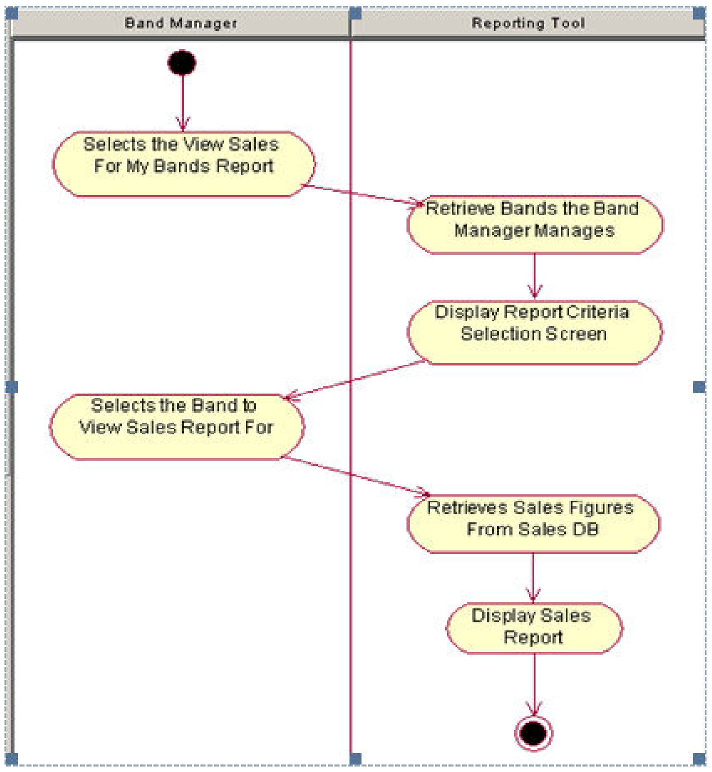

Band Manager
Reporting Tool
Selects the View Sales For My Bands Report
Retrieve Bands the Band Manager Manages
Display Report Criteria Selection Screen
Selects the Band to View Sales Report For
Retrieves Sales Figures From Sales DB
Display Sales Report

Figure 2.28 Example activity diagram example

Sequence Diagram is in the Interaction Diagrams category that derieve from the Behaviour Diagrams. A sequence diagram shows a detailed flow for a specific use case. A sequence diagram can show a scenario that is an individual history of a transaction. They show the calls between the different objects in the call sequence.

A sequence diagram has two dimensions. The vertical dimension shows the sequence of messages in the time order that they occur. A message is represented as an arrow from the lifeline of an object to the lifeline of another object. The horizontal dimension shows the object instances to which the messages are sent.

Reading a sequence diagram is very simple. Start at the top left corner with the driver class instance that starts the sequence. Then follow each message down the diagram.

Figure 2.29 shows an example sequence diagram that illustrates a ticket buying scenario. The first message is sent by the *kiosk* driver class that requests the ticket. After *request* message processed by *box office* class, *show availability (seat-list)* message is sent back to *kiosk* class to preview available seat list. After messages are sent between lifeline, the scenario ends with the *eject card* message sent from *box office* class to the *kiosk* class.

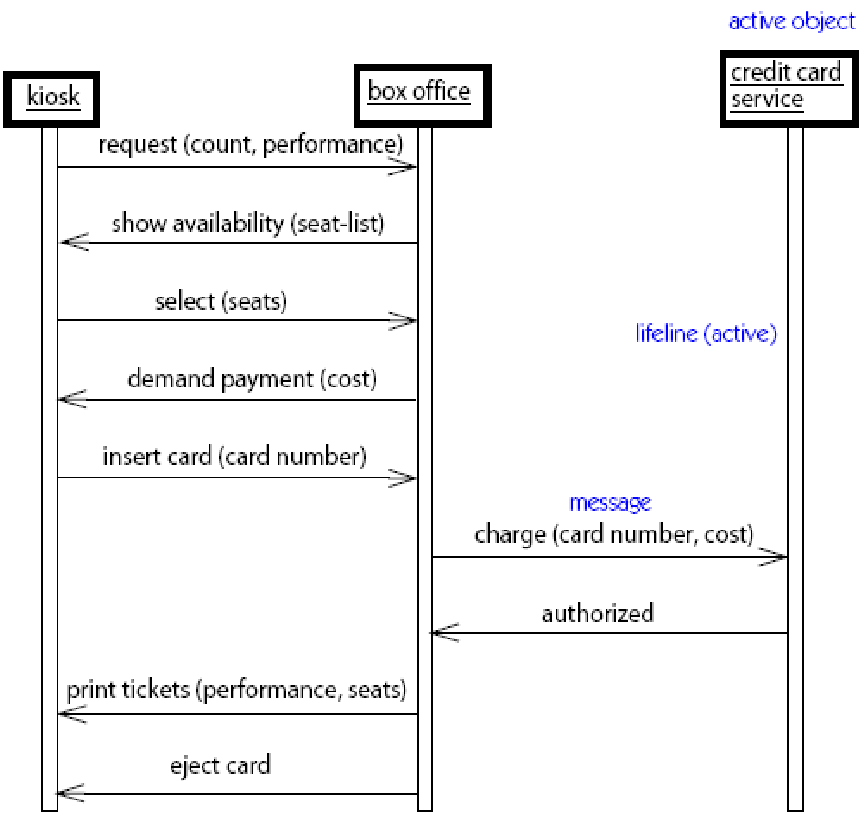

active object
kiosk
box office
credit card service
request (count, performance)
show availability (seat-list)
select (seats)
lifeline (active)
demand payment (cost)
insert card (card number)
message
charge (card number, cost)
authorized
print tickets (performance, seats)
eject card

Figure 2.29 Example sequence diagram

CHAPTER THREE

ASPECT-ORIENTED PROGRAMMING (AOP)

3.1 AOP Definition

Computer science has experienced an evolution in programming languages starting with machine language and then using more and more abstract programming languages as the software industry evolves. Each of these steps in programming language technology has advanced the ability to achieve clear seperation of concerns at the implementation level.

Today's dominant programming paradigm is Object Oriented Programming paradigm which has the idea of building a software system by decomposing a problem into objects and then writing code of those objects. Objects with completed implementations interact together to make a complete solution to the problem. The mechanisms that Object Oriented Programming (OOP) supports can easily be used to map a real domain problem into the software domain. For example, to represent a book in software platform in Object Oriented Programming paradigm, you create a *Book* class with attributes *name*, *isbn*, *author*, *publisher* ...etc and then create an instance of the *Book* class to make operations on the *Book* object. It is so easy with OOP.

Writing complex applications such as graphical user interfaces, operation systems, distibuted applications while maintaining is possible with Object Oriented Programming.

However there have been found some programming problems that OOP technique may not be enough to clearly capture. Clearly capturing means that implementation of design decisions is not scattered throughout the code, not resulting in tangled code that is excessively difficult to develop and maintain. These kind of concerns that result the tangled code in the implementation are named as cross-cutting concerns. Such cross-cutting concerns can range from high level notions like security and quality of service to low-level notions such as caching and buffering.

Aspect Oriented Programming technique provides some mechanisms to solve cross-cutting concerns in a more effective and modular way. Some of these mechanisms are such that: join point, pointcut, advice, aspect, aspect weaver, inter-tpe declaration, context exposing ...etc. These mechanisms make it possible to better program cross-cutting concerns by seperately specifying these concerns and then weave or compose them together into a coherent implementation.

All programming languages since Fortran have had a way for the seperation of concerns by creating and calling subprograms. Subprograms in these programming languages are a good way to implement crosscutting concers. In the OOP technique; usage of inheritance, polymorphism, helper classes ...etc are also good ways. However, often cross-cutting concerns can not clearly be implemented by using these ways. Implementations of these concerns become tangled into other elements. To overhelm these kind of cross-cutting issues, AOP provides aspects: mechanisms beyond subprograms and inheritance for localizing the expression of a cross-cutting concern.

Seperating the expressions of multiple concerns in programming systems with AOP guarantees simpler system evolution, more comprehensible systems, adaptability, customizability, and easier reuse. By aggregating cross-cutting concerns into aspects, there will be no tangled code and this will result as making the aspect code and the base code easier to understand.

Aspect code and base code is woven to a single implementation by aspect weavers before the execution of software systems. Aspects are a seperate layer that are built on top of the current implementation, so that base code is not aware of the aspects. This leads to simpler base code to develop and maintain.

Effectively achieving seperation of concerns in programming leads to high quality products. In today's increased software complexity, there are specialized algorithms for distribution, authentication, access control, synchronization, encryption, redundancy, logging and so forth which are also possibly to be cross-cutting concerns. Base code developers should not have any knowledge about these algorithms. To provide a way to easily include these cross-cutting concerns' implementations into the software products makes base code developers focus on the real problem domain except cross-cutting concerns. This increases the productivity of the developers. Base code developers do not lose any time to have knowledge outside the expertise of the real problem domain. Every developer tries to do their best in their problem domains (authentication algorithm developers only focus on authentication, encryption algorithm developers only focus on encryption ...etc.)

Aspect Oriented Programming is growing rapidly and it is used in many areas, such as middle-ware, security, fault tolerance, quality of service, and operating systems ...etc. AOP is not yet a fully mature discipline and needs to be used in more applications to be improved.

3.2 AOP related terms

Aspect Oriented Programming technique provides some mechanisms to solve cross-cutting concerns in a more effective and modular way. Some of these mechanisms are such that: join point, pointcut, advice, aspect, aspect weaver ... etc. These mechanisms make it possible to better program cross-cutting concerns by specifying these concerns and then weave or compose them together into a coherent implementation. In this section, these mechanisms in the underlying AOP environment will be discussed detailed.

Cross-cutting concern is the most important reason about why Aspect Oriented Programming technique exists. Cross-cutting is a concern that repeats inside the cody which makes the code tangled. As an example, if you decide to implement exception handling in your code, you will possibly use try-catch code template in each function of the software project. As a result, because the try-catch code template is scattered throughout the implementation, we may state exception handling is a concern that cross-cuts the current system. Example of cross-cutting concerns can range from high level notions like security and quality of service to low-level notions such as caching, logging, exception handling, buffering ... etc. With Aspect Oriented Programming technique, you will possibly overhelm these kind of programming issues.

Join point is a mechanism in the underlying AOP environment that states a point during execution of a program. There are several types of points during program flow that can be used as join points. Some of these join points are as follows:

- Constructor call
- Constructor call execution
- Method call
- Method call execution
- Field get
- Field set
- Exception handler execution
- Class initialization
- Object initialization

As an example, “before cash transfer function starts to execute“ point can easily be captured by the usage of join points.

Pointcut is a mechanism in the underlying AOP environment that is a group of different join points. To represent a pointcut with the collection of join points, join points are connected with logical operators such as *AND, OR, XOR* ...etc. A pointcut can have one or more join points. If a control needs different conditions to be met during execution, a pointcut is defined that is a group of different join points with each states a specific point in the program flow.

Figure 3.1 shows an example pointcut written is AspectJ programming language syntax. The details of AspectJ programming language will be given in the following sections. *Set* pointcut in Figure 3.1 returns true if:

- An operation is done on an instance of *Nokta* class ***AND***
- If method executed starts with *set* letters with any arguments and return values.

```
pointcut set() : execution(* *.set*(..) ) && this(Nokta);
```

Figure 3.1 Example pointcut code

Advice is a mechanism in the underlying AOP environment that is used to execute a code segment when a pointcut returns true. As an example, sentence part written as italic style in the following sentence: "before cash transfer function starts to execute, *control whether active user that tries to make transfer has enough rights*" can easily be captured by the usage of advice.

Advice mechanism allows using of algorithms that are intended to solve cross-cutting concerns. The invocation of these specialized algorithms' implementations or implementations themselves are placed into the code body of advice. When a pointcut related with advice returns true, the implementations of these specialized algorithms are executed to solve cross-cutting concerns in an effective and modular way. Advice mechanism with pointcut specified has a critical role for the seperation of concerns.

There are three types of advices:

- Before advice is the simplest type of advice. Invoked before the join point is invoked.
- After advice has three specialized types.
    - After throwing advice runs if the join point throws an exception
    - After returning advice runs after join point is executed, if no exception is thrown.
    - Unqualified advice runs no matter what the outcome of the join point.
- Around advice is a good advice type that is supported in the AOP environment. It allows the execution of a given control other rather than the current control in the program flow.

Aspect is a mechanism in the underlying AOP environment that is constructed by the usage of join points, pointcuts, advices. It is a composed structure and like a class in the Object Oriented Programming technique.

Inter-type declaration is a mechanism in the underlying AOP environment that allows programmers to modify base code (component structure) without any modifications in base code. As an example, programmer can add a new property to an existing class in component class hierarchy.

Aspect Weaver is a tool that composes the base code and aspect code. Aspect weaver accepts the base code and aspect code as inputs and then outputs a coherent program. The woven output code may itself be source code in a language like C and other programming languages. The woven code is then compiled using a traditional compiler suitable with woven code's programming languge. Figure 3.2 illustrates the idea (Kiczales & et al., (n.d.)).

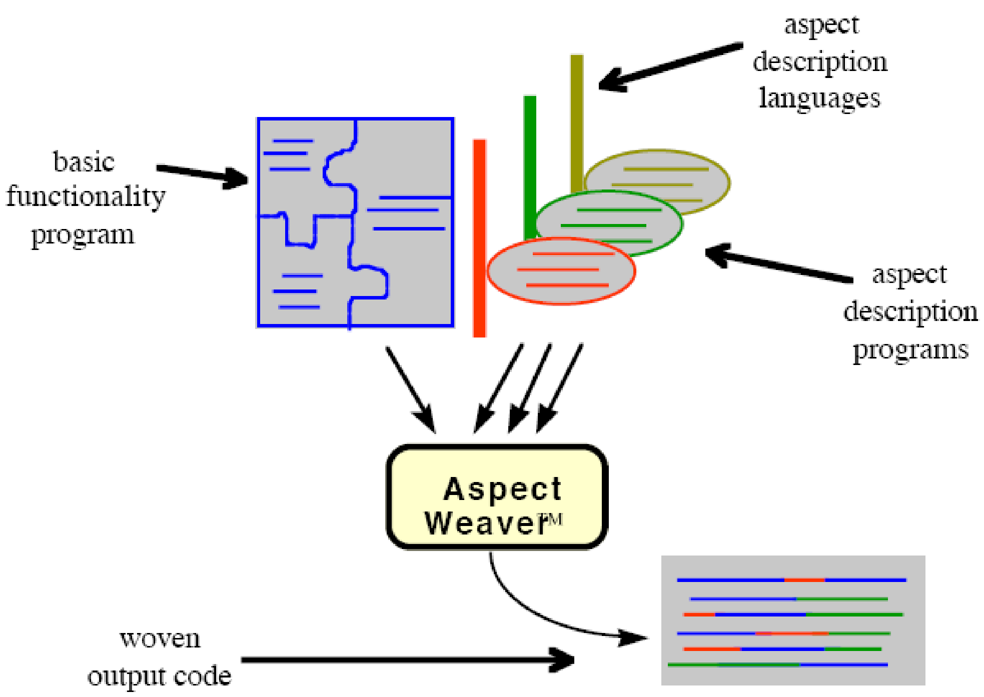

aspect description languages
basic functionality program
aspect description programs
Aspect Weaver™
woven output code

Figure 3.2 Aspect weaver behaviour

3.3 AOP Challenges

AOP technique provides some mechanisms to solve cross-cutting concerns in a more effective and modular way. AOP isn't a new computation theory that solves yet-unresolved problems. It is a programming technique that targets a specification problem, modularization of cross-cutting concerns.

This section discusses AOP technique challenges. Some of these challenges are given in a comparative mode with other programming techniques.

AOP is suitable for many cross-cutting concerns. In an AOP book or tutorial, you will possibly find an example of AOP used to implement logging and tracing. Because these examples are for the beginners and don't show complex details of AOP, it's commonly assumed that AOP is good just for tracing and logging.

Tracing and logging are the “hello world” examples of AOP. At the system level, security, transaction management, and thread-safety concerns can be implemented using AOP. Many business logic problems (low level functions) can also be implemented using AOP.

AOP usage follows the same path as Object Oriented Programming. When started to program in OOP technique, classes are written at first step, account, customer, window, and so on. After some experience in OOP, other kind of objects such as commands, actions, and observers are started to be programmed. Design patterns are started to be used. This evolution schema is the same for AOP. At first step, “hello world” examples are started to be programmed and then, more complex cross-cutting concerns are started to be implemented using sophisticated AOP constructs.

AOP does not solve any new problems. AOP isn’t a new computation theory that solves yet-unresolved problems. It is a programming technique that targets a specific problem, modularization of cross-cutting concerns. Aspects are a seperate layer that are built on top of the current implementation to solve cross-cutting concerns in an effective and modular way. AOP does not reject the current OOP technique in any way.

To solve cross-cutting concerns in the coding level, developers wrestle with implementation overhead again and again. AOP handles these kind of programming problems in a new way by reducing these overheads by the usage of aspects in the implementation.

AOP vs. well-designed interfaces. AOP is good way for seperation of concerns. AOP seperates base code from aspects making them largely independent of each other. This seperation makes it possible to swap implementations of a module without affecting other modules in the system. This similar strategy is also nearly valid in Object Oriented Programming technique by creating abstract interfaces which allow you to create various implementations of the same interface. In this situation, OOP developers say well-designed interfaces can be used instead of AOP. But a closer look at how each approach impacts on the application as a system tells a different story.

In AOP, base code is not aware of aspects, meaning that every cross-cutting concern implementation is only stored in aspects. This is not the same in OOP because the base code has the invocations to interface methods meaning that base code is aware of cross-cutting concerns' implementations. To use a different implementation with the same interface for a cross-cutting concern in AOP do not require any change in the base code. In OOP, base code must be modified to target new implementation.

AOP vs. design patterns. A second commonly argued alternative to AOP is the usage of design patterns which most developers are quite familiar. Design patterns represent solutions to recurring problems where OOP doesn't offer direct solutions. Design patterns definitely help improve modularizing cross-cutting concerns to some extent, but upon closer examination. If fact, design patterns can improve the complexity of the base implemetation as compared to using AOP techniques. Design patterns also require change in base code while AOP does not require.

AOP vs. application frameworks. A third alternative to AOP can be the usage of application frameworks. Some frameworks such as EJB are commonly used to modularize cross-cutting concerns. Such frameworks understand commonly encountered concerns in a particular domain, impose restrictions on user code and then implement the chosen concerns themselves. For example EJB (Enterprise Java Beans) can handle persistence management, transaction management, authentication and authorization, concurrency concerns automatically. But to use application frameworks instead of AOP has some disadvantages:

- Application frameworks have a learning curve which also exists in AOP technique. But the difference between frameworks and AOP is that; in AOP you will learn details for once and then use the same AOP constructs in development, in application frameworks you will have to learn for every new framework because the new frameworks offer new solution styles to solve the same problem.

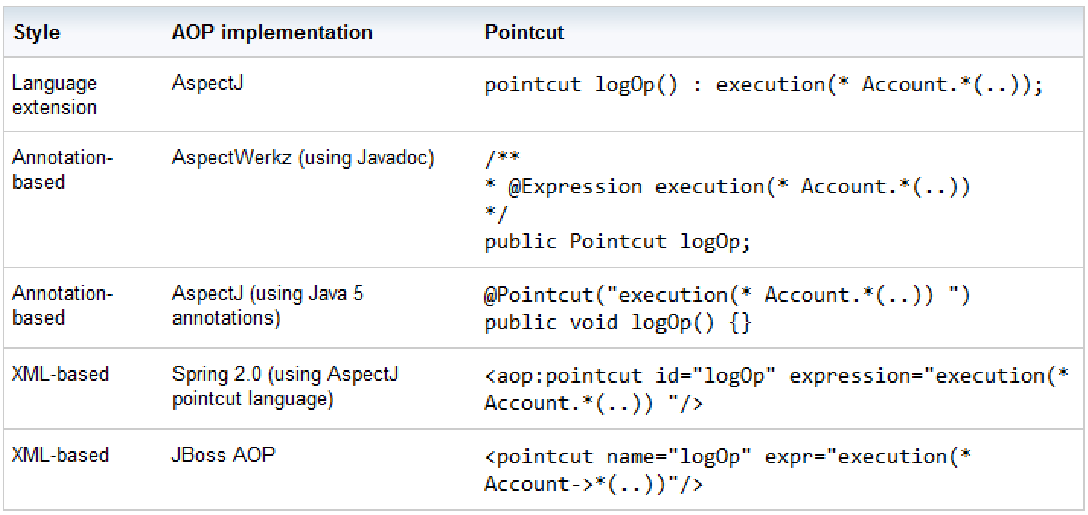
- You don’t have to like the solution that an application framework offers or the application framework may not offer a solution for a specific problem or a part of it. In these situations you have limited solutions to the problems which feel you helpless. Because AOP constructs are generic and can be used for a variety of domain problems, you are free to implement cross-cutting concerns in your own codes.

AOP raises the level of abstraction. AOP raises the level of abstraction more than it has ever been before. AOP modularizes cross-cutting concerns into aspects, seperate from classes. It removes cross-cutting concerns’ implementations from the base code. This leads to a higher level of abstraction because you need to make an effort to understand the aspects’ interactions with classes. The higher the abstraction of your classes means the less clear the program flow is.

But developers like the abstraction in programs. It is proved since the evolution of assembly language up to now. The program flow is hard to understand in higher abstraction levels but the program code is simpler. The modularization level raises. This makes the software developers focus more on the real business function implementations rather than focus on the cross-cutting concerns. The situation is the same for developers who implement the cross-cutting concerns. They focus more on the cross-cutting concerns’ details rather than other business function implementations. This kind of seperation of concerns mechanism leads to more productivity.

AOP simplifies debugging cross-cutting functionality. Debugging requires right tools that understand the exact type of an object and the exact control flow, give linear flow to the navigation between different entities. Despite aspects make control flow less explicit (raising the level of abstraction), choosing right aspect debuggers make cross-cutting functionality debugging easier. For example, with recent improvements to the Eclipse AJDT plug-in, debugging aspect-oriented programs is almost as easy as debugging object oriented ones.

Languages for AOP are very similar with each other. In fact, every AOP implementation uses a new language but these implementations are very similar to each other. Figure 3.3 lists some AOP implementations with pointcut construct compared in each of implementations. The same situation is also true for the other constructs in AOP implementations.



Style	AOP implementation	Pointcut
Language extension	AspectJ	`pointcut logOp() : execution(* Account.*(..));`

Annotation-based	AspectWerkz (using Javadoc)	`/**`
		`* @Expression execution(* Account.*(..))`
		`*/`
		`public Pointcut logOp;`

Annotation-based	AspectJ (using Java 5 annotations)	`@Pointcut("execution(* Account.*(..)) ")`
		`public void logOp() {}`

XML-based	Spring 2.0 (using AspectJ pointcut language)	`<aop:pointcut id="logOp" expression="execution(* Account.*(..)) "/>`
XML-based	JBoss AOP	`<pointcut name="logOp" expr="execution(* Account->*(..))"/>`

Figure 3.3 Pointcut definition in different AOP implementations

AOP can be adopted incrementally. Because aspects are a seperate layer than base code, there is minimum risk to make a software system AOP enabled. As aspects coded, these aspects can be included to the software system without any change in the base code.

For example, tracing aspects can be a good starting point to adopt AOP. Because tracing aspects are the “hello world” examples of AOP, developer has the chance to have some experience about the AOP constructs and AOP paradigms. After having experience with tracing aspects, writing other more complex cross-cutting concerns' implementation can be started.

Another good example of starter aspect can be writing a testing aspect. Here you can inject faults to the running system. For example you can inject a fault to simulate a network error. You have the chance to see the results of how the designed software behaves when such kind of a network error occurs. Can it be handled in a smart way or does the system crash?. For example you can inject a fault to simulate a database level error. You have the chance to see the results of whether transaction structure of the system is secure or not ... etc. These kind of aspects are also named as development aspects. Development aspects help in improving the code coverage and boosting confidence that you have a solid product.

All of these aspects offer a pluggability feature that does not force you to include aspects in production. Removing of these aspects from the software system before the production is so easy. Removing of aspect files from the project is the solution. No modification to the main code before production means no new risk before production too.

3.4 AspectJ

Gregor Kiczales and colleagues at Xerox PARC developed AspectJ as the most popular general purpose AOP implementation and made it available in 2001. IBM's research team then offered the more powerful but less usable Hyper/J which emphasizes the continuity of the practice of modularization of cross-cutting concerns. Beside these, there are some other AOP implementations such as AspectWerkz, JBoss-AOP, PostSharp, Spring, GlassBox, AspectC++ ...etc.

Aspect Oriented Programming in language AspectJ offers a great deal of power and improved modularity. AspectJ is the most popular general purpose AOP implemention which is available since 2001.

Because the Eclipse Foundation's AspectJ have been used in the application of this thesis, details of AspectJ language will be discussed detailed in this section.

AspectJ is an extension to Java, and the convention in Eclipse is to keep pure Java code (even in AspectJ projects) in .java files, and to use the .aj extension for source which uses AspectJ specific constructs. For example new aspects will be created in .aj files. This means that the Java editor is still, by default, used for .java files, and the AspectJ editor used for .aj files. This editor is an extension of the Java editor, so it can be used for Java code as well.

The AspectJ editor is designed to behave in an equivalent way for AspectJ code as the Java editor does for Java code. For example; breakpoints, watches are set in the sameway in AspectJ editor like in the Java editor. Because AspectJ editor extends Java editor, syntax colouring extends to AspectJ keywords such as aspect, pointcut, round, proceed ... etc.

For an experienced Java developer to become familiar with AspectJ language syntax is so simple because the AspectJ language uses Java programming language as base. When AspectJ language specific constructs are learned, writing the whole aspect code is the composition of the Java code and AspectJ language constructs.

Details of Eclipse Foundation's AspectJ language can be found at http://eclipse.org/aspectj/. This web site consists of different types of contents such as: links to AspectJ development tools, news and events on the AOP area, recent books and articles published about AOP, documents for AOP, bugs posted about AspectJ development tools ... etc.

3.4.1 Join points and Pointcuts

Consider the following Java class:

```
class Point
{
  private int x, y;

  Point(int x, int y) { this.x = x; this.y = y; }

  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }

  int getX() { return x; }
  int getY() { return y; }
}
```

setX method in this piece of program says that, when method named *setX* with an *int* argument called on an object of type *Point,* then the method body *this.x = y* is executed. One pattern that can be inferred from this description can be as the following: “when someting happens, then something gets executed.”

We can define instances of “things that happen” pattern as join points in AspectJ. Join points consist of things like method calls, method executions, object instantiations, constructor executions, field references, handler executions ... etc.

Pointcuts in AspectJ is the collection of join points and pickout these join points. For example the pointcut code below:

```
pointcut setter(): target(Point) &&
                   (call(void setX(int)) ||
                    call(void setY(int)));
```

pickouts each call to *setX(int)* or *setY(int)* when called on an instance of *Point* class.

Pointcut definitions consist of a left-hand side and a right-hand side, seperated by a colon. The left-hand side consists of the pointcut name and the pointcut parameters which correspond to the data available when the events happen (context exposing mechanism). The right-hand side consists of the pointcut itself, meaning that collection of join points.

Some of the example pointcuts are listed below:

execution(void Point.setX(int)): picked out when a particular method body executes.

call(void Point.setX(int)): picket out when a particular method is called.

handler(ArrayOutOfBoundsException): picked out when an exception handler executes.

this(SomeType): picket out when the object currently executing is of type *SomeType.*

target(SomeType): picket out when the target object is of type *SomeType.*

within(MyClass): picked out when the executing code belongs to class *MyClass*.

cflow(call(void Test.main())): picket out when the join point is in the control flow of a call to the *Test* class's *main* method with no argument supplied.

It is possible to use wildcards in the definition of pointcuts. For example;

execution(* *(..)): means the execution of any method regardless of return type and parameter types.

call(* set(..)): means the call to any method named *set* regardless of return type and parameter types.

call(* .new(int, int)): means the call to any class's constructor which takes exactly two arguments with type *int.*

Pointcuts compose through the operations *or ("||")*, *and ("&&")*, and *not ("!").* For example;

target(Point) && call(int *()): means any call to any method with return type *int* and no parameters on an instance of *Point*.

call(* *(..)) && (within(Line) || within(Point)): means any call to any method where the call is made from the code in *Line*'s or *Point*'s type declaration.

!this(Point) && call(int *(..)): means any call to any method with return type int and regardless of parameter types when the executing object is any type except *Point.*

Like classes, interface declarations can also be given in the definition of pointcuts. For example;

call(* MyInterface.*(..)): means any call to any method regardless of parameter types and regardless of return type in *MyInterface*'s signature.

When methods and constructors run, there are two different times associated with them. These are when they are called and when they are executed. These two interesting times are represented by call and execution join points, each of them has a different role in aspect programming.

At a call join point, the enclosing code is that of the call site. At an execution join point, the program is already executing the method, so the enclosing code is the method itself. For example;

call(void m()) && withincode(void m()): means any call to a method named *m* with no parameters and no return value where the call is made from the same *m* method. This pointcut only capture directly made recursive calls.

execution(void m()) && within(void m()): means the execution of a method named *m* with no parameters and no return values where the execution is in the same *m* method. This pointcut is the same as **execution(void m())** pointcut.

AspectJ has context exposing mechanism via pointcut parameters which correspond to data available when an event happens. Becasue these parameters will be accessible inside an advice, parameters improve the flexibility of AspectJ. For example;

```
pointcut setter(Point p): target(p) &&
                          (call(void setX(int)) || call(void setY(int)));
```

setter pointcut has one parameter of type *Point*. This means that any advice that uses this pointcut has access to a *Point* object from each join point picket out by *setter* pointcut.

```
pointcut testEquality(Point p1, Point p2): target(p1) &&
                                           args(p2) &&
                                           call(boolean equals(Object));
```

In the above *testEquality* pointcut, there are two parameters. *Args* is a special pointcut in AspectJ that makes it possible to access the parameters of a method picket out by the pointcut. In this example, *p1* is the target *Point* object and *p2* is the argument *Point* object to be compared with *p1*.

The use of the parameters in pointcuts is very flexible in AspectJ. The most important rule while using parameters is that: all pointcut parameters must be bound at every join point picked by the pointcut. For example, definition of the below pointcut will result in a compilation error:

```
pointcut wrongPointcut(Point p1, Point p2):
                              (target(p1) && call(void setX(int))) ||
                              (target(p2) && call(void setY(int)));
```

At the time one of the join points picket out by *wrongPointcut* pointcut, there is only one *Point* object as target. Meaning that pointcut parameters should be mapped with the join points' arguments and the target object.

3.4.2 Advice

Advice defines pieces of aspect implementation that execute at well-defined points during the execution of a program. Those points can be given by named pointcuts or by anonymous pointcuts. Anonymous pointcuts are the pointcuts which do not have a name.

AspectJ supports all types of advices given in AOP literature such as before advice, after advice and around advice.

The before advice code written below uses a named pointcut with name *setter*. In this example, it can also be seen that pointcut parameters are visible to the advice. These parameters are used by the advice code block for further processing.

```
pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                      (call(void setX(int) || call(void setY(int)));

before(Point p1, int newval): setter(p1, newval) {
   System.out.println("About to set something in " + p1 + " to the new value " + newval);
}
```

Before advice runs just before the join point picked out by the pointcut:

```
before(Point p, int x): setter(p, x) {
    if (!p.assertX(x)) return;
}
```

The after advice written below runs after each join point picked out by the pointcut, regardless of whether it returns normally or throws an exception:

```
after(Point p, int x): setter(p, x) {
          if (!p.assertX(x))
                    throw new PostConditionViolation();
}
```

The after returning advice written below runs after each join point picked out by the pointcut, but only if returns normally. The return value is also visible to the advice code block to be further processed by the advice. Below advice code uses an anonymous pointcut.

```
after(Point p) returning(int x): target(p) && call(int getX()) {
   System.out.println("Returning int value " + x + " for p = " + p);
}
```

The after throwing advice written below runs after each join point picket out by the pointcut, but only when it throws an exception of type *Exception.* Instance of the *Exception* type can also be accessed inside the advice to be further processed. The below example also uses an anonymous pointcut.

```
after() throwing(Exception e): target(Point) && call(void setX(int)) {
   System.out.println(e);
}
```

The around advice traps the execution of the join point; it runs instead of the join point. The original action associated with the join point can also be invoked by the special *proceed* call:

```
void around(Point p, int x): target(p)
                             && args(x)
                             && call(void setX(int)) {
   if (p.assertX(x))
         proceed(p, x);
   p.releaseResources();
}
```

3.4.3 Aspect

Aspect is a mechanism that is the composition of join points, pointcuts, advices ...etc. It is a composed structure and like a class in the Object Oriented Programming technique. Below is a complete aspect program with one pointcut and two advices.

```
public aspect SetterAspect
{
        pointcut setter(Point p1, int newval): target(p1) && args(newval)
                                        (call(void setX(int) || call(void setY(int)));

        before(Point p1, int newval): setter(p1, newval) {
            System.out.println("About to set something in " + p1 + " to the new value " + newval);
        }

        after(Point p, int x): setter(p, x) {
                if (!p.assertX(x)) throw new PostConditionViolation();
        }
}
```

CHAPTER FOUR

AOP in MDA

4.1 AOP in MDA Approach

The increasing complexity of current software applications, along with the emergence of new technologies and the demand of end users for a high quality in the software systems, require developers to deal with a growing set of software requirements. Examples of these requirements are concurreny, distribution, persistence, fault recovery, synchronization, authentication, authorization ... etc which affect a large number of components in the software system that cause the existence of cross-cutting concerns. Implementations of these cross-cuttings concerns scatter throughout the entire code which makes the code tangled. The code tangling hinder the comprehension, maintainability and evolution of software systems.

In order to manage cross-cutting behaviour issues which hinder the reusability, adaptability and modularity of a software system, a possible approach is to employ the principle Seperation of Concerns. There may possible be two dimensions in a software development process where the seperation of concerns principle appears. These are horizontal dimension and vertical dimension.

In the horizontal dimension, concerns that should be seperated appear in the same abstraction level of the system lifecycle (analyses, design, implementation ... etc). In the vertical dimesion, concerns that should be seperated appear in different abstraction levels of the system lifecycle. Concerns in both dimensions should be clearly seperated to increase the usability, adaptability and modularity of a software system.

Model Driven Architecture (MDA) process defines some abstraction levels for the clear seperation of concerns in the vertical dimension which are Computation

Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM). CIM is the most abstract level in modeling and only consists what the system is expected to do without any information technology details. PSM is the least abstract level with the details of a particular type of platform. Abstraction levels decrease through the transformation done from CIM to PIM and then, from PIM to PSM.

However, clear separation of concerns in the horizontal level is not addressed in MDA approach. MDA approach lacks mechanisms for identifying and separating cross-cutting concerns.

Aspect Oriented Programming (AOP) approach complements Object Oriented Programming (OOP) which may not be enough to clearly capture some programming problems. These problems are the seperation of cross-cutting concerns. AOP provides aspects: mechanisms beyond subprograms and inheritance for localizing the expression of a cross-cutting concern. By aggregating cross-cutting concerns into aspects, there will be no tangled code and this will result as making the aspect code and the base code easier to understand, maintain, develop ... etc.

The introduction of aspect-oriented constructs in programming has undoubtedly been one of the major advances in modularizing of software. The usage of AOP approach in programs improves the following qualities such as:

- The modules of a software are better modularized.
- The better modularization lead to a clear seperation of concerns and therefore the artifacts are better maintainable and reusable.
- The time-to-market is reduced by a better modularized and simpler design, resulting in reduction of costs.

If aspect-oriented constructs are applied in the early phases of software engineering, the improvement of the above mentioned qualities can also be applied to the artifacts at this stage which makes the appearance of Aspect Oriented Modeling (AOM).

AOP addresses the problem of seperation of concerns in the horizontal level. However, techniques used in the context of AOP concantrate in the system implementation phase (i.e code level). Therefore such techniques are more suitable for development processes in which the effort is made at the coding level.

Seperation of cross-cutting concerns (horizontal level) at the modeling level is being tackled in the area of Aspect Oriented Modeling (AOM). The most attention is being made on the programming languages level (AspectJ, Hyper/J ... etc). There are works in AOM which focus on techniques for the identification, analyses, management and representation of cross-cutting concerns in the modeling phase by using UML extension mechanisms (UML Profiles). Because there is the lack of automatic tool support for modeling and managing the relationships among the base model (component model) and the cross-cutting model, this has been an hindrance in the wide-spread adoption of AOM in the MDA approach.

The main approach can be summarized as the following:

- Raising the abstraction level of aspect modeling through the use of PIM models representing cross-cutting concerns independent on business models.
- Promoting the reuse of cross-cutting concerns modeled as PIM elements.
- Managing the relationships among the base model (component model) and the cross-cutting model at the modeling level (making the introduction of aspect-oriented constructs at the modeling level).

In this section, two related works that address the issue of seperation of cross-cutting concerns at the modeling level and then our aspect modeler tool which proposes a more pragmatic and efficient way for modeling aspects will be explained detailed.

4.2 Work 1: Weaving Security Aspects into UML 2.0 Design Models

In this section, we will give the details of an approach for systematically weaving security aspects into UML design models (Mouheb & et al., 2009). This approach provides an end-to-end approach for systematically weaving security aspects which are also built as UML models into UML design models (base models). The process starts with specifying the needed security requirements and ends with injecting the corresponding solutions at the appropriate locations in the design models.

The main steps of proposed approach are the following:

- Specification of Security Requirements: The designer should be able to specify the security requirements that he/she wants to enforce on his/her design. To achieve this, a UML profile is defined such that security requirements can be attached to UML design elements as stereotypes parameterized by tagged values.
- Specification of Security Solutions: The security expert provides a security solution as a security aspect for each security requirement covered by the security requirements specification profile.
- Definition of UML Join Points: A security solution mainly consists of security behaviours (advices in AOP jargon) that should be injected before/after/around some specific points (join points in AOP jargon) of UML design.
- Design weaving: This step represents the actual addition of security solutions into UML design.

Figure 4.1 illustrates the proposed approach:

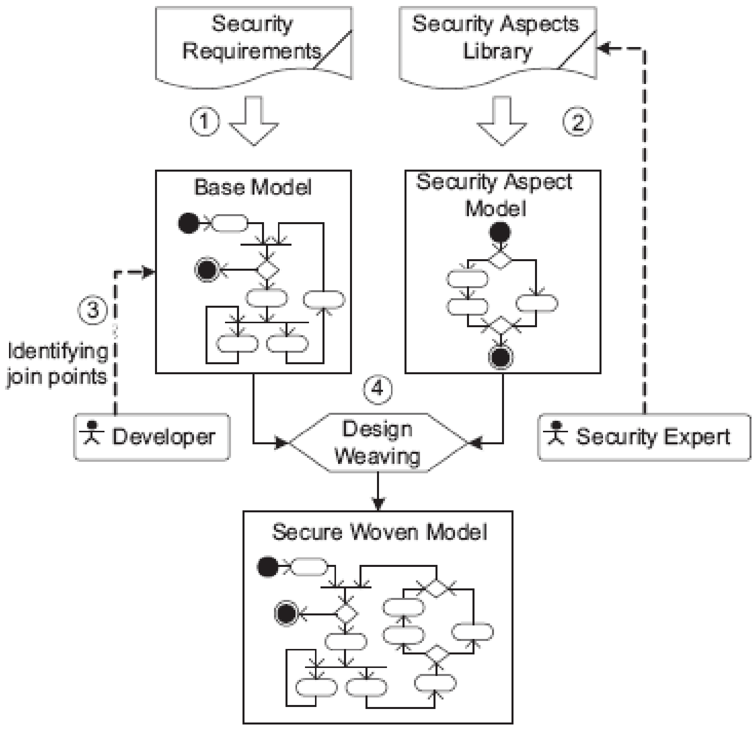
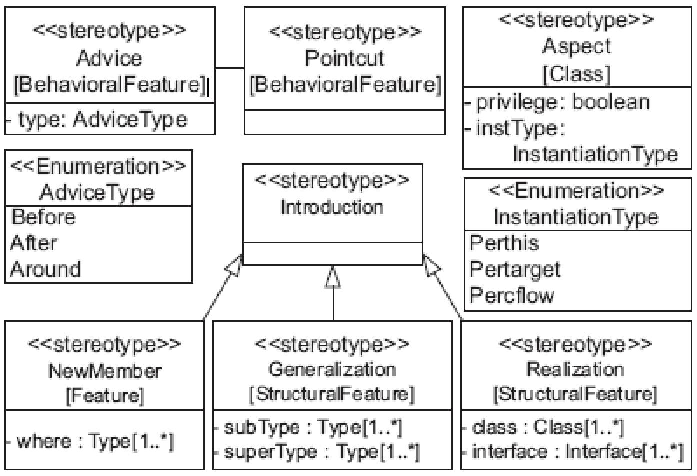


Figure 4.1 Overview of the proposed approach

This approach defines a UML Profile called AOSM Profile that allows the specification of security solutions at the modeling level.

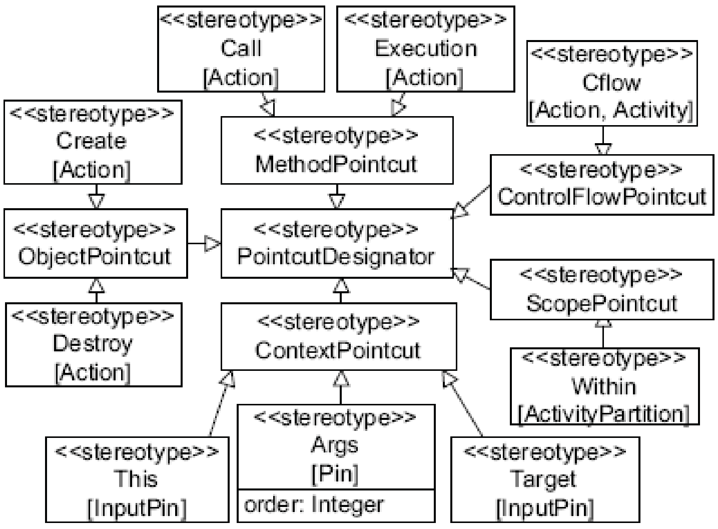
Aspects Specification: An aspect is modeled as a stereotyped class. Advices are modeled as special kind of operations stereotyped by the name <<advice>>. The advice behaviour is specified in behavioural diagrams (sequence diagrams, activity diagrams ... etc). The advice type is given by a tag *type* whose values are provided in the enumeration *AdviceType*. The location where an advice should be injected is specified by the meta-element *Pointcut*.



<<stereotype>> Advice [BehavioralFeature]
- type: AdviceType
<<stereotype>> Pointcut [BehavioralFeature]
<<stereotype>> Aspect [Class]
- privilege: boolean
- instType: InstantiationType
<<Enumeration>> AdviceType
Before
After
Around
<<stereotype>> Introduction
<<Enumeration>> InstantiationType
Perthis
Pertarget
Percflow
<<stereotype>> NewMember [Feature]
- where : Type[1..*]
<<stereotype>> Generalization [StructuralFeature]
- subType : Type[1..*]
- superType : Type[1..*]
<<stereotype>> Realization [StructuralFeature]
- class : Class[1..*]
- interface : Interface[1..*]

Figure 4.2 The Meta-Model specifying aspects

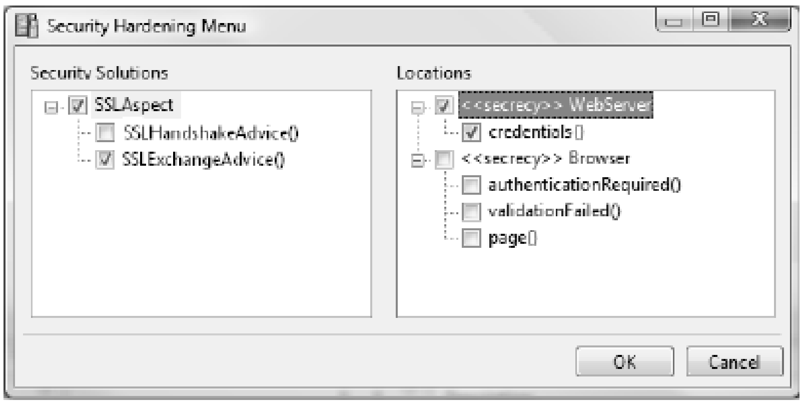
Pointcuts Specification: The meta-model proposed for the specification of pointcuts is presented in Figure 4.3.



<<stereotype>> Call [Action]
<<stereotype>> Execution [Action]
<<stereotype>> Cflow [Action, Activity]
<<stereotype>> Create [Action]
<<stereotype>> MethodPointcut
<<stereotype>> ControlFlowPointcut
<<stereotype>> ObjectPointcut
<<stereotype>> PointcutDesignator
<<stereotype>> ScopePointcut
<<stereotype>> Destroy [Action]
<<stereotype>> ContextPointcut
<<stereotype>> Within [ActivityPartition]
<<stereotype>> Args [Pin]
order: Integer
<<stereotype>> This [InputPin]
<<stereotype>> Target [InputPin]

Figure 4.3 The Meta-Model specifying pointcuts

The designer instantiates the pointcuts of the SSL aspect (i.e the name of the security aspect in the proposal) provided by the security aspect by choosing the elements of his/her model where the SSL advices should be injected. The instantiation process is implemented with a utility tool. Figure 4.4 illustrates the process:



SSLAspect [✓]
- SSLHandshakeAdvice() [ ]
- SSLExchangeAdvice() [✓]

<<secrecy>> WebServer [✓]
- credentials() [✓]

<<secrecy>> Browser [ ]
- authenticationRequired() [ ]
- validationFailed() [ ]
- page() [ ]

Figure 4.4 User defined join points

The actual join points where the SSL advices should be injected are identified and linked to the corresponding advices.

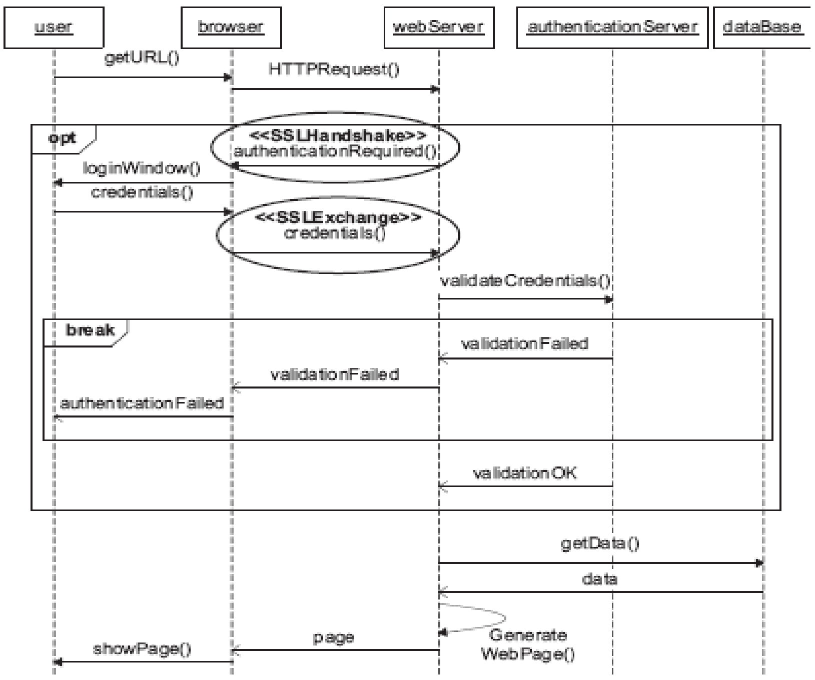


Figure 4.5 Identifying join points in the base model

As the last step, the advices of the SSL aspect are woven into the base model and a single woven model is produced. Weaving process is also done automatically. The specification of the SSL aspect using the AOSM profile is given in Figure 4.6. This specification makes it possible to represent aspects (or aspect-oriented constructs) at the modeling level.

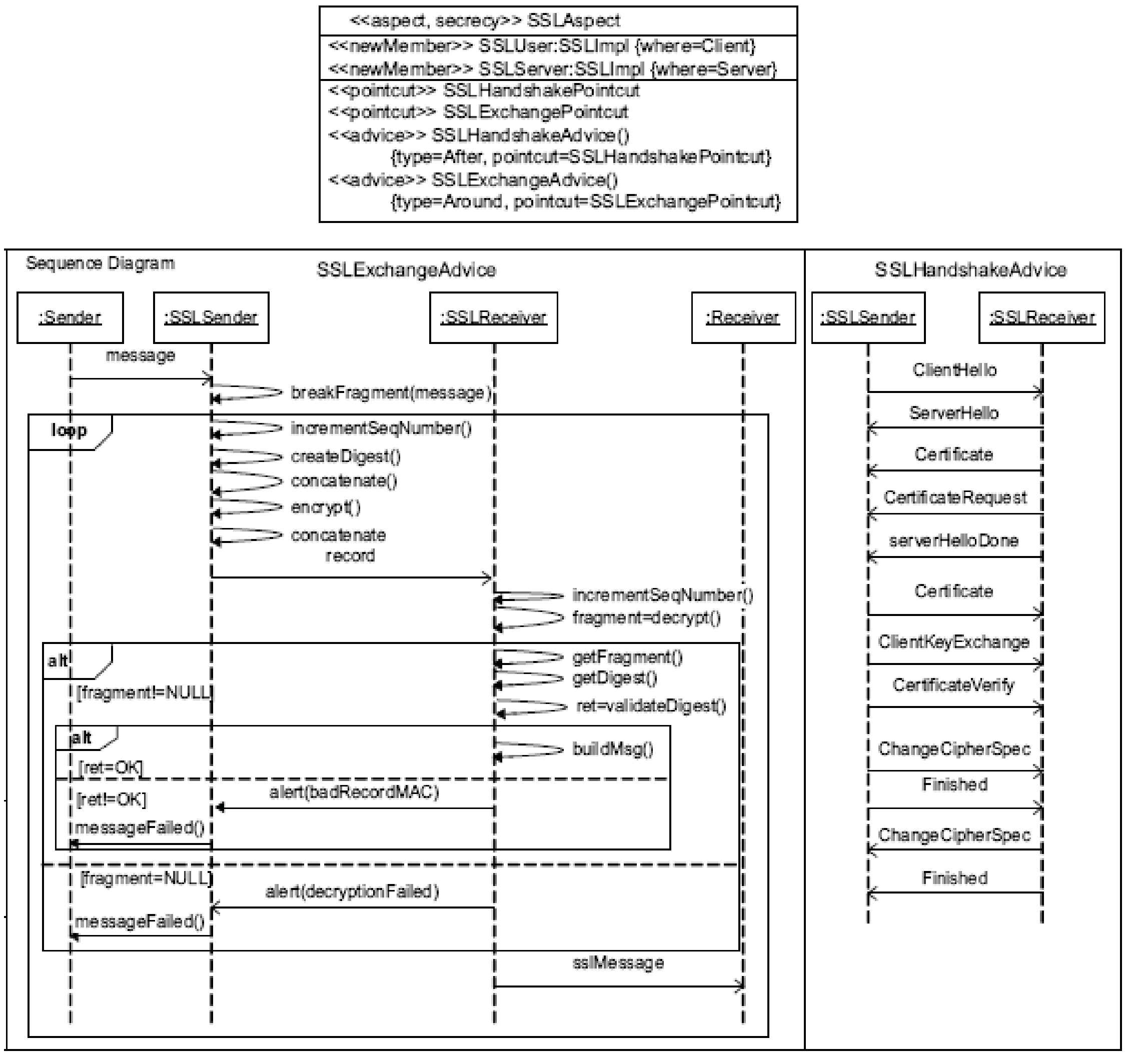


Figure 4.6 The specification of the SSL aspect using AOSM profile

4.3 Work 2:Designing and Weaving Aspect-Oriented Executable UML Models

In this section , we will give the details of an approach for designing and weaving aspect-oriented executable UML models (Fuentes&Sanchez, 2007). This approach focuses on two primary issues: modeling of aspect-oriented constructs (aspect, pointcut, join points ... etc) and execution of the built models (composition of the base model and aspect model) to simulate the system behaviour correctly. While explaining the details of how modeling of aspect-oriented constructs is done, we will somewhere give details of how a model is executed.

A straightforward and simple mechanism for visualising how a system model works when all design models are composed together, is to execute it and observe its behaviour. In order to make a software system model executable, this model must contain a complete and precise behaviour description. UML and its Action Semantics provides the basis for complete and precise behaviour modelling of software systems. Several tools conforming to UML and its Action Semantics has the ability to execute UML models.

The main steps of constructing aspect-oriented UML executable models can be defined as the following:

- First, a common UML executable model is constructed to model the non cross-cutting concerns, i.e, the base model.
- Cross-cutting concerns are modelled as aspects using the AOEM (Aspect Oriented Executable Model) profile. AOEM profile is the UML Profile that has been built special to this approach.
- How cross-cutting concerns must be composed with the concerns they cross-cut is specified by means of a pointcut model. AOEM is also used for the specification of the pointcut model.
- The base model and aspect models are composed, which produces the woven model. The woven model is also an UML executable model.

To perform the weaving operation (weaving the base model and the aspect model), designed models are exported to XMI format which is also an OMG standard. After some model transformation steps on models which are represented in XMI files, a single XMI representation of the woven model is produced. The woven model can then be imported to any tools conforming to OMG modeling standards to be executed.

An Online Book Store System example used to illustrate the approach. In this system some concerns: Persistence, Encryption, Currency Conversion are the cross-cutting concerns. Despite all the study to model these cross-cutting concerns is given in the original document, we will only show the process of how Persistence cross-cutting concern is modelled and woven to the base model.

In order to construct executable models, two basic elements are required; an action language and an operational semantics.

The operational semantics of UML is still in the process of standardisation. Several tools implementing non-standard operational semantics for UML models already exist (iUML, Rational Rose RT, Rhapsody ... etc).

The idea behind the operational semantics of UML is quite simpler to undertand. Firstly, the global system structure is established as a set of components. Then, the structure of each component is detailed as a set of class diagrams. Then the behaviour of each class is detailed by using state machines. Transitions and states in state machines have associated procedures which are specified using an action language.

Class diagram in Figure 4.7 shows the structure of the Online Book Store System.

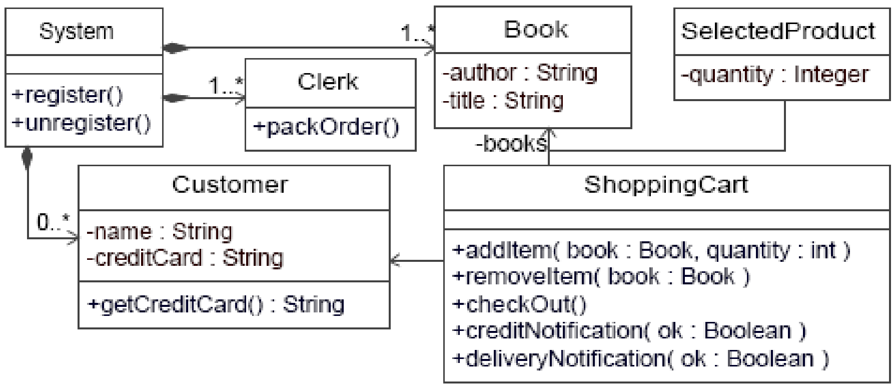


Figure 4.7 Class diagram for OBS system

Statechart diagram in Figure 4.8 shows the behaviour of ShoppingCart class. Each procedure in this diagram will be specified using the UML action language.

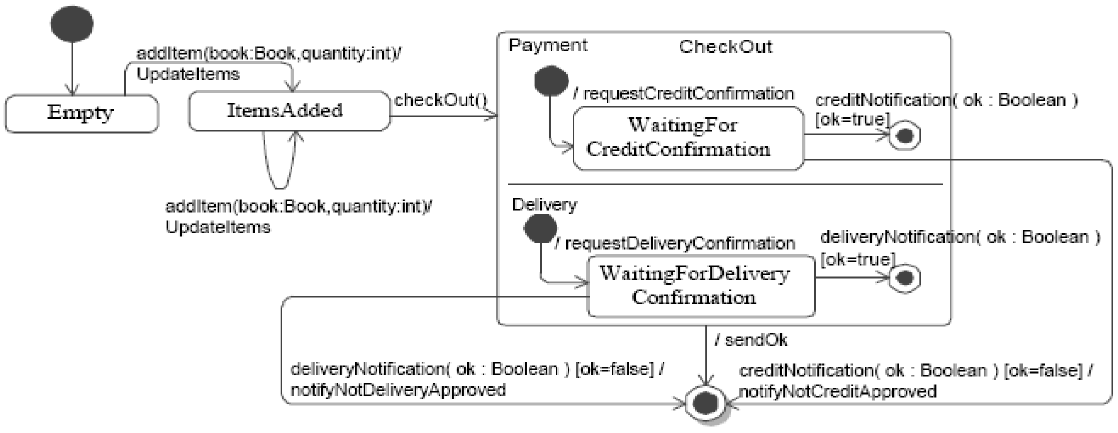

addItem(book:Book,quantity:int)/
UpdateItems
Empty
ItemsAdded
checkOut()
addItem(book:Book,quantity:int)/
UpdateItems
Payment
CheckOut
/ requestCreditConfirmation
WaitingFor
CreditConfirmation
creditNotification(ok : Boolean)
[ok=true]
Delivery
/ requestDeliveryConfirmation
WaitingForDelivery
Confirmation
deliveryNotification(ok : Boolean)
[ok=true]
/ sendOk
deliveryNotification(ok : Boolean) [ok=false] /
notifyNotDeliveryApproved
creditNotification(ok : Boolean) [ok=false] /
notifyNotCreditApproved

Figure 4.8 Statechart diagram for shoppingcart class

The UML action language does not enforce any notation for drawing actions. A new UML profile that is special to this approach has been developed to specify the set of actions. Advice behaviours while modeling aspects will also be modelled using a subset of this profile.

The designed profile works as follows: procedures are represented by UML activity diagrams (meaning that, advice behaviours while modeling aspects will also be drawn as activity diagrams). Actions are nodes of activity diagrams. Inputs and outputs are depicted as pins. To distinguish each specific action in the activity diagram, some stereotypes for these special actions have been built.

Figure 4.9 shows the behaviour of *UpdateItems* procedure in Figure 4.8. <<ReadSelf>>, <<CreateLinkObject>>, <<AddStructuralFeature>> are stereotypes to distinguish each action.

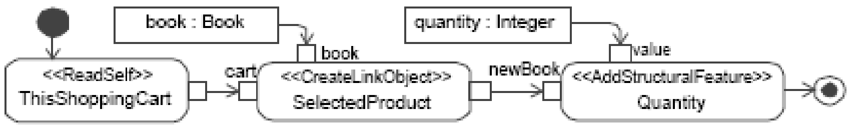


Figure 4.9 UpdateItems procedure

An **aspect** is modelled as a common class with special operations which model advices. Advices differ from common operations in that they are never invoked explicitly and they are executed by the aspect-oriented weaver without the knowledge of the base class designer. For this reason, advices do not have parameters. Consequently, each aspect-oriented language has to provide some mechanisms to allow advices to retrieve the information related to the join point. In AOEM profile, actions to retrieve the information related to the join point are represented with special stereotypes. Advices are modeled using activity diagrams.

Figure 4.10 shows the *Persistence* aspect model which is a cross-cutting concern in Online Book Store System. As seen from the figure, *Persist* advice is modelled AOEM profile that uses activity diagrams to depict the behaviour of the advice.

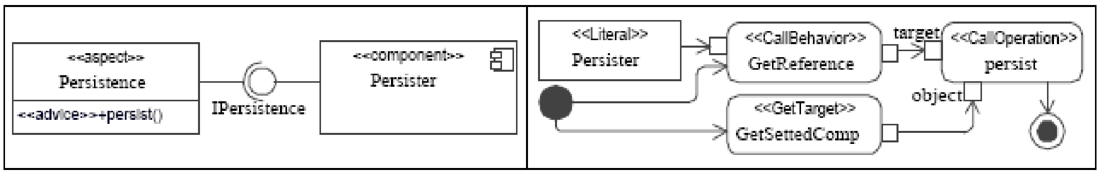


Figure 4.10 Persistence aspect model

To complete aspect-oriented model, pointcut models that specify how to compose the cross-cutting concerns modelled as aspects with the design they cross-cut are built.

A pointcut expression is a pattern that matches several join points and associates them with one or more advices. At the modeling level, UML diagrams with wildcards (e.g., “*” to represent any sequence of characters or “?” to represent any sequence of arguments) are used to model pointcuts. In this approach, to intercept interactions between objects, sequence diagrams are selected to model pointcuts.

A pointcut, according to the AOEM profile, is expressed by means of a sequence diagram, stereotyped as <<pointcut>>. This stereotype has a tagged value called *advice:* an ordered collection of aspect advices. The specific message of the sequence diagram that must be intercepted is stereotyped as <<joinpoint>>. <<joinpoint>> steretype has two tagged values:

- Point: which indicates whether the interception point is either the sending or the reception of the message.
- Time: which specifies when the advice is executed related to the join point (before, after, around)

Figure 4.11 shows the pointcut model for Persistence aspect. This pointcut returns true after any method starting with *“add”* pattern on the ShoppingCart class is executed.

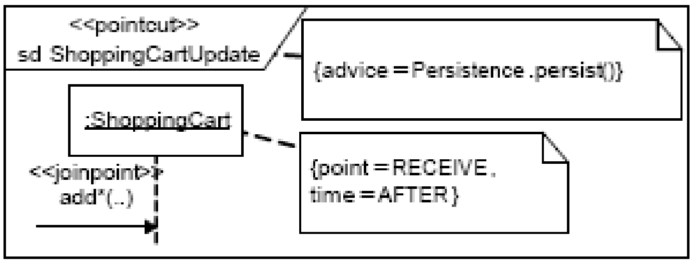


Figure 4.11 Pointcut model for Persistence aspect

Up to here; base model, aspect model and the pointcut model have been built. However, to be able to execute entire model set, aspect behaviours must be added to the modules they cross-cut according to the pointcut specifications, i.e., the weaving process has to be executed.

The task of the model weaver is to inject the advice behaviours into the places indicated by the pointcut specifications. The weaving process is defined as a chain of model transformations illustrated in Figure 4.12.

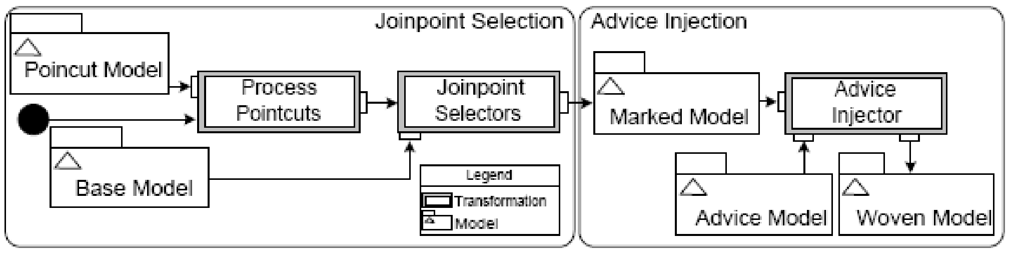


Figure 4.12 Model weaving process

The pointcut model is processed by the ProcessPointcuts model transformation , which generates a set of model transformations, called JoinpointSelectors. A JoinpointSelector serves to search all the joinpoints that are selected by a pointcut. These joinpoints are stereotyped as <<selectedjoinpoint>>, and the JoinpointSelector adds two tagged values to this stereotype: the advice that must be executed on that joinpoint and the advice execution time (before, after, around). This information will be required by the AdviceInjector model transformation in the next step. After applying the

JoinpointSelectors to the base model, the marked model, how and where advices must be injected is obtained.

In the AdviceInjector model transformation step, the corresponding advices must be injected into the selected joinpoints. The AdviceInjector model transformation takes as inputs the MarkedModel and the AdviceModel and outputs the woven model.

Figure 4.13 shows the woven model when Persistence advice is injected into the UpdateItems procedure it cross-cuts.

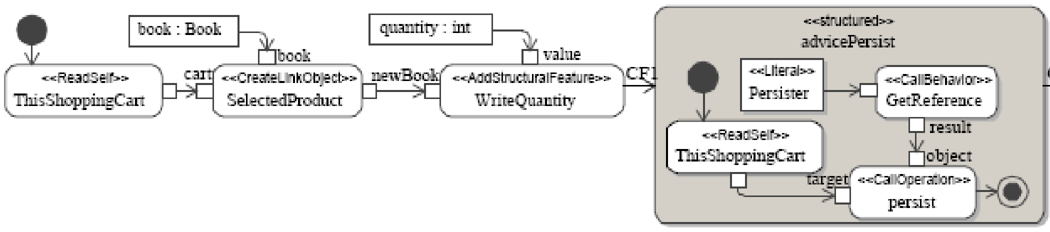

book : Book
book
quantity : int
value
<<ReadSelf>> ThisShoppingCart
cart
<<CreateLinkObject>> SelectedProduct
newBook
<<AddStructuralFeature>> WriteQuantity
CF1
<<structured>> advicePersist
<<Literal>> Persister
<<CallBehavior>> GetReference
result
object
<<ReadSelf>> ThisShoppingCart
target
<<CallOperation>> persist
CF2

Figure 4.13 The woven model

4.4 Our Aspect Modeler Tool (AspectModeler)

We have designed and developed an aspect modeler tool named as AspectModeler that allows the definition of aspects in a practical and efficient way. The output of the AspectModeler is the programming code (i.e., AspectModeler tool makes the code generation of the modeled aspects automatically) and this feature of AspectModeler makes it an end-to-end product that adopts aspects automatically into the project design environment.

As mentioned in the above related works for aspect-oriented software modeling, there are mainly three issues to be solved; specification of a UML Profile that represents aspect-oriented constructs (aspect, advice, pointcut ... etc) in UML models, poincut modeling mechanism that matches several join points and associates them with one or more advices, weaving of aspect models with the base models as the last step.

Becasue UML does not support Aspect Oriented Modeling in its standard specification, the primary solution to represent aspect-oriented constructs in UML is the usage of UML Profile mechanism. UML Profiles generated by different tool vendors are different than each other. This hinders the full adoption of Aspect Oriented Modeling to the software modeling area. As a result, model transformation languages lack stability and maturity to deal with UML Profiles.

To overhelm the issue of weaving aspect models with the base models, AspectModeler proposes an idea that delegates this job to the existing and mature aspect weaver tools. AspectModeler allows developers to build their own models (the base models and the aspect models) as UML models without specification of any UML Profile for aspect-oriented constructs, i.e., the developers that build their own models should not be aware of whether their models will be an aspect model or a base model. Models are pure UML models without aspect-oriented constructs. AspectModeler only uses these models without any modification that enables software developers to model aspects and includes aspect models into the existing software system by generating aspect oriented programming code in a seperate file. Because this seperate file contains a valid code in a specific programming language, exisiting aspect weaver tools related to the generated code can be used to produce a complete and correct aspect-oriented behaviour. This approach eliminates the model weaving step and delegates it to the existing technologies.

Modeling of aspect-oriented constructs (aspect, advice, pointcut ...etc) is done with user friendly interfaces that AspectModeler contains. These interfaces allow software developers, who are intended to include aspects into the existing software, to easily define and manage aspect-oriented constructs at the modeling level. Code generation in AspectModeler is not a last step action, i.e., software developer can also preview the programming code of the aspect-oriented constructs he/she defined so far.

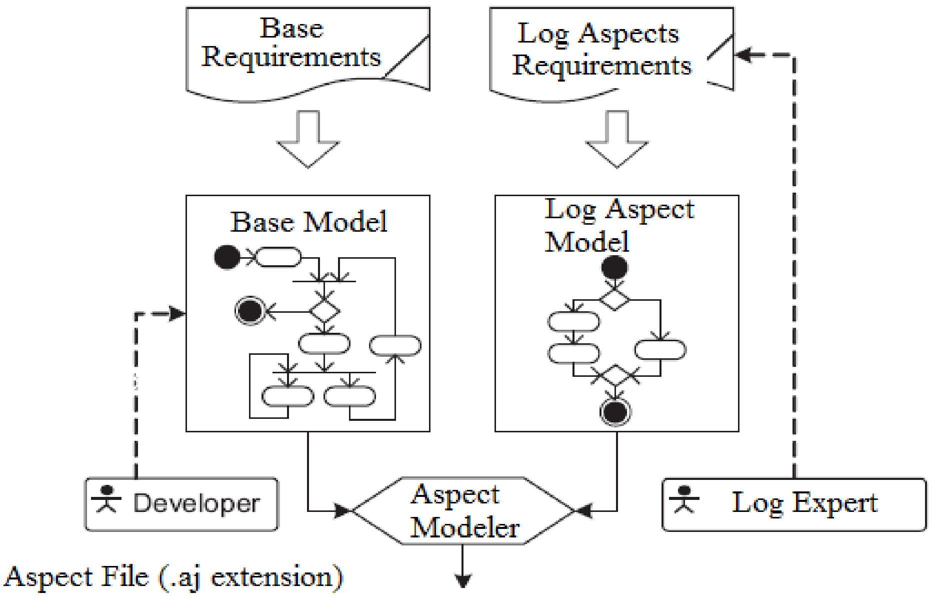
The program code generated by the AspectModeler is in AspectJ programming language. Becasue AspectJ is the most popular and dominant programming language in AOP environment, the output of the AspectModeler will possibly be valid in a wide range of applications. The output of AspectModeler is an aspect file with *.aj* extension and inclusion of the code generated by Aspect Modeler to the existing project's design environment is simply putting the generated aspect file into the existing project's design environment.

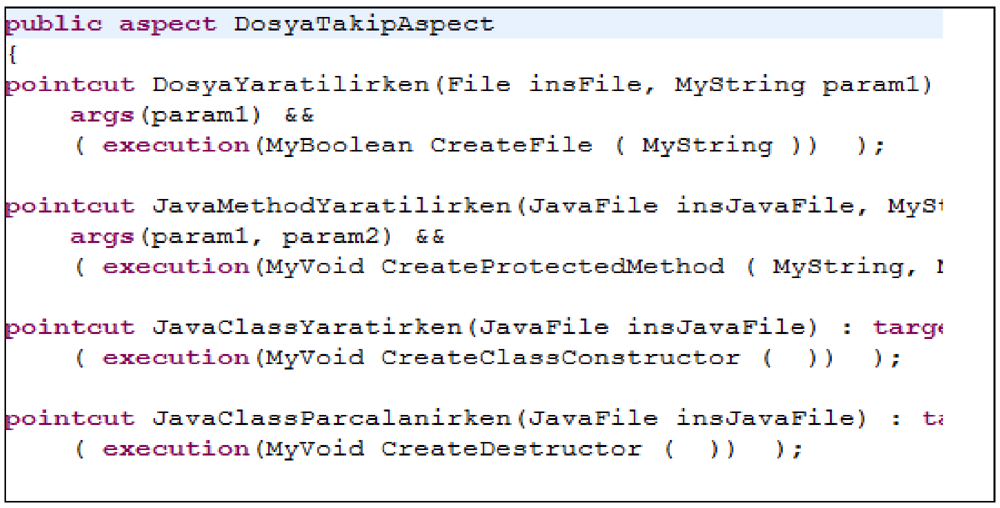
AspectModeler takes two inputs: base model and aspect model. Inputs are in XMI format which is an OMG standard. Models in XMI format are processed according to the XMI standard specification and shown to the AspectModeler user in an user friendly interface to clearly model aspects. The input mechanism enforced by AspectModeler makes it a product conforming to the OMG standards, i.e., inputs don't constitute a problem as long as they are produced through an MDA-enabled tool. Figure 4.14 illustrates the approach proposed by AspectModeler.

While explaining AspectModeler tool in this section, we will use two models; one for base model and the other for aspect model.

Base model has been built for modelling a programming language file processing system.This system has the following functionalities:

- A file can be created or deleted. File existence can be checked according to a given FilePath parameter.
- A specific substring in the file can be highlighted.
- Programming language specific constructs such as constructor, method, property, destructor can be generated. There are two programming languages supported: C# and Java in this version.





```
public aspect DosyaTakipAspect
{
pointcut DosyaYaratilirken(File insFile, MyString param1)
    args(param1) &&
    ( execution(MyBoolean CreateFile ( MyString ))  );

pointcut JavaMethodYaratilirken(JavaFile insJavaFile, MyS
    args(param1, param2) &&
    ( execution(MyVoid CreateProtectedMethod ( MyString, I

pointcut JavaClassYaratirken(JavaFile insJavaFile) : targe
    ( execution(MyVoid CreateClassConstructor (  ))  );

pointcut JavaClassParcalanirken(JavaFile insJavaFile) : ta
    ( execution(MyVoid CreateDestructor (  ))  );
```

Figure 4.14 AspectModeler approach

Aspect model has been built for modelling a log system for the programming language file processing system (base model). This system has logger classes specialized for each of the programming language constructs: PropertyLogger, MethodLogger, ClassLogger ... etc. Software developers who are intended to use log aspect can trace the running system with comprehensive log messages specific to each functionality in the running system.

Static structures of the aspect model and the base model are shown in Figure 4.15 and Figure 4.16, respectively. Diagrams both in Figure 4.15, Figure 4.16 and in the other parts of this section is built by using ArgoUML tool v0.28.1 (http://argouml.tigris.org). This tool has a very advanced support for UML diagrams and import/export utilities for XMI files that we have needed while developing and testing AspectModeler tool.

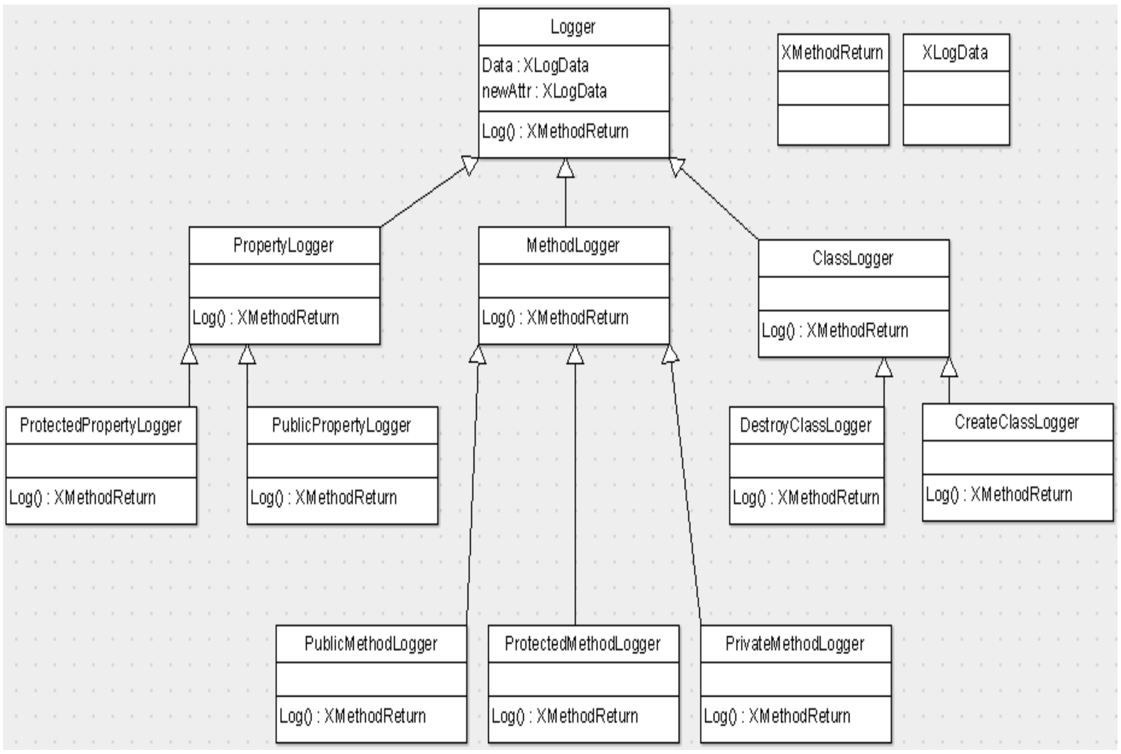


Figure 4.15 Log aspect model

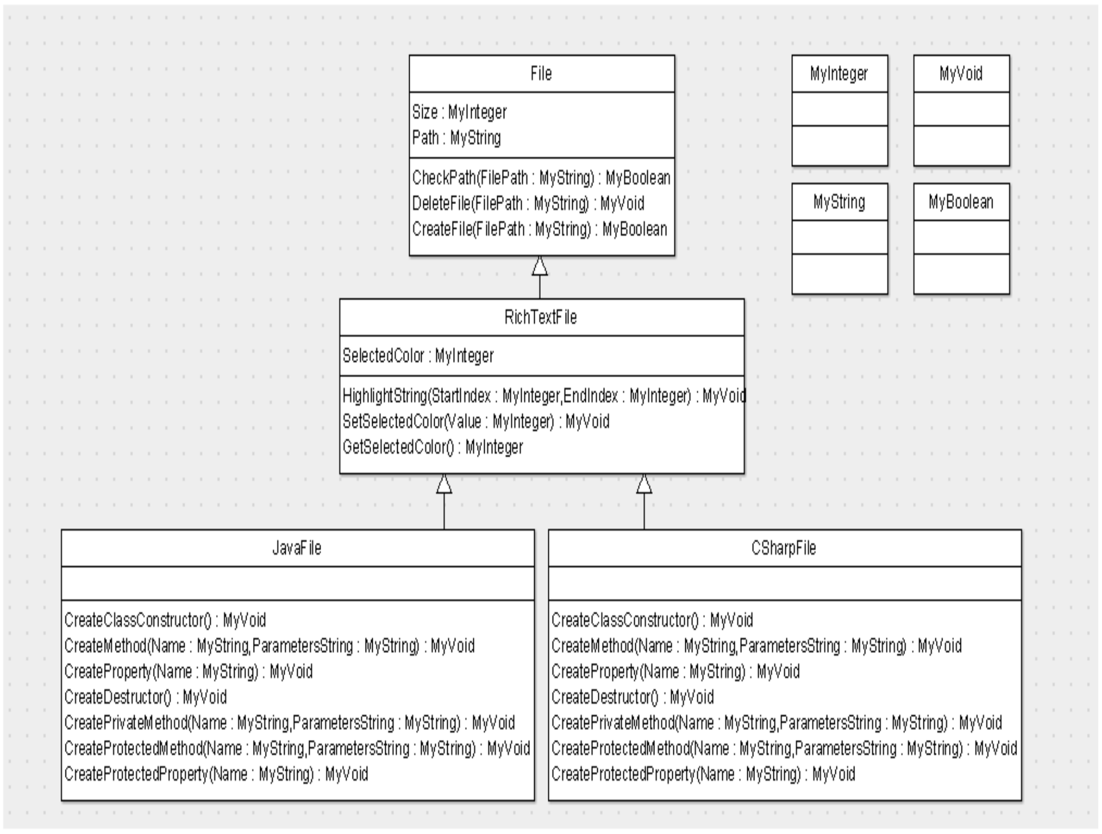

File
Size : MyInteger
Path : MyString
CheckPath(FilePath : MyString) : MyBoolean
DeleteFile(FilePath : MyString) : MyVoid
CreateFile(FilePath : MyString) : MyBoolean
MyInteger
MyVoid
MyString
MyBoolean
RichTextFile
SelectedColor : MyInteger
HighlightString(StartIndex : MyInteger,EndIndex : MyInteger) : MyVoid
SetSelectedColor(Value : MyInteger) : MyVoid
GetSelectedColor() : MyInteger
JavaFile
CreateClassConstructor() : MyVoid
CreateMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProperty(Name : MyString) : MyVoid
CreateDestructor() : MyVoid
CreatePrivateMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProtectedMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProtectedProperty(Name : MyString) : MyVoid
CSharpFile
CreateClassConstructor() : MyVoid
CreateMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProperty(Name : MyString) : MyVoid
CreateDestructor() : MyVoid
CreatePrivateMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProtectedMethod(Name : MyString,ParametersString : MyString) : MyVoid
CreateProtectedProperty(Name : MyString) : MyVoid

Figure 4.16 PL file processing system model

4.4.1 Input Models

AspectModeler takes two inputs: base model and aspect model. AspectModeler expects input models in XMI format. According to the XMI standard specification, input files are processed and a more human readable representation of the models are shown to the user. AspectModeler now supports XMI files with version 1.0 and version 1.2.

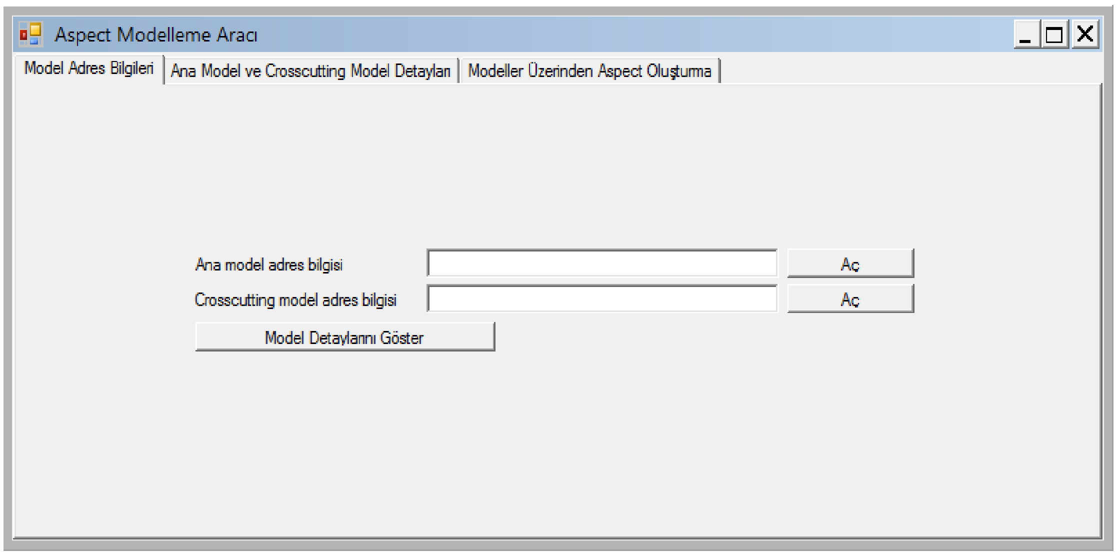
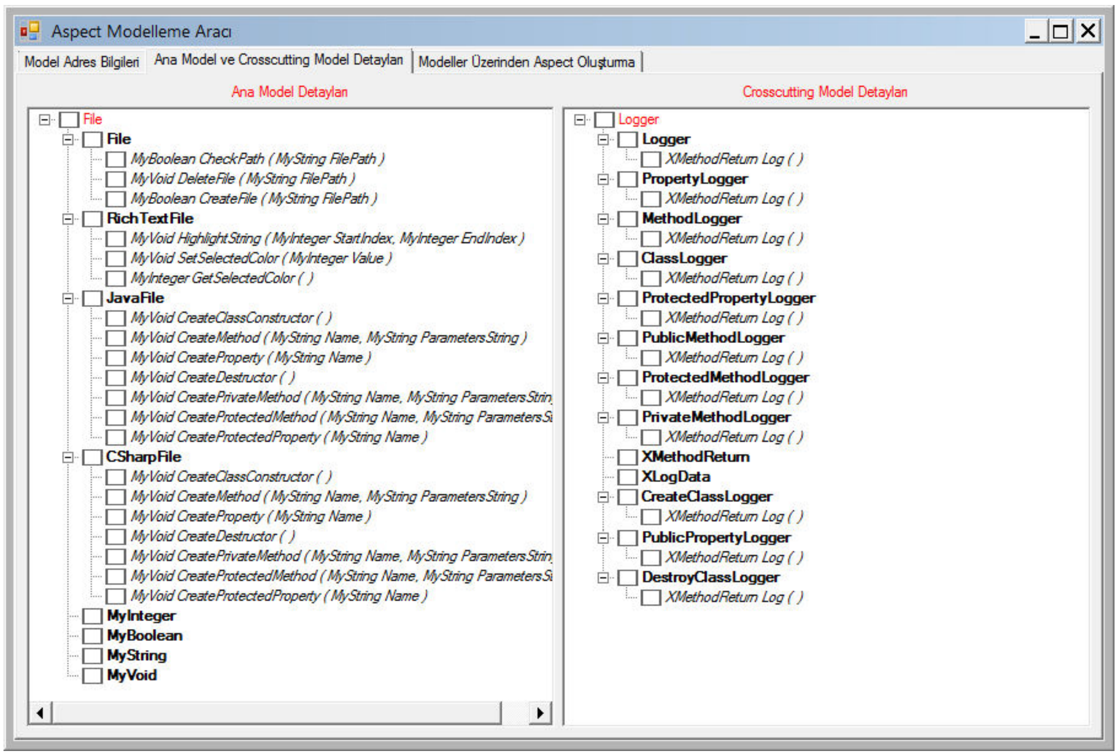

Aspect Modelleme Aracı
Model Adres Bilgileri
Ana Model ve Crosscutting Model Detayları
Modeller Üzerinden Aspect Oluşturma
Ana model adres bilgisi
Aç
Crosscutting model adres bilgisi
Aç
Model Detaylarını Göster

Figure 4.17 Input model files


Aspect Modelleme Aracı
Model Adres Bilgileri
Ana Model ve Crosscutting Model Detayları
Modeller Üzerinden Aspect Oluşturma
Ana Model Detayları
Crosscutting Model Detayları

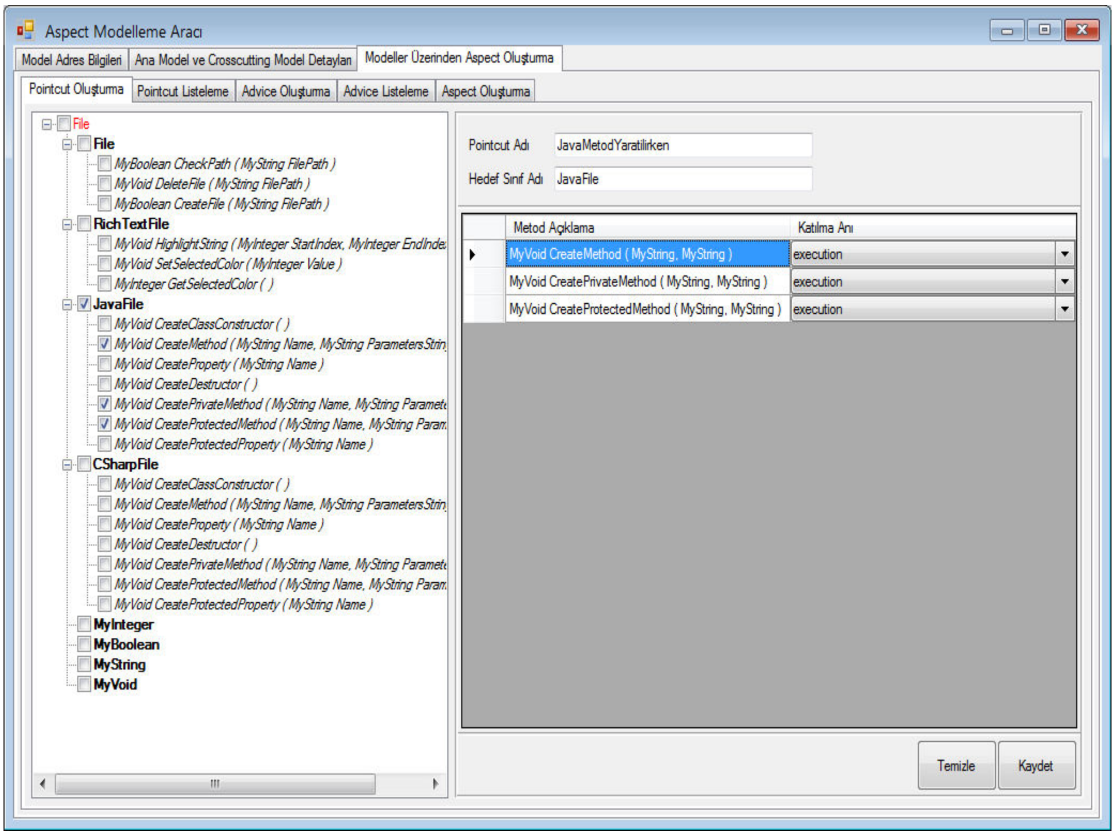
```
File
  File
    MyBoolean CheckPath ( MyString FilePath )
    MyVoid DeleteFile ( MyString FilePath )
    MyBoolean CreateFile ( MyString FilePath )
  RichTextFile
    MyVoid HighlightString ( MyInteger StartIndex, MyInteger EndIndex )
    MyVoid SetSelectedColor ( MyInteger Value )
    MyInteger GetSelectedColor ( )
  JavaFile
    MyVoid CreateClassConstructor ( )
    MyVoid CreateMethod ( MyString Name, MyString ParametersString )
    MyVoid CreateProperty ( MyString Name )
    MyVoid CreateDestructor ( )
    MyVoid CreatePrivateMethod ( MyString Name, MyString ParametersStrin
    MyVoid CreateProtectedMethod ( MyString Name, MyString ParametersSt
    MyVoid CreateProtectedProperty ( MyString Name )
  CSharpFile
    MyVoid CreateClassConstructor ( )
    MyVoid CreateMethod ( MyString Name, MyString ParametersString )
    MyVoid CreateProperty ( MyString Name )
    MyVoid CreateDestructor ( )
    MyVoid CreatePrivateMethod ( MyString Name, MyString ParametersStrin
    MyVoid CreateProtectedMethod ( MyString Name, MyString ParametersSt
    MyVoid CreateProtectedProperty ( MyString Name )
  MyInteger
  MyBoolean
  MyString
  MyVoid
```

```
Logger
  Logger
    XMethodReturn Log ( )
  PropertyLogger
    XMethodReturn Log ( )
  MethodLogger
    XMethodReturn Log ( )
  ClassLogger
    XMethodReturn Log ( )
  ProtectedPropertyLogger
    XMethodReturn Log ( )
  PublicMethodLogger
    XMethodReturn Log ( )
  ProtectedMethodLogger
    XMethodReturn Log ( )
  PrivateMethodLogger
    XMethodReturn Log ( )
  XMethodReturn
  XLogData
  CreateClassLogger
    XMethodReturn Log ( )
  PublicPropertyLogger
    XMethodReturn Log ( )
  DestroyClassLogger
    XMethodReturn Log ( )
```

Figure 4.18 Representation of XMI files as treeview

4.4.2 *Poincut Modeling*

Pointcuts are modelled through a seperate interface in AspectModeler. On the left panel, static structure of the base model is shown to pick up join points easily. One pointcut can have one or more join points each specifies a point during the program flow. Join points are listed in the editable grid component which allows the software developer to change the type of the join points (call, execute ... etc). When an item that represents a method is clicked on the left part, join point definition that specifies the selected method call is automatically added to the join point list grid.



- File
  - File
    - MyBoolean CheckPath (MyString FilePath)
    - MyVoid DeleteFile (MyString FilePath)
    - MyBoolean CreateFile (MyString FilePath)
  - RichTextFile
    - MyVoid HighlightString (MyInteger StartIndex, MyInteger EndInde
    - MyVoid SetSelectedColor (MyInteger Value)
    - MyInteger GetSelectedColor ( )
  - [✓] JavaFile
    - MyVoid CreateClassConstructor ( )
    - [✓] MyVoid CreateMethod (MyString Name, MyString ParametersStrin
    - MyVoid CreateProperty (MyString Name)
    - MyVoid CreateDestructor ( )
    - [✓] MyVoid CreatePrivateMethod (MyString Name, MyString Paramet
    - [✓] MyVoid CreateProtectedMethod (MyString Name, MyString Param
    - MyVoid CreateProtectedProperty (MyString Name)
  - CSharpFile
    - MyVoid CreateClassConstructor ( )
    - MyVoid CreateMethod (MyString Name, MyString ParametersStrin
    - MyVoid CreateProperty (MyString Name)
    - MyVoid CreateDestructor ( )
    - MyVoid CreatePrivateMethod (MyString Name, MyString Paramet
    - MyVoid CreateProtectedMethod (MyString Name, MyString Param
    - MyVoid CreateProtectedProperty (MyString Name)
  - MyInteger
  - MyBoolean
  - MyString
  - MyVoid

Pointcut Adı: JavaMetodYaratilirken
Hedef Sınıf Adı: JavaFile

Metod Açıklama	Katılma Anı
MyVoid CreateMethod (MyString, MyString)	execution
MyVoid CreatePrivateMethod (MyString, MyString)	execution
MyVoid CreateProtectedMethod (MyString, MyString)	execution

Figure 4.19 Pointcut modeling

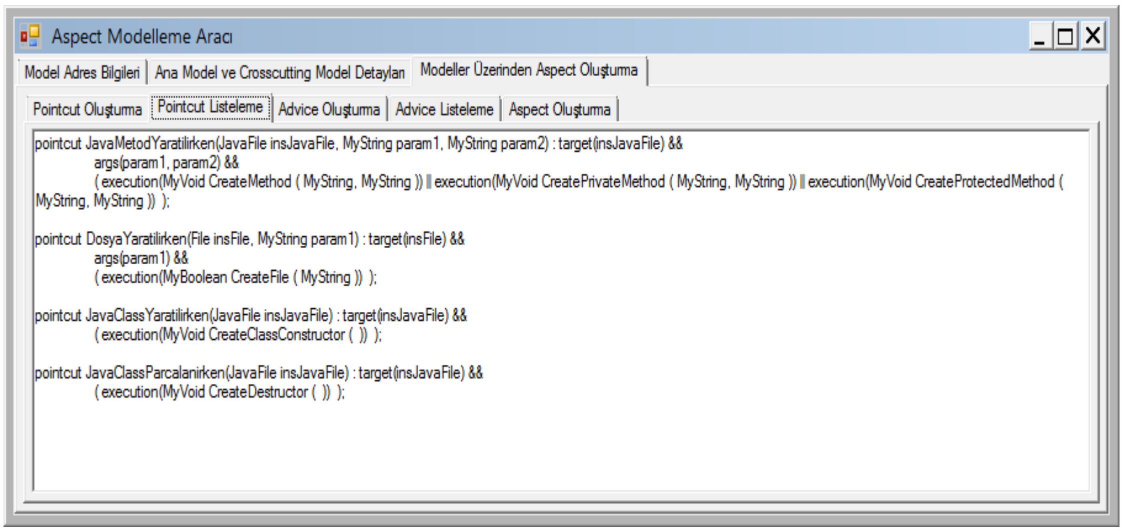
Pointcut model in Figure 19 pick up join points that includes the following points:

- On the execution of CreateMethod with two string parameters and with return type MyVoid.
- On the execution of CreatePrivateMethod with two string parameters and with return type MyVoid.
- On the execution of CreateProtectedMethod with two string parameters and with return type myVoid.

4.4.3 Pointcut List

Listing of modelled pointcuts are made through a seperate interface. This interface allows designers to review the list of pointcuts with the generated code preview. Code generation of pointcuts in the AspectJ language is one of the intelligent parts of AspectModeler. Pointcut code generation is so implemented to use context exposing mechanism in AspectJ. AspectJ has context exposing mechanism via pointcut parameters which correspond to data available when an event happens. The most important rule while using context exposing mechanism, i.e., all pointcut parameters must be bound at every join point picked by the pointcut, is also applied while generating pointcut code in AspectModeler.

AspectModeler previews generated code with suitable line indents to make generated code more readable.



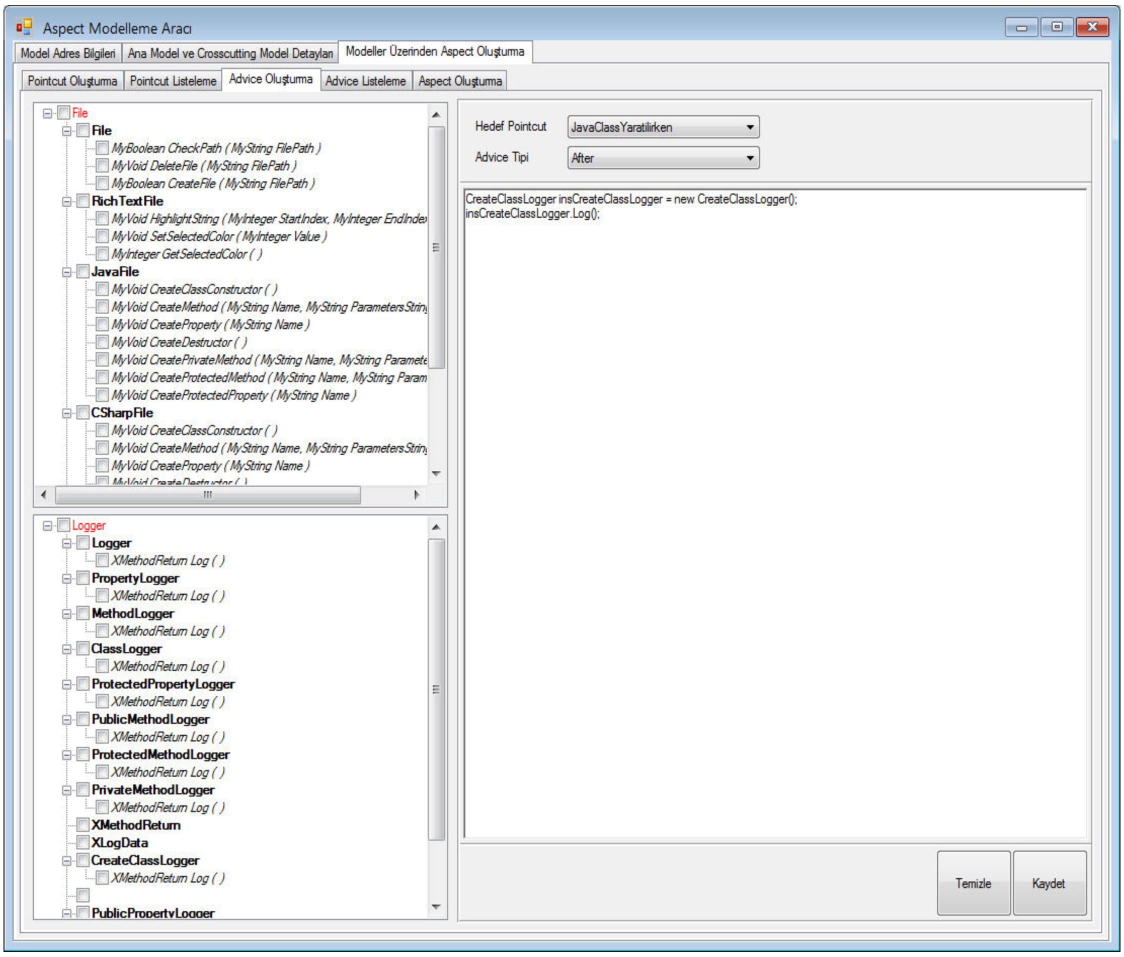
```
pointcut JavaMetodYaratilirken(JavaFile insJavaFile, MyString param1, MyString param2) : target(insJavaFile) &&
            args(param1, param2) &&
            ( execution(MyVoid CreateMethod ( MyString, MyString )) || execution(MyVoid CreatePrivateMethod ( MyString, MyString )) || execution(MyVoid CreateProtectedMethod (
MyString, MyString ))  );

pointcut DosyaYaratilirken(File insFile, MyString param1) : target(insFile) &&
            args(param1) &&
            ( execution(MyBoolean CreateFile ( MyString ))  );

pointcut JavaClassYaratilirken(JavaFile insJavaFile) : target(insJavaFile) &&
            ( execution(MyVoid CreateClassConstructor (  ))  );

pointcut JavaClassParcalanirken(JavaFile insJavaFile) : target(insJavaFile) &&
            ( execution(MyVoid CreateDestructor (  ))  );
```

Figure 4.20 Pointcut list

4.4.4 *Advice Modeling*

Advice modeling is done through a seperate interface in AspectModeler. Two advice types: before and after types are supported at this version of AspectModeler. This interface has also a left panel that aims to make advice modeling task easier. Because advices define pieces of aspect implementation that execute at well-defined points during the execution of a program, left panel contains both base model and aspect model representations in treeviews. Designer can choose easily which method(s) to execute on the log system model according to the pointcut and advice type definition.

An advice has three fields that must be assigned by the designer: pointcut info, advice type and advice implementation. Pointcut info can be selected from a combobox component that contains pointcut list defined in the previous interfaces. Because pointcuts are selected from a predefined set of values, advices in the AspectModeler has named pointcuts. Advice implementation is manually coded by the developer by using the helper model representations on the left side. When clicked on the items in the model representations, code generation is automatically done according to the clicked item's type. For example when a class item is clicked on the left panel, a new object creation code is automatically written into the advice code section.



```
File
  File
    MyBoolean CheckPath ( MyString FilePath )
    MyVoid DeleteFile ( MyString FilePath )
    MyBoolean CreateFile ( MyString FilePath )
  RichTextFile
    MyVoid HighlightString ( MyInteger StartIndex, MyInteger EndIndex
    MyVoid SetSelectedColor ( MyInteger Value )
    MyInteger GetSelectedColor ( )
  JavaFile
    MyVoid CreateClassConstructor ( )
    MyVoid CreateMethod ( MyString Name, MyString ParametersStrin
    MyVoid CreateProperty ( MyString Name )
    MyVoid CreateDestructor ( )
    MyVoid CreatePrivateMethod ( MyString Name, MyString Paramete
    MyVoid CreateProtectedMethod ( MyString Name, MyString Param
    MyVoid CreateProtectedProperty ( MyString Name )
  CSharpFile
    MyVoid CreateClassConstructor ( )
    MyVoid CreateMethod ( MyString Name, MyString ParametersStrin
    MyVoid CreateProperty ( MyString Name )
    MyVoid CreateDestructor ( )
```

```
Logger
  Logger
    XMethodReturn Log ( )
  PropertyLogger
    XMethodReturn Log ( )
  MethodLogger
    XMethodReturn Log ( )
  ClassLogger
    XMethodReturn Log ( )
  ProtectedPropertyLogger
    XMethodReturn Log ( )
  PublicMethodLogger
    XMethodReturn Log ( )
  ProtectedMethodLogger
    XMethodReturn Log ( )
  PrivateMethodLogger
    XMethodReturn Log ( )
  XMethodReturn
  XLogData
  CreateClassLogger
    XMethodReturn Log ( )

  PublicPropertyLogger
```

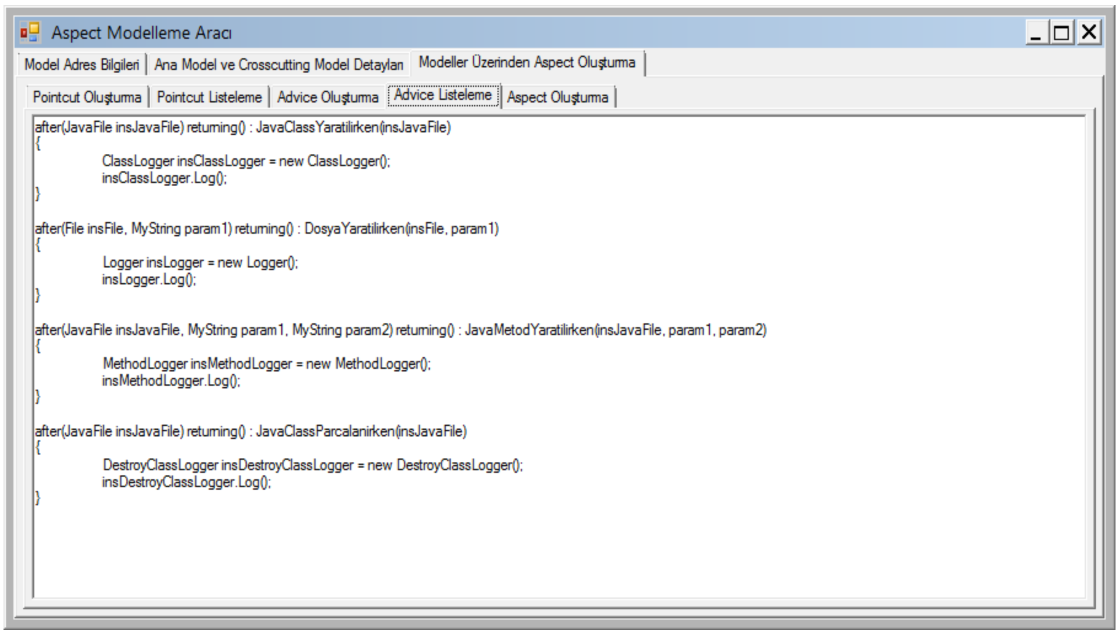
Hedef Pointcut: JavaClassYaratilirken
Advice Tipi: After

```
CreateClassLogger insCreateClassLogger = new CreateClassLogger();
insCreateClassLogger.Log();
```

Figure 4.21 Advice modeling

4.4.5 *Advice List*

Listing of modelled advices are made through a seperate interface. This interface allows designers to review the list of advices with the generated code preview. Code generation of advices in the AspectJ language is also one of the intelligent parts of AspectModeler. AspectModeler allows the generated advices to retrieve information related to the join point (e.g., the arguments of a message). The related information can then be used by the designer while writing advice code.



```
after(JavaFile insJavaFile) returning() : JavaClassYaratilirken(insJavaFile)
{
        ClassLogger insClassLogger = new ClassLogger();
        insClassLogger.Log();
}

after(File insFile, MyString param1) returning() : DosyaYaratilirken(insFile, param1)
{
        Logger insLogger = new Logger();
        insLogger.Log();
}

after(JavaFile insJavaFile, MyString param1, MyString param2) returning() : JavaMetodYaratilirken(insJavaFile, param1, param2)
{
        MethodLogger insMethodLogger = new MethodLogger();
        insMethodLogger.Log();
}

after(JavaFile insJavaFile) returning() : JavaClassParcalanirken(insJavaFile)
{
        DestroyClassLogger insDestroyClassLogger = new DestroyClassLogger();
        insDestroyClassLogger.Log();
}
```

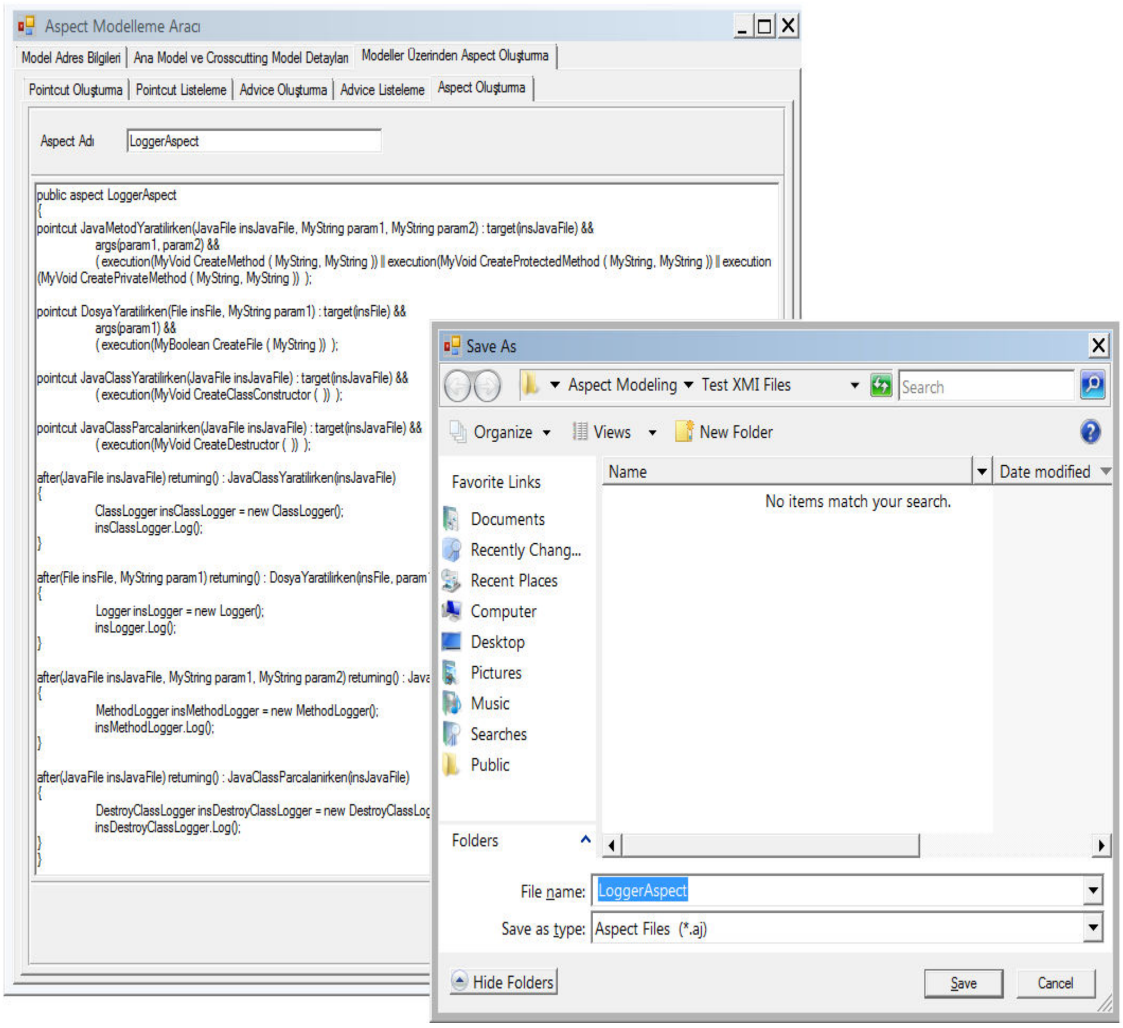
Figure 4.22 Advice list

4.4.6 Aspect Definition

An aspect is constructed by the usage of join points, pointcuts, advices. So that, after completing the steps of pointcut modeling and advice modeling, aspect definition is simply as the definition of aspect name.

To define an aspect, there are three inputs required in AspectModeler. These are pointcut list, advice list and the aspect name. Because the first two inputs have been defined in the previous interfaces in AspectModeler, aspect definition interface only requires aspect name as input.

The generated whole aspect code in aspect definition interface is a valid AspectJ language code. Aspect definition interface also allows the designer to save aspect code as an aspect file (with *.aj* extension) into any directory. By saving the aspect code into the project design environment, aspect modeling process is ended. After saving aspect file into the project design environment and then compiling the project, aspect weaver tool of the AspectJ language automatically weaves the advices into the appropriate locations in the component code (i.e., base code).



```
public aspect LoggerAspect
{
pointcut JavaMetodYaratilirken(JavaFile insJavaFile, MyString param1, MyString param2) : target(insJavaFile) &&
            args(param1, param2) &&
            ( execution(MyVoid CreateMethod ( MyString, MyString )) || execution(MyVoid CreateProtectedMethod ( MyString, MyString )) || execution
(MyVoid CreatePrivateMethod ( MyString, MyString )) );

pointcut DosyaYaratilirken(File insFile, MyString param1) : target(insFile) &&
            args(param1) &&
            ( execution(MyBoolean CreateFile ( MyString )) );

pointcut JavaClassYaratilirken(JavaFile insJavaFile) : target(insJavaFile) &&
            ( execution(MyVoid CreateClassConstructor ( )) );

pointcut JavaClassParcalanirken(JavaFile insJavaFile) : target(insJavaFile) &&
            ( execution(MyVoid CreateDestructor ( )) );

after(JavaFile insJavaFile) returning() : JavaClassYaratilirken(insJavaFile)
{
            ClassLogger insClassLogger = new ClassLogger();
            insClassLogger.Log();
}

after(File insFile, MyString param1) returning() : DosyaYaratilirken(insFile, param
{
            Logger insLogger = new Logger();
            insLogger.Log();
}

after(JavaFile insJavaFile, MyString param1, MyString param2) returning() : Java
{
            MethodLogger insMethodLogger = new MethodLogger();
            insMethodLogger.Log();
}

after(JavaFile insJavaFile) returning() : JavaClassParcalanirken(insJavaFile)
{
            DestroyClassLogger insDestroyClassLogger = new DestroyClassLog
            insDestroyClassLogger.Log();
}
}
```

Figure 4.23 Aspect definition

AspectModeler is a tool that enables modeling of aspects in a practical and efficient way. In addition to modeling aspects, AspectModeler also generates aspect code in AspectJ language that makes it an end-to-end product in aspect modeling process.

Advantages of AspectModeler compared to the other works discussed in the related work sections can be listed as the following:

- AspectModeler eliminates the step of specification of a UML profile for modeling aspects. In AspectModeler approach, base models and aspect models are pure UML models without stereotypes for aspect-oriented constructs. Designers for aspect models don't have to encapsulate their models with aspect-oriented stereotypes, i.e., specification of aspects, advices ...etc. Base model and aspect model designers independently build their models as PIMs and AspectModeler uses these models without any modification to model aspects and includes aspect models into the existing software system by generating aspect oriented programming code in a seperate file.
- AspectModeler eliminates the step of model weaving mechanism. AspectModeler plays a bridge role between base models and aspect models to model aspects. Because aspect oriented programming code in AspectJ is automatically generated as the last step of AspectModeler, there is no need to define an extra process for weaving of base models and aspect models. After saving aspect file, generated at the last step of AspectModeler, into the project design environment and then compiling the project, aspect weaver tool of the AspectJ language automatically weaves the advices into the appropriate locations in the component code (i.e., base code). Because aspect weaver tool for AspectJ is a mature tool, this delegation approach ensures proper completion of model weaving at program code level.
- AspectModeler has a userfriendly interface for modeling pointcuts. Selection of join points can easily be done through the left panel that shows the details of base model. By clicking the model items on the left panel, join points are automatically added to grid component which lists join points of the current pointcut modelled.

- AspectModeler generates aspect oriented programming code in AspectJ language. Becasue AspectJ is the most popular and dominant programming language in AOP environment, the output of the AspectModeler will possibly be valid in a wide range of applications.
- AspectModeler expects model inputs in XMI format which is an OMG standard to represent MOF-based models in XML documents. The input mechanism enforced by AspectModeler makes it a product that can accept inputs generated by different tool vendors who conform the standards of OMG and its XMI and UML foundations specially.
- Aspect weaving mechanism approach in AspectModeler has some benefits when it is subject to maintain aspect behaviour. In AspectModeler approach, because aspect oriented programming code is only stored in aspect files, modification of aspect files is enough to change the aspects behaviour. But in model weaving approach, because all advices in the aspect model are injected into the base model (i.e., single woven model is produced in the model weaving approach), modification of advices cross-cuts the wowen model (i.e., modification job also becomes a cross-cutting concern). There are two alternatives in model weaving approach; one is to modify the aspect model and doing all the model transformations again to produce a single woven model, the other is to modify the woven model which is a laborious task.

Disadvantages of AspectModeler compared to the other works, discussed in the related work sections, can be listed as the following:

- AspectModeler generates aspect oriented programming code only in AspectJ language. This selection is a valid selection in today's AOP environment, because AspectJ is the most popular and dominant language. Unfortunately, this selection may not be valid with the birth of a more popular and dominant aspect-oriented programming language. In this situation, AspectModeler may have to support also this new AOP-based programming language.

- At this version of AspectModeler, we only support the representation of class diagrams stored in XMI files. Other works have also supports on activity diagrams, sequence diagrams ... etc. Representation of other diagram types in UML should be supported not to hinder the Aspect Oriented Modeling approach.
- During pointcut modeling, AspectModeler only allows the individual selection of joinpoints. If there are some join points that have the same pattern, designers have to select join points manually which may be a time-consuming task. Selection of join points with extra mechanisms (e.g., usage of wildcards) should be supported to overhelm this issue.

CHAPTER FIVE
CONCLUSION & FUTURE WORK

5.1 Conclusion

Model Driven Architecture (MDA) is an approach to software development to use visual models as a single resource for software developers. To achieve this goal, MDA process defines some abstraction levels for the clear seperation of concerns in the vertical dimension which are Computation Independent Model (CIM), Platform Independent Model (PIM) and Platform Specific Model (PSM).

However, the clear seperation of concerns in the horizontal level is not addressed in MDA approach. MDA approach lacks mechanisms for identifying and seperating cross-cutting concerns.

Aspect Oriented Programming (AOP) approach complements Object Oriented Programming(OOP) which may not be enough to clearly capture some programming problems. However, techniques used in the context of AOP concantrate in the system implementation phase (i.e code level). Seperation of cross-cutting concerns (horizontal level) at the modeling level is being tackled in the area of Aspect Oriented Modeling (AOM).

There are related works on Aspect Oriented Modeling that address the issue of seperation of cross-cutting concerns at the modeling level. Because UML does not support Aspect Oriented Modeling in its standard specification, solutions from different tool vendors are different. As a result, modeling of aspects is still in process and modeling tools still lack stability and maturity to deal with aspect models.

We have designed and developed an aspect modeler tool named AspectModeler that allows the definition of aspects in a practical and efficient way. This tool eliminates some steps that are necessary in other approaches. AspectModeler has a number of interfaces; base model and aspect model representation, pointcut modeling, pointcut list, advice modeling, advice list, aspect definition where each of the interfaces plays a critical role to make aspect modeling easier. The output of AspectModeler is an aspect file that is a valid AspectJ language code. To run the existing software system with modelled aspects is simply as putting the generated aspect file into the existing project's design environment.

5.2 Future Work

We have developed a tool named AspectModeler that enables modeling of aspects in a practical and efficient way. There are some disavantages of AspectModeler as well as it has some advantages compared to the other works. Disadvantages of AspectModeler can also be represented as future works to be developed to facilitate the full adoption of Aspect Oriented Modeling.

AspectModeler generates aspect oriented programming code only in AspectJ language in this version. With the birth of a more popular and dominant aspect oriented programming language, this new AOP-based programming language may have to supported by AspectModeler.

Only the representation of class diagrams stored in XMI files is supported in this version. Representation of other UML diagram types such as activity diagram, sequence diagram ... etc should be supported in future releases.

Pointcut modeling mechanism should be improved in future releases. Selection of join points with same pattern is done by individually selecting these points.

AspectModeler should make it possible to use wildcards to write joints (i.e., representation of more that one join point in a single expression).

Advice body is modelled at programming language level in this version. In future releases, AspectModeler should make it possible to model advice body with UML diagrams such as: activity diagram, sequence diagram ...etc.

REFERENCES

Bell, D. (June 15, 2003). *UML basics: An introduction to the unified modeling language.* Retrieved April 10, 2009, from http://www.ibm.com/developerworks/rational/library/769.html

Duby, C. K. (September, 2003). *Accelerating embedded software development with a model driven architecture*. Retrieved May 11, 2010, from http://www.omg.org/mda/mda_files/MDA_overview.pdf

Frankel, D. S. (2003). *Applying MDA to enterprise computing.* Indiana: Wiley Publishing.

Fuentes, L., & Sanchez, P. (2007). Designing and weaving aspect-oriented executable UML models. *Journal Of Object Technology, 6* (7), 109-136.

Gally, M. (May, 2007). *What is MDD / MDA and where will it lead the software development in the feature?*. Retrieved December 3, 2009, from http://seal.ifi.uzh.ch/fileadmin/User_Filemount/Vorlesungs_Folien/Seminar_SE/SS07/SemSE07-Matthias_Gally.pdf

Igor, S., & Jadranka, V. (n.d.). *Model driven architecture (MDA)*. Retrieved August 8, 2009,from http://www.softwareresearch.net/fileadmin/src/docs/teaching/SS07/SaI/Salevski_Veseli_Praesentation.pdf

Igor, S., & Jadranka, V. (June, 2007). *Introduction to model driven architecture (MDA)*. Retrieved September 12, 2009, from http://www.softwareresearch.net/fileadmin/src/docs/teaching/SS07/SaI/Salevski_Veseli_paper.pdf

Kiczales, G., Irwin, J., Lamping, J., Loingtier, J., Lopes, C. V., Maeda, C., et al. (n.d.). *Aspect-oriented programming*. Retrieved October 10, 2009, from http://fsl.cs.uiuc.edu/images/9/9c/Kiczales97aspectoriented.pdf

Mouheb, D., Talhi, C., Lima, V., Debbabi, M., Wang, L., & Pourzandi, M. (March 2, 2009). *Weaving security aspects into UML 2.0 design models.* Retrieved March 31, 2010, from http://www.aspect-modeling.org/aosd09/papers/aom5s-mouheb.pdf

Sims, O. (2002). *MDA – real value*. Retrieved June 10, 2009, from http://www.omg.org/mda/mda_files/OMG-Information-Day-Sims_01-01.pdf

Watson, A. (n.d.). *Visual modelling: past, present and future*. Retrieved December 17, 2008, from http://www.uml.org/Visual_Modeling.pdf