CHAPTER 1

INTRODUCTION

1.0 Introduction

Software Product Line approach is a development methodology which approves the ability of increase the productivity and reduce time and costs of developing products. The main idea of SPL is the rapid development of systems member by using reusable assets from all phases of the development life cycle (Klaus Pohl, 2005). Several methods have been developed for SPL such as KobrA, FAST, FODA, and PuLSE. KobrA method is considered as the most practical method compared to others (Michalis, 2002).

KobrA was created at Fraunhofer Institute for Experimental Software Engineering (IESE) at the beginning of this decade. Its development methodology combines the SPL with Component Based Development. It use component concept to drive the developments in all phases of the software life-cycle, so components aren't just as an executable modules implement through specific construct such as JavaBeans (Colin Atkinson, 2001). It uses Unified Modeling Language (UML) to describe components at conceptual level.

KobrA design to be suitable for both single system and family of systems, it divided into two phases - the framework and application phase. The former one provides a generic description of the software elements which makes up a family of applications (reference architecture), it involves all the variable features of applications. But the later one uses the framework repeatedly to build up single product from that family of products. So KobrA goal is to develop applications containing specific variants corresponding to particular customers' requirements. Modeling variability is a key to the former. In this research we only concentrate on framework phase.

Model driven architecture is a new approach for software development is introduced by Object Management Group few years ago (OMG, 2005). In MDA the software development is driven by constructing models in all phases of the development life cycle. The main modeling language of MDA is Unified Modeling Language (UML) and its subset like MOF (OMG, 2008).

It is a new way to design applications. The purpose of this approach is to separate the logic description of system, from any technical platforms. Indeed, the technical platform is going through many changes over time, unlike the logical description, therefore the idea of separating the two of them will make it easier developing systems with less costs to migrate into new technology, MDA capture this separation by developing two models: the Platform independent model (PIM) and platform specific model (PSM).

MDA automatically enable the transformation between models using OMG's Query/View/Transformation (QVT) tool. It reduce the time and cost of developing software through reusing PIM, PSM or QVT rules.

1.1 Problem Statement

Despite the advantages of using KobrA method in developing a family of products in some domain but without automation support the effort and time of development being consumed, it would be difficult to build framework for a family of product. In addition, facing the continuous change in platforms and embraces of new ones would be a challenge.

1.2 Objectives

The objective of this research is basically to reengineering KobrA using MDA which reduces human intervention. The specific goals are:

• Understanding the difference between MDA and KorbA.

• Find the corresponding KobrA artifacts to PIM and PSM in MDA.

• Develop Metamodels including mapping rules.

1.3 Thesis And Outline

This thesis is divided into four chapters:

1. Introduction:
    - It contains a simplified introduction about Software Product Line , KobrA and Model Driven Architecture. Motivation for thesis : KobrA weaknesses.

2. Literature review:
    - Complete description about what is KobrA and MDA. The different between MDA and KobrA
3. Reengineering KobrA using MDA:
    - Reengineering steps to enhanced KobrA using MDA tool. The different between MDA and KobrA.
4. Conclusion:
    - The thesis results toward reengineering KobrA using MDA , which enhanced KobrA method.

CHAPTER 2

LITERATURE REVIEW

2.0 Introduction

In this chapter we present a literature review that spans areas related to this research: KobrA method, Model Driven Architecture approach and preview for Software Product Line methodology.

2.1 Software Product Line

Software Product Line Development is a development methodology that focuses on high-level reuse of large software pieces. In contrast to other methodology it produces family of products, all products would be result of integration rather than creation.

"*A software product line is a set of software- intensive system sharing a common managed set of features that satisfy the specific needs of a particular market segment or mission and that are developed from a common set of core assets in prescribed way* "(Klaus Pohl, 2005).

That's mean SPL develops core assets which contains the commons and variables features of systems, it used to produce family of systems instead of developing them from scratch. SPL goals is to increase the productivity (the core assets are reused), quality (those reused are verified and tested) and decrease time to market (Klaus Pohl, 2005).

2.1.1 Software Product Line Methods

Several methods have been established for product-line engineering such as feature-oriented domain analysis (FODA), Product Line UML-based Software Engineering (PLUS) and KobrA. But they are supporting family of systems from high level of abstraction without concrete guidance on how they effectively apply in practices (Michalis, 2002). Therefore the practitioners will face challenges and obstacle while they are applying them and limit their use in practice. Except KobrA method is considered as the most practical and concrete method compared to others (Michalis, 2002).

2.2 KobrA Method

KobrA stands for "Komponentenbasierte Anwendungsentwicklung" that is German which means "component based application development". It develops in BMBF- supported KobrA project by Softlab GmbH, Psipenta GmbH, GMDFIRST and Fraunhofer IESE (Colin Atkinson, 2000).

KobrA approach is combination of two reuse concepts, the reuse in small concept in component based approach and reuse in large concept in software product line methodology (Colin Atkinson, 2002).

Atkinson et al. claim that "*the product-line and component-based approaches to software development seem to have complementary strengths. They both represent powerful techniques to support reuse, but essentially at the opposite ends of the granularity spectrum*".

In KobrA architecture the high level description of component separated from implementation technology (Colin Atkinson, 2001).

KobrA components (komponents) aren’t physical but rather logical , they have properties of class and module of Unified Modeling Language (UML) which means they represent their own behavior like class and act as containers for other components like module (Colin Atkinson , 2002).

2.2.1 KobrA Activities

The KobrA method fundamentally has two major phases which are the framework and application engineering. The first phase provide generic reusable framework with contain common and variable features of products family but in second phase the products are initiated from framework. KobrA phases aim to develop applications corresponding to particular customers’ requirements (Colin Atkinson, 2002). In this research we only concentrate on framework phase or it’s also called in literature domain engineering. Variability modeling is essential to this phase.

2.2.2 KobrA Framework

KobrA framework contains set of komponents organized in a form of tree, the identification of variabilities define along with creation of komponents. The variabilities are features vary from product to another, they represent using UML stereotype and decisions models which contain the relation between the variable features (Colin Atkinson, 2001). Each komponents are described at two levels of abstraction, the specification and realization level through interrelated suit of UML diagrams. So the overall framework will be a set of komponent specifications and realizations.

2.2.2.1 Komponent Specification

The specification level describes visible characteristics and behavior of komponents. It contains information intended to be externally visible to other komponents. It defines the interface of komponent with list of operations that it supports, but also with additional behavioral and structural information. These are described by using four models as it is shown in Figure 2.1 (Colin Atkinson, 2001):

1. Structure model: include (i) class diagram and (ii) object diagram. (i) Define the classes, operations, attribute and the relationship between classes. It contains simple, komponent and subject class (komponent under specification). (ii) Instances of class and it show how they link together.

2. Functional models are textual description for komponent operations. It contains description for effect of execute operations.

3. Behavior model use state diagram to captures the dynamic behaviors of komponent

4. Decision models are textural model contain information about variation between komponent models. Each diagram has decision model because they all contain variability, the decision models organize in hierarchy and the result of one decision will effects the other decisions.

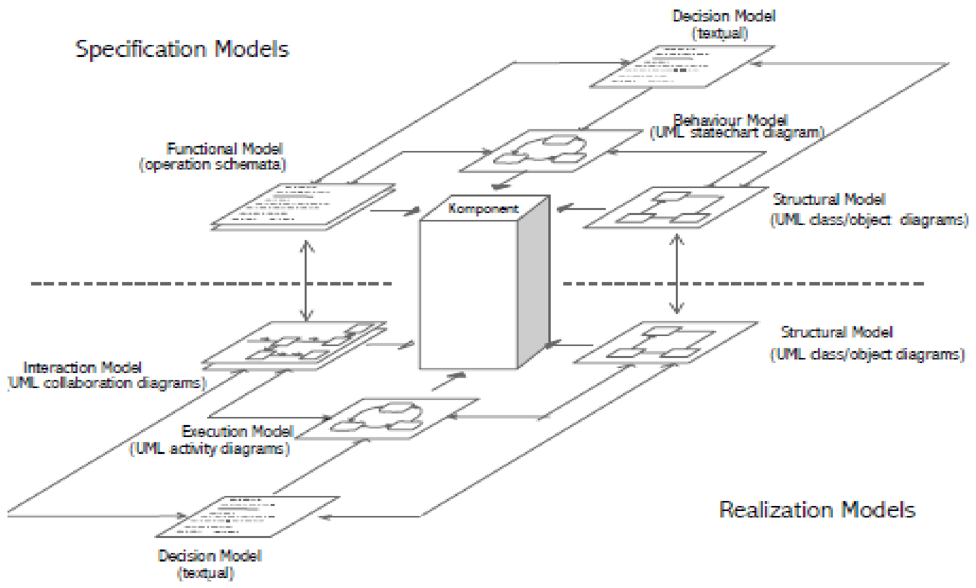

Figure 2.1 UML-based Component Modeling (Colin Atkinson, 2001)

2.2.2.2 Komponent Realization

Realization level describes internal structure of komponents (private design), how it makes use of other komponents and what internal data structure it uses. These are described by using four models as it is shown in Figure 2.1 (Colin Atkinson, 2001):

1. Structure model: include (i) class diagram and (ii) object diagram. (i) Define the classes, operations, attribute and the relationship between classes. It contains simple, komponent and subject class (komponent under specification). (ii) Instances of class and it show how they link together.

2. Interaction model use collaboration diagrams to describe how operations of komponent are realized in term of interaction.

3. Activity models used to describe the algorithms used to realize the operations of the komponent

4. Decision models are textural model contain information about variation between komponent models. Each diagram has decision model because they all contain variability, the decision models organize in hierarchy and the result of one decision will effects the other decisions.

2.2.3 Komponent Modeling

The models that used in KobrA are based on some principles and guidance, in addition KobrA develops specific formalism for model komponents.

2.2.3.1 Modeling Principles

KobrA model komponents based on four principles, to make sure that description of komponents is relatively explicit and systematic (Colin Atkinson, 2002):

1. Uniformity: all komponents in tree model using same set of UML models as describe before, therefore every komponent can be treated as system in its own right or can be used again with other system by that it encourage the reuse concept.

2. Parsimony: this principle emphasized that each diagrams that describe single komponent should contain only needed information.

3. Locality: the locality principle means that no komponent has comprehensive view to all komponent in containment tree.
4. Encapsulation: Separate the specification of komponent from realization. Specification which describes what is komponent do and the realization describes how komponent realize its specification.

2.2.3.2 Model Formalism

KobrA models based on UML but it depends on its own formalism to model komponents using UML stereotype. The komponents tagged with stereotype <<Komponent>>, the subject tagged with stereotype <<Subject>> and the variability tagged with stereotype <<variant>> (Joachim Bayer, 2001).

The variant stereotypes are applied in class diagram for komponent and simple classes, in functional models in description of komponent operation, in state diagram in its internal activity and in message of collaboration and sequences diagram, in addition the relation between all variant elements are defined textual form in the decision models, therefore KobrA provides guidance for developer to choice the variable entities during framework specialization phase (Colin Atkinson, 2002).

2.2.4 Containment Tree

The framework tree known as containment and it created by the recursive developments process of nested komponents of realizations and specifications. Having tree structure enable avoid the repetition between models in which the parent / child relationship represents (a parent is composed of its children) (Colin Atkinson, 2002). That’s mean one komponent can be a part of another komponent and one big komponent can include other many small komponents (Colin Atkinson, 2000).

As example taken from Library system the ReservationManager komponent identified during the realization of LoanManager komponent, and once the LoanManager realization complete its sub komponent such as ReservationManager can be model (Joachim Bayer, 2001).

Butting komponents in tree require several fundamental principles and guidance to drive this process.

2.2.4.1 Consistency Rules

There’re six consistency rules must be satisfied to make sure that komponents containment tree is well formed and consistent, as it show in Figure 2.2 (Colin Atkinson, 2002):

1. Intra-diagram rules: ensure that all individual diagrams are well formed.

2. Inter-diagram rules: the diagrams within a specification or a realization are consistent with each other.

3. Realization rules: The realizations komponents must be correct representation of its specification.

4. Specialization rules: ensure that a specialized component conforms to the component from which it was specialized.

5. Containment rules: ensure that a component's relationships with other components are consistent with its location in the containment tree.

6. Clientship rules (contract): ensure that a client and server both fulfill their contract.

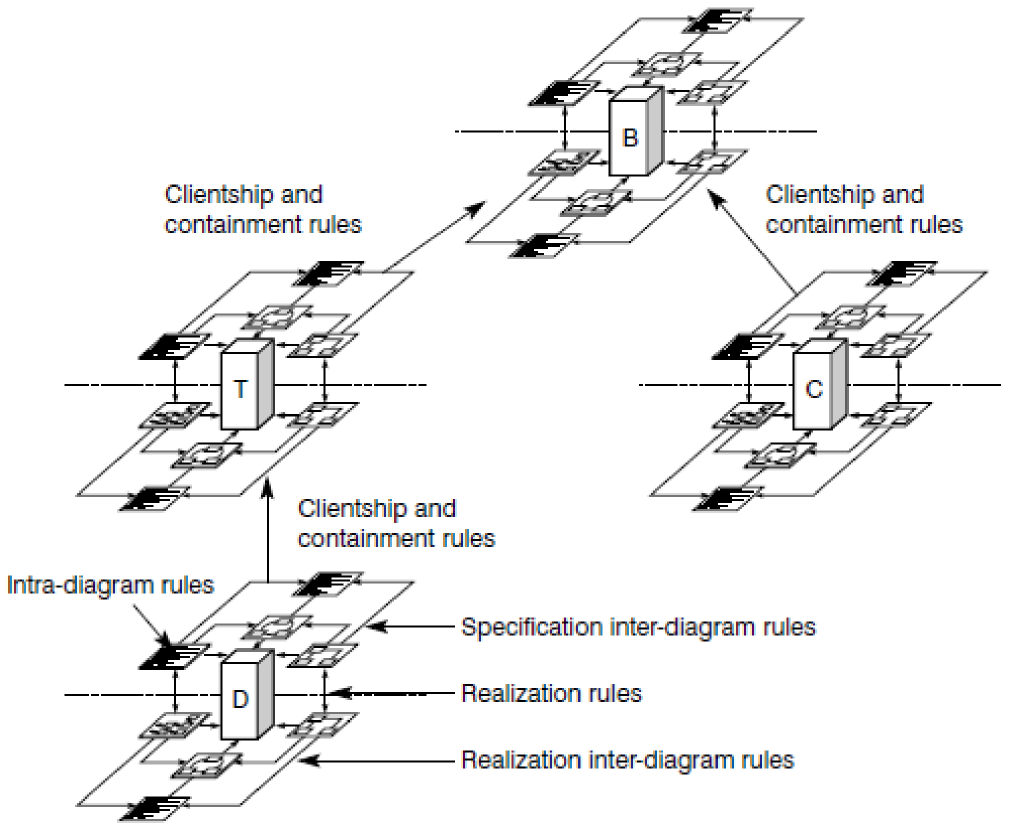
B
Clientship and containment rules
Clientship and containment rules
T
C
Clientship and containment rules
Intra-diagram rules
Specification inter-diagram rules
D
Realization rules
Realization inter-diagram rules

Figure 2.2 Consistency Rules (Colin Atkinson, 2002)

2.2.4.2 Visibility Rules

What component can see in tree are introduce by visibility rules which based on komponent position in the containment hierarchy, they originally adapted from UML package visibility rules.

First rule: komponent can see another komponent only if its immediate child, Second rule: komponent can see another komponent only if its immediate parent, Third rule: komponent can see another komponent if both share same parent, Fourth rule: komponents can see up and across from its location in the tree (Colin Atkinson, 2002).

2.2.5 KobrA Property

KobrA method consider systematic method, it provide the developer with set of precise and unambiguous guidance. The consistency rules give concrete guidance as to which models should be used and what the necessary information they should contain (Colin Atkinson, 2002).

KobrA separation of concern principles is identified as being central aspect for developing complex systems, it make use of three basic principles (Colin Atkinson, 2000):

1. Separate the component description from implementation make it compatible with many of practical implementation and middleware technologies.
2. Separate the description of what komponent do (specification) from how do it (realization).
3. Separation of process and products, what products should be built from process, KobrA representation of system separately from activities and guidelines used to create and maintain them.

2.3 Model Driven Architecture (MDA)

Model driven architecture is new approach for software development is introduced by Object Management Group (OMG, 2003) in few years. The central idea of MDA is to use models to drive the development in all phases of software lifecycle. It raises a slogan “Design once and build it on many”.

2.3.1 MDA Structure

The MDA approach emphasis on two kinds of models with respect to specific platforms: the Platform Independent Models (PIM) and the Platform Specific Models (PSM).

2.3.1.1 Platform Independent Model (PIM)

The Platform Independent Model is high level abstraction model, it represent the business functionality and behavior excluding the platform specific details (Anneke Klepper, 2003). The PIM define by The Unified Modeling Language.

The Platform Independent Models provides two basic advantages (MDA, 2001):

1. The developer responsible for defining business functionality without any platform detail, make it easy validate correctness of model. The PIM keep intact.

2. Since the functionality is extract from any platform details, so it’s easy to produce implementation on different platforms.

2.3.1.2 Platform Specific Model (PSM)

The platform Specific Model (PSM) represent the specification of platform, it specifies how system functionality brought to specific platform and produce as result of transforming PIM (MDA, 2001).

The PSM describe in one of two ways (MDA, 2001):

1. Using UML diagrams such as class and sequence diagram.
2. Interface definition in a concrete implementation technology (e.g. XML , Java).

2.3.2 Metamodel

Metamodel have important role in MDA, the metamodel is model for describe model, in other words the metamodel is used to define language for expressing model at high level of abstraction than modeling language itself (Anneke Klepper, 2003). As in natural language, all languages have grammars that describe structure of language. The programming languages have metamodel called Backus–Naur Form (BNF), to describe right syntax.

MDA based on metamodeling language Meta Object Facility (MOF), which is used to define the Unified modeling language. All model of UML fundamentally based on MOF.

2.3.3 MDA- Development Process

MDA development process is carried out through appropriate transformation or mapping between models. The transformation is process of converting one model to another model (OMG, 2003).

The Query View Transform (QVT) is standard language for specifying model transformation in the MDA, It introduce by OMG with collection of transformation rules to illustrate the model elements mapping (MOF, 2008).

There are four kinds for model transformation (MDA, 2001):

1. PIM to PIM: In this transformation the models are enhanced, filtered without any platforms information, therefore the transformation for model refinement.

2. PIM to PSM: In this transformation the PIM is refined to be expected to execute on specific platform

3. PSM to PSM: This kind of transformation need for component realization and deployment, which relate to platform model refinement.

4. PSM to PIM: This transformation is often used for abstracting models of existing implementations into platform independent models, and the result of transformation would be same as PIM to PSM transformation.

2.3.4 MDA Benefits

MDA develops with goals of increase productivity , portability , cross-platforms interoperability by separating system abstract architecture from platform concrete architecture and it ability to implement abstract architecture into different platform automatically by that it reduce time and cost of development (OMG,2003).

2.4 MDA And KobrA

KobrA provides foundation for automation as it uses UML notation to model the components (Colin Atkinson, 2002). UML notation can be used with any developmental practice, thus MDA development process would be suitable to incorporate with KobrA approach as it concentrate on high level specification .

According to description of komponents, KobrA separate the component specification (interface) from realization (design) where it allows replacing one component with another by keeping the interface and replace design of component. Depending on model driven architecture principles this means component specification capture at level of platform independent model, and the realization at level of platform specific model.

Mapping between artifacts is more systematic in MDA has been automated which is manually and not a systematic in KorbA.

CHAPTER 3

REENGINEERING KOBRA USING MDA

3.0 Introduction

In this chapter we aim to show how to reduce human intervention in process of developing family of product in some domain by using MDA automation facility. MDA automated process is carried out through transformation between PIM and PSM using QVT specification , this can be achieved by building metamodels of each source and target model, then defining a mapping between them (OMG,2003). The library system would be used as case study. It is represented by Fraunhofer Institute for Experimental Software Engineering (IESE) (Joachim Bayer, 2001) to illustrate basic KorbA concepts. The library framework used to initiates a family of system such as national and academic libraries with different features.

3.1 Reengineering KobrA

KobrA framework is the representation of a set of komponents, in order for komponents to be established, KobrA define two different tasks: the specification (interface) and realization (design). Both tasks represented by using UML models (Colin Atkinson, 2001).

But most of the work in KobrA requires the human interfere, therefore the quality and cost of development consumed especially in phasing continuous change in platforms and embraces of new ones.

The MDA approach is basically depends on using models in all phase of software development, the basic step in MDA is to separate the logic description of system from any technical platforms, allowing the same logical description to be implemented automatically with different platform technologies (OMG, 2005). So using MDA would:

- Decrease the development effort: especially in making platform specific code.
- Decrease the cost of development: the system specification is designed once and implemented with different platforms (redeployment).
- Increase the quality: the faults and errors are reduced (less human intervention).

Therefore using MDA to reengineering KobrA would present results that would solve KobrA insufficiency issue. KobrA method is considered as good candidate to be reengineering using MDA for two reasons:

- KobrA's ability to separate komponent implementation from abstract description makes it easy for MDA to be used with.
- KobrA basically depends on using UML models to describe its core development artifacts (komponents).

3.2 Reengineering Steps

MDA engineering will be used for library framework which is developed by KobrA, which would be obtained by following a number of steps which are as follow. MDA steps help classifying KobrA artifacts according to it.

- Step one: PIMs and PSM.
- Step two: Developing Metamodels.
- Step three: Mapping rules.
- Step four: Automating transformation.

3.2.1 PIM And PSM

According to KobrA concepts, the komponents are the main component of the framework. Each komponent is described with two levels of abstraction: the *specification* and *realization* level using UML models (Colin Atkinson, 2001).

The specification describes what komponent should do and what the visible properties are, it also represents the requirements needed to meet business objective. On the other hand the realization describes how to accomplish the requirements defined in specification and represents the private design of komponent. So depending on model driven architecture principles, specification models are considered as PIM and the realization models as PSM.

3.2.2 PIMs Metamodels

We have developed metamodels for komponent specification which include, class, state and class of realization which all together represent PIMs. The reason of considering the class realization as PIM metamodel because it is refinement of class specification metamodel.

3.2.2.1 Class Metamodel

This meta-model extends the UML class meta-model (OMG, 2007). To model variability, isVariant attribute is used in metamodel elements as Boolean value which determines the variability or non variability of elements (see figure 3.1). As in figure 3.1 Class metamodel has seven elements: class, simple, komponent, subject, property, operation and association.

The abstract class is specialized into simple, komponent and subject. Komponent and simple instances contain name and isVariant attributes. Whereas subject instances contain name attribute, as shown in figure 3.1.

In general the class is made up of set of properties and operations. The abstract class has association with operation and property instances. The operation instances have name, id and isVariant attributes. But the property instances have isComposite , name and id attributes , as shown in figure 3.1. IsComposite indicates whether the association end composite or not.

It is important to point out that the property represents the attribute of class and association end of association, therefore abstract association has association with property instances, as shown in figure 3.1.

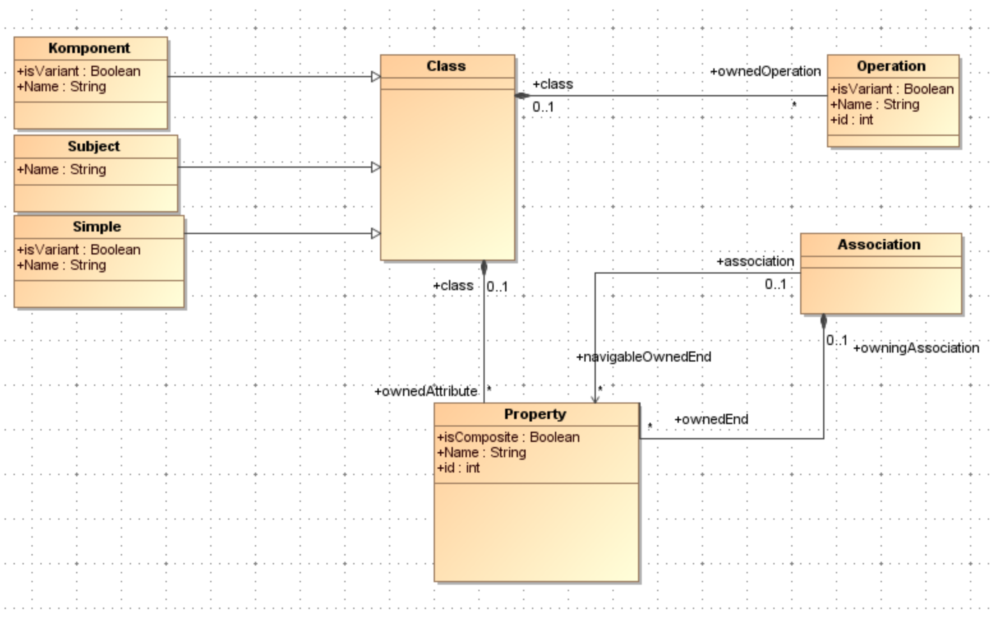

Figure 3.1 Class Specification Metamodel

3.2.2.2 State Metamodel

This meta-model extends the UML state meta-model (OMG, 2007). To model variability, isVariant attribute is used in metamodel elements as Boolean value which determines the variability or non variability of elements (see figure 3.2). As in figure 3.2 State metamodel has five elements: stateMachine, state, internalActivity, transition and transitionString.

The State machine consists of a set of states and transitions. StateMachine is abstract class has association with state instances. State instances contain isSimple and name attributes, it is associated with internalActivity instances which contain name and is Variant attributes, as shown in figure 3.2.

The Transition connects a source and a target state. Transition instances have kind attribute. Furthermore, transition instances associated to transitionString instances which have name attribute, as shown in figure 3.2.

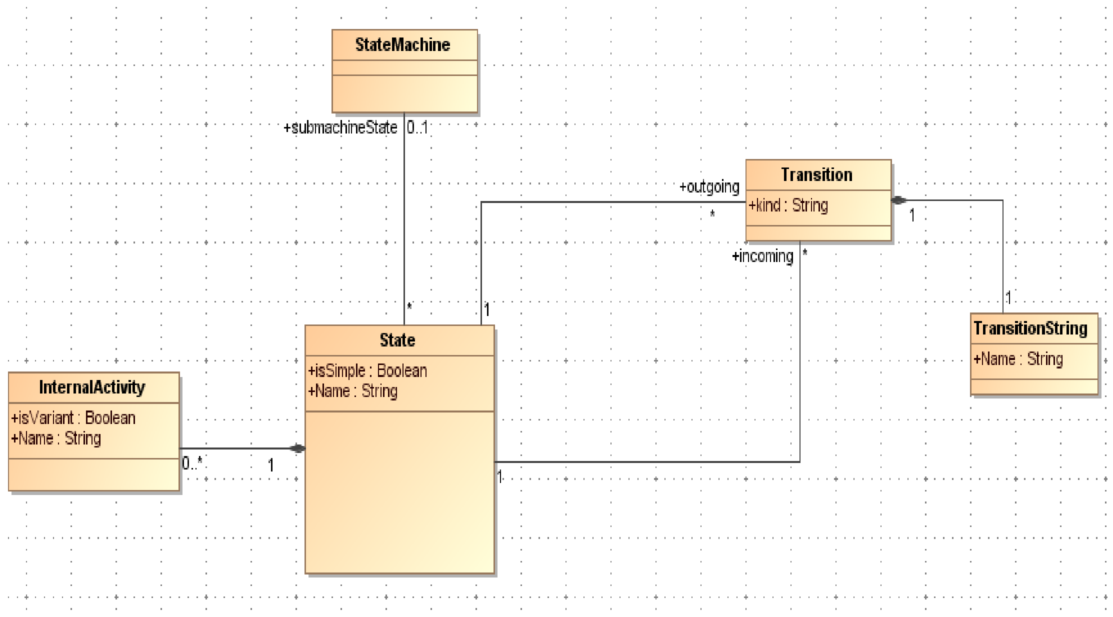

Figure 3.2 State Metamodel

3.2.2.3 Class Realization Metamodel

This meta-model extends the UML class meta-model (OMG, 2007). To model variability, isVariant attribute is used in metamodel elements as Boolean value which determines the variability or non variability of elements (see figure 3.3). As in figure 3.3 Class metamodel has seven elements: classPIM, simplePIM, komponentPIM, subjectPIM, propertyPIM, operationPIM and associationPIM .

The abstract class is specialized into simplePIM, komponentPIM and subjectPIM. KomponentPIM and SimplePIM instances contain name and isVariant attributes. Whereas SubjectPIM instances contains name attribute, as shown in figure 3.3.

In general the class is made up of set of properties and operations. The abstract class has association with operationPIM and propertyPIM instances. The operationPIM instances have name, id and isVariant attributes. But the propertyPIM instances have isComposite , name and id attributes, as shown in figure 3.3. IsComposite indicating whether the association end composite or not.

It is important to point out that the propertyPIM that represent the attribute of class and association end of abstract associationPIM , therefore abstract associationPIM has association with propertyPIM instances, as shown in figure 3.3 .

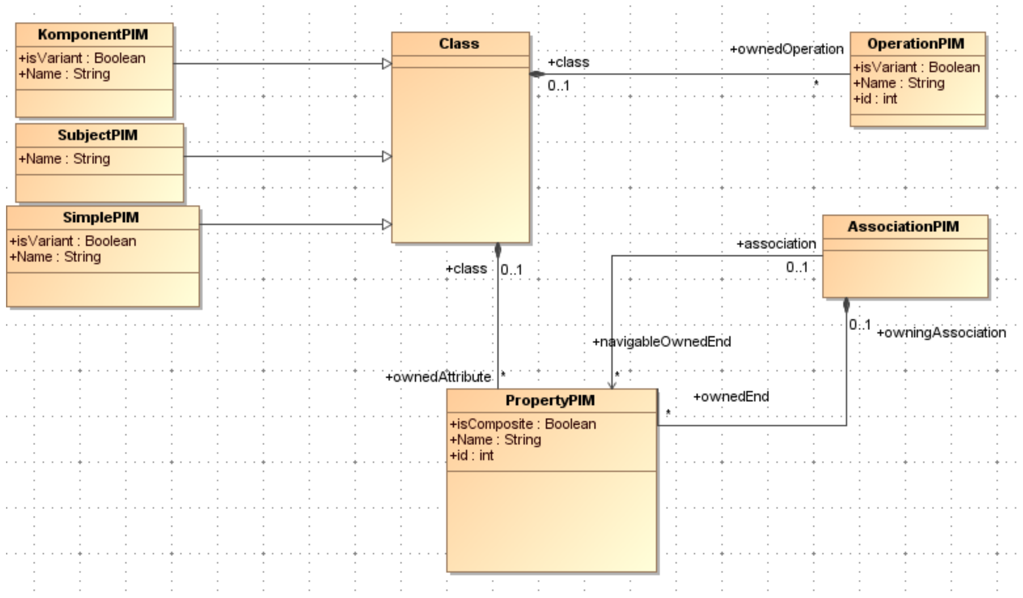

Figure 3.3 Class Realization Metamodel

3.2.3 PSM Metamodel

The PSM metamodel we develop for Activity model of komponent realization. This meta-model extends the UML activity meta-model (OMG, 2007). To model variability, is Variant attribute is used in metamodel elements as Boolean value that determine the variability or non variability of elements (see figure 3.4). As in figure 3.4 Activity metamodel has nine elements: activityNode, initialNode, activityFinalNode, decisionNode, activity, activityPartition, controlFlow, activityEdge and constraint.

The abstract activityNode is specialized into initialNode, activityFinalNode and decisionNode. All classes mention above are abstract classes, as shown in figure 3.4.

The activityNode has association with activity and activityPartition instances. Activity instances have name and isVariant attributes. Whereas activityPartition instances have isVariant and name attributes, as shown in figure 3.4.

The target and source node of activityNode are link by controlFlow. The abstract activityNode has association with abstract ActivityEdge which is specialized into controlFlow. The controlFlow instances have isVariant attribute. ControlFlow is associated with abstract constraint class, as shown in figure 3.4.

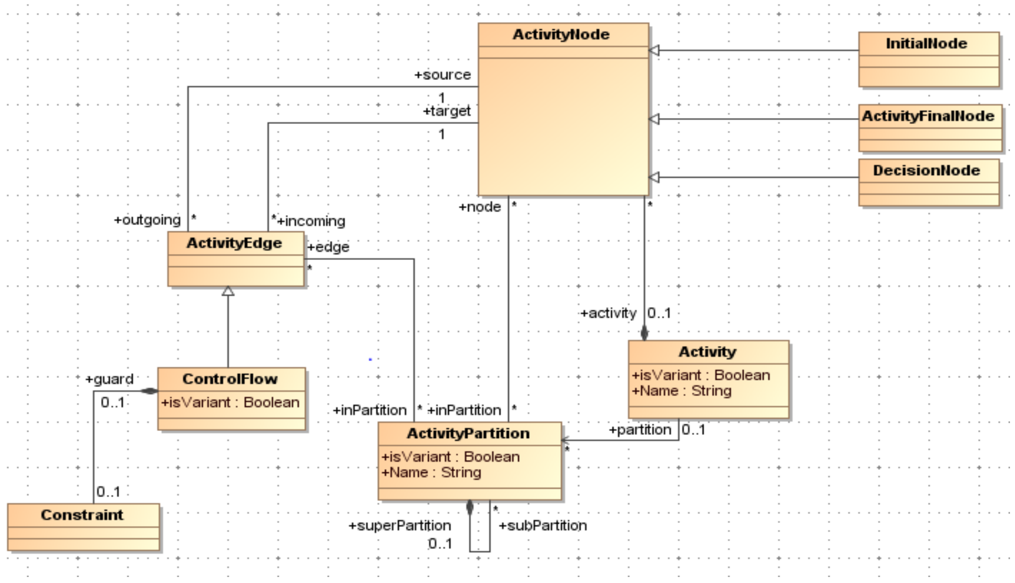

Figure 3.4 Activity Metamodel

3.2.4 Mapping Rules

This section explains the rule of mapping need to be done among models for library framework as an example of software product line need to be developed using KorbA. The model transformation is a process of converting a model expressed by one metamodel to another model which expressed using different metamodel. The transformation is done by mapping (OMG, 2003) using standard language also developed by OMG.

The mapping contains set of rules which specify which model elements should map to another models element (OMG, 2003). At first we need to illustrate metamodels mapping .The library meta-models mapping divided into two parts: the structural meta-models and behavior meta-models mapping. Note that the mapping according to QVT is at metamodels level where instances of the source is essential.

In structural meta-models mapping, the class metamodel (PIM) is mapped to class Realization metamodel (PIM), therefore the mapping is horizontal because both metamodels describe komponent at the same level of abstraction (Frank Truyen, 2006). Table 3.1 shows the structural metamodel elements mapping.

Elements in class metamodel (PIM)	**Corresponding elements in class realization metamodel (PIIM)**
Subject	SubjectPIM
Subject Operations	SubjectPIM Operation
Variant Subject Operation	Variant SubjectPIM Operation
Komponent	KomponentPIM
Komponent Operation	KomponentPIM Operation
Simple	SimplePIM
Variant Simple	Variant SimplePIM
Simple attributes	SimplePIM attributes

Table 3.1 Structural mapping rules

In the behavior meta-models mapping, the state metamodel (PIM) is mapped to activity metamodel (PSM), therefore the mapping is vertical because both metamodels describe komponent at different level of abstraction (Frank Truyen, 2006). Table 3.2 shows the behavior metamodel elements mapping.

Elements in state metamodel (PIM)	Corresponding elements in activity metamodel (PSM)
Internal state activity	Activity
Variant InternalActivity	Variant Activity
Transition String	Activity

Table 3.2 behavioural mapping rules

3.2.5 Automating Transformation

The purpose of this section is to show how to automate the transformation between models. The transformation between models realized by using MediniQVT, which is a tool that implement the QVT specification defined by OMG for model transformation (OMG, 2008).

The MediniQVT tool inputs are 1) source metamodel, 2) target metamodel, 3) source model and 4) mapping rules. The source and target metamodels define in Ecore using Eclipse Modeling Framework, but the source model must be conforming to source metamodel. The MediniQVT produces target model as output that is conforms to given target metamodel.

The automated transformation divided into two parts:

- Structural models transformation.
- Behavioral models transformation.

3.2.5.1 Structural Models Transformation

The structural models represent the structure nature of komponents. In the structural transformation the class specification transform to class realization, both model consider PIM because they describe the komponent from same level of abstraction. To automate models transformation the MediniQVT tool should take number of inputs as illustrate below.

3.2.5.1.1 Source Metamodel

The source metamodel is Class metamodel which describe at section 3.2.2.1, it design at magic draw tool, as shown in figure 3.1.

Class metamodel would used in MediniQVT as ecore file which obtain by export Class metamodel from magic draw as EMF XMI file and then import EMF XMI file into Eclipse Modeling Framework (EMF) tool to create ecore file.

3.2.5.1.2 Target Metamodel

The target metamodel is Class realization metamodel which describe at section 3.2.2.3, as shown in figure 3.3.

Class realization metamodel would used in MediniQVT as ecore file and obtained as we describe the Class metamodel.

3.2.5.1.3 Source Model

The source model is PIM (class model) which is instance of Class metamodel. PIM instance created at EMF tool as java file, with existence of ecore file of Class metamodel. PIM instance would used in Medini as XMI file which obtain by export instance as XMI file from EMF and then import XMI file to Medini . The XMI file of PIM shows in figure 3.5

PIMInstance.xmi

```
<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:class="http:///class.ecore"
xsi:schemaLocation="http:///class.ecore class.ecore">

  <class:Subject Name="LoanManager"/>

  <class:Operation Name="loanItem" id="1"/>

  <class:Operation isVariant="true" Name="ReserveItem" id="1"/>

  <class:Komponent Name="MessageHandler"/>

  <class:Operation Name="DisplyMessage" id="2"/>

  <class:Simple Name="Account"/>

  <class:Property Name="id" id="3"/>

  <class:Simple isVariant="true" Name="Reservation"/>

</xmi:XMI>
```

Figure 3.5 Structural, PIM instance – XMI file

3.2.5.1.4 Mapping Rules

The mapping rules specify which model elements should map to another models element.

The structural model mapping rules represent in table 3.1. At MediniQVT we create qvt file to represent the mapping rules as in figure 3.6.

classSpeToclassRea.qvt

```
 transformation PIM2PIIM(PIM :class , PIIM:pim ) {

 top relation SubjectToSubjectP {

 sn : String; on:String; onv:Boolean;

 checkonly domain PIM s : class::Subject { Name=sn };

 enforce domain PIIM si :pim::SubjectPIM { Name = sn };

 checkonly domain PIM o : class::Operation {isVariant = onv,

 Name = on , id = 1 };

 enforce domain PIIM oi :pim::OperationPIM{isVariant = onv,

 Name = on , id = 1 };}

 top relation komtokomp{

 kn:String ; isk : Boolean; okv:Boolean ; okn : String;

 checkonly domain PIM k :class::Komponent{ Name=kn
,isVariant=isk};

enforce domain  PIIM ki :pim::KomponentPIM{Name=kn ,
isVariant=isk};

checkonly domain PIM ok : class::Operation {isVariant = okv,

 Name = okn , id = 2};

enforce domain PIIM oki :pim::OperationPIM{

 isVariant = okv,

 Name = okn , id = 2};}

 top relation classtoclass{

 cn: String ; civ : Boolean ; pn : String ;

 checkonly domain PIM c :class::Simple{ Name=cn , isVariant=civ};

 enforce domain  PIIM ci :pim::SimplePIM{Name=cn ,isVariant=civ};

 checkonly domain PIM ca : class::Property{  Name = pn , id = 3};

 enforce domain PIIM cai :pim::PropertyPIM{Name = pn , id = 3};

}}}
```

Figure 3.6 Structural, qvt file

3.2.5.1.5 Target Model

The target model is PIIM (class realization model) which is instance of Class realization metamodel. PIIM instance would produce as result of running MediniQVT with ecore file of both Class realization and Class specification, in addition the PIM. Note that PIIM produce as XMI file as shown in figure 3.7

```
                         PIIMInstance.xmi
<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:pim="http:///pim.ecore"
xsi:schemaLocation="http:///pim.ecore pim.ecore">

 <pim:SimplePIM isVariant="true" Name="Reservation"/>

 <pim:PropertyPIM Name="id" id="3"/>

 <pim:SimplePIM Name="Account"/>

 <pim:OperationPIM Name="DisplyMessage" id="2"/>

 <pim:KomponentPIM Name="MessageHandler"/>

 <pim:SubjectPIM Name="LoanManager"/>

 <pim:OperationPIM isVariant="true" Name="ReserveItem"id="1"/>

 <pim:OperationPIM Name="loanItem" id="1"/>

</xmi:XMI>
```

Figure 3.7 PIIM instance – XMI file

3.2.5.2 Behavioral Models Transformation

The behavioral models represent the behavior aspects of komponent. In the behavioral transformation the state model (PIM) transform to activity model (PSM), both model at different level of abstraction. To automate models transformation the MediniQVT tool should take number of inputs as illustrate below.

3.2.5.2.1 Source Metamodel

The source metamodel is State metamodel which describe at section 3.2.2.2 , it design using magic draw tool , as shown in figure 3.2

State metamodel would used in Medini as ecore file which obtain by export State metamodel from magic draw as EMF XMI file and then import EMF XMI file into Eclipse Modeling Framework (EMF) tool to create ecore file.

3.2.5.2.2 Target Metamodel

The target metamodel is Activity metamodel which describe at section 3.2.3, as shown in figure 3.4.

Activity metamodel would used in Medini as ecore file and obtained as we describe the State metamodel.

3.2.5.2.3 Source Model

The source model is PIM (state model) which is instance of State metamodel. PIM instance created at EMF tool as java file , with existence of ecore file of State metamodel .

PIM instance would used in Medini as XMI file which obtain by export instance as XMI file EMF and then import XMI file to Medini . The XMI file of PIM shows in figure 3.8

PIMInstance.xmi

```
<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:stat="http:///stat.ecore" xsi:schemaLocation="http:///stat.ecore
stat.ecore">

  <stat:State isSimple="true" Name="accountIdentified"/>

  <stat:InternalActivity Name="loanItem"/>

  <stat:InternalActivity Name="returnItem"/>

  <stat:InternalActivity isVariant="true" Name="reserveItem"/>

  <stat:Transition kind="external"/>

  <stat:TransitionString Name="setAccout"/>

  <stat:TransitionString Name="cloaseAccout"/>

</xmi:XMI>
```

Figure 3.8 Behaviour , PIM instance XMI file

3.2.5.2.4 Mapping Rules

The mapping rules specify which model elements should map to another models elements. The structural models mapping rules represent in table 1.2. At Medini we create qvt file to represent the mapping rules as in figure 3.9

StateToActivity.qvt

```
 transformation PIM2PSM(PIM :stat , PSM:activity ) {

 top relation IntertoAc {

 ian :String ;d:Boolean;

 checkonly domain PIM s :stat::State{};

 checkonly domain PIM is:stat::InternalActivity{Name = ian
,isVariant=d};

enforce domain  PSM  a :activity::Activity{Name = ian ,
isVariant=d};}

top relation trStringtoActivity {

 trn:String;

 checkonly domain PIM s:stat::TransitionString{Name = trn};

 enforce domain  PSM  a :activity::Activity{

 Name = trn};

 } }
```

Figure 3.9 Behavior, qvt file

3.2.5.2.5 Target Model

The target model is PSM (activity model) which is instance of Activity metamodel.

PSM instance would produce as result of running Medini with ecore file of both Activity and State metamodel, in addition the PIM. Note that PSM produce as XMI file as shown in figure 3.10

```
PSMInstance.xmi

<?xml version="1.0" encoding="UTF-8"?>

<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:activity="http:///activity.ecore"
xsi:schemaLocation="http:///activity.ecore activity.ecore">

  <activity:Activity Name="cloaseAccout"/>

  <activity:Activity Name="setAccout"/>

  <activity:Activity isVariant="true" Name="reserveItem"/>

  <activity:Activity Name="returnItem"/>

  <activity:Activity Name="loanItem"/>

</xmi:XMI>
```

Figure 3.10 PSM instance – XMI file

CHAPTER 4

CONCLUSION

4.0 Conclusion And Discussion

In this research we have presented our results toward reengineering KobrA method using MDA approach. The lack of automation in KobrA which reduces the quality of engineering process and product, the time and the cost of development.

MDA has built-in machinery of automation and synchronization between metamodels which provides an opportunity to enhance KorbA. The key in MDA is formazling metamodels which enables automation. MDA development depends on two different abstraction levels PIM(application level) and PSM (implementation level) and automation of mapping among them using standard mapping language like QVT.

Special PIMs and PSM are developed for KobrA. A customization to UML metamodels is done to adopt KorbA concepts. A QVT rules are developed to automate mapping from PIM to PIM and PIM to PSM. It is also tested using MediniQVT engine. The PIMs would be reused with different PSMs through applying MDA development process. That's mean the komponent interface (PIM) can be reused with other komponent design (PSM). Moreover, KobrA can meet the continuous change in platforms and embraces of new ones , through adopting reengineering approach .

The concepts of KorbA is well organized with MDA as mapped only into two abstraction levels which are more easier to manipulate in terms of engineering. Furthermore it adds a classification terminology help distinguish KorbA artifacts. More important KorbA has no implementation model so MDA has added this descriptor (PSM).

In this research MDA enhanced KobrA basically in two points: firstly in automated mapping between Komponent description and implementation , secondly in brings high degree of reuse for KobrA artifacts which reduce the complexity especially the instable of requirements. In library systems framework develop by KobrA has no automated mapping from komponent description to komponent implementation and also in dealing with different implementation technology.

REFRENCES:

Anneke Kleppe , Jos Warmer , Wim Bast ,“MDA Explained - The Model Driven Architecture: Practice and Promise”, Addison-Wesley, 2003 .

Colin Atkinson, Joachim Bayer, Oliver Laitenberger and Jörg Zettel. ”Component-Based Software Engineering: The KobrA Approach.” Fraunhofer Institute Experimental Software Engineering (IESE) , 2000.

Colin Atkinson, Joachim Bayer, Christian Bunse, Erik Kamsties, Oliver Laitenberger, Roland Laqua, Dirk Muthig, Barbara Paech, Jürgen Wüst, and Jörg Zettel. Component-based product line engineering with UML. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

Colin Atkinson, Barbara Paech , Jens Reinhold , Torsten Sander. ” Developing and Applying Component-Based Model-Driven Architectures in KobrA.” Fraunhofer Institute for Experimental Software Engineering, Kaiserslautern. Germany, 2001.

Frank Truyen.” The Fast Guide to Model Driven Architecture.”the Cephas Consulting Corp , January 2006 .

Klaus Pohl, Günter Böckle, Frank J. van der Linden.” Software Product Line Engineering: Foundations, Principles and Techniques, 1st ed.” Springer, 2005.

Michalis Anastasopoulos, Colin Atkinson, and Dirk Muthig.” A Concrete Method for Developing and Applying Product Line Architectures.” Fraunhofer Institute Experimental Software Engineering (IESE) , 2002.

Joachim Bayer , Dirk Muthig and Brigitte Göpfert. “The library system product line - a kobra case study.” Technical report, Fraunhofer IESE-Report No. 024.01/E, IESE, 2001.

Model Driven Architecture (MDA) Document number ormsc/2001-07-01 Architecture Board ORMSC1 July 9, 2001

OMG.MDA: Executive Overview. 2005. http://www.omg.org/mda/executive_overview.htm (accessed 10 21, 2005).

OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification version 1.0. OMG, 2008.

OMG: MDA-Guide, Version 1.0.1, omg /03-06-01 (2003) .

OMG. "OMG Unified Modeling Language (OMG UML),Superstructure, V2.1.2." 2007 b: OMG Document Number: formal/2007-11-02.