Visual Representation of a Customizable Software Maintenance Process Model

Fuzhi Chen

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
for the Degree of Master of Computer Science at
Concordia University
Montreal, Quebec, Canada

June, 2009

Your file Votre référence
ISBN: 978-0-494-63130-0
Our file Notre référence
ISBN: 978-0-494-63130-0

Abstract

Visual Representation of a Customizable Software Maintenance Process Model

Fuzhi Chen

Managing the evolution of complex and large software systems involves many different types of resources and knowledge such as software artefacts, user expertise, tools and techniques, etc. Variations and interrelationships among these types of resources and knowledge create well-known challenges for maintainers. Current research mainly focuses on establishing comprehension model, and developing tools to tackle a specific aspect of maintenance problems. Little research has been conducted to study how resources and knowledge work collaboratively together to provide guidance to maintainers to complete specific maintenance tasks in a given context.

In this research, we introduce a customizable maintenance process model, which extends an existing IEEE standard process model, to allow visually link various resources (e.g. tools, artifacts, maintainers etc.) and knowledge to relevant maintenance process elements. A visual metaphor has been created to graphically represent the process model. Finally, a tool environment has been developed to provide utilities for maintainers to create, customize and apply our maintenance process to provide guidance for maintainers for their maintenance tasks.

Acknowledgements

I would like to thank my supervisor, Dr. Rilling, for his guidance, patience, support, and confidence throughout my research work. Without his guidance, this thesis would not have been possible. I am grateful that he provided me the opportunity to pursue this research.

I would also like to thank my wife, family and friends for their dedicated support, help and encouragement. I would like them to know that they are loved.

Table of Contents

List of Figures.....vii

List of Tables.....ix

Chapter 1 Introduction.....1

1.1 The Current State of Software Maintenance Process Models.....2

1.2 The Current State of Process Modeling Languages and Meta-models.....4

1.3 Research Hypothesis and Goals.....5

1.4 Organization of Thesis.....6

Chapter 2 Background.....7

2.1 Software Maintenance.....7

2.2 Software Maintenance Process Models.....9

2.2.1 Quick Fix Model.....10

2.2.2 Iterative Enhancement Model.....11

2.2.3 Full-reuse Model.....13

2.2.4 ISO/IEC 14764 Maintenance Process Model.....14

2.3 Process Modeling Languages and Their Notations.....17

2.3.1 BPEL.....18

2.3.2 BPMN.....20

2.3.3 UML's Activity Diagram.....25

2.4 Visualization Techniques.....28

2.4.1 Zooming and Panning.....29

2.4.2 Overview and Details.....32

2.4.3 High Complexity.....33

2.5 Existing Process Modeling Tools.....34

2.5.1 EPF Composer.....34

2.4.2 IBM Rational Method Composer.....36

2.4.3 Spemmet.....37

2.4.4 APSEE.....38

Chapter 3 Software Maintenance Process Modeller.....40

3.1 Maintenance Process Model.....42

3.1.1 Maintenance Process Model Adoption.....43

3.1.2 Process Execution Ability.....45

3.1.3 Process Flow 46
3.2 Process Meta-model and Graphical Notations 48
3.2.1 Process Meta-model 48
3.2.2 Graphical Notations 52
3.2.3 Layout Strategy 56
3.3 Stakeholders and Their Roles 58
3.4 Establishing Context-sensitive Supports 60
3.5 Data/User Experience Collection 63
3.6 Tool Integration 64
Chapter 4 Implementation 70
4.1 System Requirements 70
4.2 System Overview 71
4.3 System Implementation 73
4.3.1 Process Meta-model 74
4.3.2 Graphical Notations 77
4.3.3 Model – Notations Mapping 78
Chapter 5 Application 81
5.1 Process Design 81
5.2 Process Template Creation 84
5.3 Process Customization 88
5.4 Process Application 92
5.4.1 Process Instantiation 93
5.4.2 Process Navigation 93
5.4.3 Process Application 94
5.4.4 Feedback Collection 98
5.5 Discussion and Limitation 99
Chapter 6 Related Work 101
Chapter 7 Conclusion and Future Works 104
Bibliography 106

List of Figures

Figure 2-1 The Quick Fix Model [6] 11
Figure 2-2 The Iterative Enhancement Model [6] 12
Figure 2-3 The Full-reuse Model [6] 13
Figure 2-4 ISO/IEC--14764 Maintenance Process Model [9] 16
Figure 2-5 Event Notation of BPMN [23] 22
Figure 2-6 Activity Notation of BPMN [23] 22
Figure 2-7 Gateway Notation of BPMN [23] 22
Figure 2-8 Sequence Flow Notation of BPMN [23] 23
Figure 2-9 Message Flow Notation of BPMN [23] 23
Figure 2-10Association Notation of BPMN [23] 23
Figure 2-11 Pool Notation of BPMN [23] 24
Figure 2-12 Lane Notation of BPMN [23] 24
Figure 2-13 DataObject Notation of BPMN [23] 25
Figure 2-14 Group Notation of BPMN [23] 25
Figure 2-15 Annotation Notation of BPMN [23] 25
Figure 2-16 An Example of UML's Activity Diagram (Adopted From [48]) 26
Figure 2-17 An Example of UML's Activity Diagram with Swimlane [48] 28
Figure 2-18 The Construction of Zooming and Panning [52] 30
Figure 2-19 The EPF Approach [65] 35
Figure 2-20 The Notations of the APSEE Meta-model Elements[16] 39
Figure 3-1 The Overview of the Software Maintenance Process Modeller 40
Figure 3-2 The Tailored Software Maintenance Process Model 44
Figure 3-3 A Sample Process Model 48
Figure 3-4 The Maintenance Process Meta-model 51
Figure 3-5 Activity Notation of SMPM 54
Figure 3-6 Task Notation of SMPM 54
Figure 3-7 StartPoint Notation of SMPM 54
Figure 3-8 EndPoint Notation of SMPM 55
Figure 3-9 Mandatory Task-step Notation of SMPM 55
Figure 3-10 Optional Task-step Notation of SMPM 55
Figure 3-11 Unexecuted Optional Query Notation of SMPM (Grey) 55
Figure 3-12 Unexecuted Mandatory Query Notation of SMPM (Grey) 55
Figure 3-13 Failed Optional Query Notation of SMPM (Red) 55
Figure 3-14 Successful Optional Query Noation of SMPM (Green) 56
Figure 3-15 Failed Mandatory Query Notation of SMPM (Red) 56
Figure 3-16 Successful Mandatory Query Notation of SMPM (Green) 56
Figure 3-17 Connection Notation of SMPM 56
Figure 3-18 Task-step Layout Strategy 57
Figure 3-19 Task Layout Strategy 57
Figure 3-20 Activity Layout Strategy 58

Figure 3-21 User Roles and Their Responsibilities 58
Figure 3-22 Implementation of Context-sensitive Support 62
Figure 3-23 Design of Feeback View 64
Figure 3-24 The GUI Outline of SMPM 65
Figure 3-25 The Resource View of SMPM 66
Figure 3-26 The Outline View of SMPM 67
Figure 3-27 The Properties View of SMPM 67
Figure 3-28 The Graphical Editor Area of SMPM 68
Figure 4-1 System Overview 72
Figure 4-2 System Architectural Structure 74
Figure 4-3 Maintenance Process Meta-model Class Diagram 75
Figure 4-4 Connection Class Diagram 77
Figure 4-5 Graphical Notation Class Diagram 78
Figure 4-6 Process Model to Graphical Notations Mapping 79
Figure 4-7 Graphical EditPart Class Diagram 80
Figure 4-8 Tree EditPart Class Diagram 80
Figure 5-1 The Process Design Perspective in SMPM 86
Figure 5-2 The Completed Process Template Graph 87
Figure 5-3 Customization of the Execution Type 90
Figure 5-4 Customization of the Context Level 91
Figure 5-5 Customization of the Pre-defined Queries 92
Figure 5-6 The Software Maintainer Perspective in SMPM 95
Figure 5-7 Display of Query Results 96
Figure 5-8: Feedback Collection View 98

List of Tables

Table 3-1 Context Level Description 61
Table 3-2 Feedback Ranking Level 64
Table 5-1 Task-steps and Queries for Task 5.2.2.1 83
Table 5-2 Task-steps for Task 5.2.2.2 83
Table 5-3 Task-steps and Queries for Task 5.3.2.1 84
Table 5-4: Results of all Queries of Step 5.2.2.1.a 97

Chapter 1 Introduction

Managing the evolution of complex and large scale enterprise level software systems involves many different kinds of resources, such as artifacts (e.g., source codes, documents, etc.), tools (e.g., parsers, debuggers, source code analyzers, visualization tools, etc.) and knowledge (e.g., maintenance tasks, historical data, environment settings, etc.) [1,2,3]. Currently, research in software maintenance process domain mainly focuses on developing process models to describe activities to be performed and resources to be used within a maintenance a process context [4,5]. However, there is little research on how these supporting resources (tools, artifacts, knowledge, etc.) should be integrated within a process in a given context in order to complete a maintenance task [4,5]. There exists a need to link these resources and knowledge with relevant maintenance activities in a process context. In this chapter we will introduce the present state of software maintenance process models and highlight the need to link these artifacts, tools and knowledge with software maintenance activities in a process context.

Since one of the main contributions of the presented research is to visualize and enact the software maintenance process in a tool environment, we will introduce the current state of process modeling language and meta-modeling, key techniques used for process representation/definition, which are fundamental in process visualization and enactment domain.

1.1 The Current State of Software Maintenance Process Models

Software maintenance process domain has been actively researched for many years. Many process models [6,7,8,9] have been proposed to support the evolution of software systems. Common to these process models are that: 1) they separate the entire maintenance lifecycle into phases, for example, the iterative enhancement model [6] divides the maintenance process into five phases: requirements, design, code, test and analysis, and ISO/IEC 14764 [9] organizes maintenance lifecycle into six activities/phases: process implementation, problem and modification analysis, modification implementation, maintenance review/acceptance, migration and retirement; and 2) they mainly focus on listing and describing the sequence of related activities and their task-steps to be performed within a maintenance process, and resources to be used in order to complete a maintenance task [10,5]. However, these process models do not describe how the resources (tools, artifacts, knowledge, etc.) should be integrated within the model.

Software maintenance is a knowledge intensive process involving many different types of resources and knowledge. Maintainers may use or interact with various tools, artifacts and knowledge in order to complete a particular maintenance task. There are numerous tools existed to support software maintenance, for example, reverse engineering tools such as RIGI [11] and CppETS [12] are used to extract information by parsing source code; visualization tools such as CodeCrawler [13], Creole [14] and SeeSoft [15] provide maintainers with understanding of software systems, which are useful in program comprehension; bug tracking tools such as bugzilla[1] and Eventum [2] are used to help manage modification requests and problems reports and so on. Each of these tools is used to address a specific aspect of maintenance problem. Typically, different tools are used to tackle issues in different maintenance process phases, and multiple tools are used in the same maintenance phase. Identifying tools to tackle problems in different maintenance process phases can become challenge because there is no existing guidance to guide maintainers on how to select tools for a specific maintenance task in a given maintenance process context [5]. In order to address this problem, there exists a need to link these tool resources with their related process phases.

Software artifacts and knowledge are other types of resources that are considered as important as tools to software maintenance. Since knowledge about software systems is often spread in several different software artifacts, such as source code, documentation, manuals, bug description, etc., obviously, some artifacts are more useful and important than others in a specific maintenance process phase. In order to identify these artifacts and knowledge for a particular maintenance task in a process context, there also exists a need to link these artifacts and knowledge with their

[1] http://www.bugzilla.org

[2] http://eventum.mysql.org

related process phases.

1.2 The Current State of Process Modeling Languages and Meta-models

In the past years, many process modeling languages have been created to represent various process models, for example, APSEE/APSEE-PML [16], SPADE/SLANG [17], Oikos/ESP [18], PSEE/CSPL [19], RHODES/PBOOL [20], and so on. Recently, research trend tends to unify process modeling languages according to particular application domains, for example, BPEL [21,22] and BPMN [23] are process languages used specific for modeling business processes. UML [24,25] and SyUML[3] are modeling languages used to describe software systems. Nowadays, meta-modeling technique is becoming a hotspot in process modeling research field [26]. Quite a few meta-models such as WfMC [27] and SPEM [28] have been developed to represent various process models. Particularly, SPEM is a meta-model used specific to software process engineering domains.

However, software maintenance processes have different characteristics than business processes and software development processes. Therefore, the process modeling languages and process meta-models mentioned above cannot model well with maintenance processes. Specific to software maintenance process domain, there is no such a standard process modeling language or meta-model existed. In order to

[3] http://www.omgsysml.org

describe a software maintenance related process model so that it can be visualized or enacted, developers usually either create a new process modeling language/meta-model or adapt from an existed one. For example, in [29] Rasovska et al proposed a maintenance process and described it using UML's class diagram. At the time we conducted this research, there is no existing process modeling language/meta-model for the ISO/IEC 14764 maintenance process model. In order to describe this maintenance process so that we can visualize and manipulate it, there exists need to create a process model language/meta-model to describe it.

1.3 Research Hypothesis and Goals

One goal of the presented research is to link various resources and knowledge with a software maintenance process model so that it can provide guidance on which resources and/or knowledge should be used for a given maintenance task in a process context. The maintenance process model used in the research is extended from the existing ISO/IEC – 14764 maintenance process model [9]. The second goal of the research is to graphically represent the maintenance process model in a tool environment.

Our research **hypothesis**, therefore, is that it is feasible to extend the ISO/IEC – 14764 maintenance process model to allow link various resources and knowledge with maintenance activities and their task-steps so that it can provide guidance on which resources and/or knowledge should be used for a given maintenance task in a process context.

The research goal can be further decomposed into the following sub-goals:

- Extend the ISO/IEC-14764 maintenance process model to link various resources and knowledge with their process activities and task-steps;
- Create a process meta-model to describe the extended process model;
- Create a set of graphical notations to represent the maintenance process model elements; and
- Implement the maintenance process model in a tool environment.

The **null-hypothesis** is any of the above sub-goals is not met.

1.4 Organization of Thesis

Chapter 2 provides the necessary background related to software maintenance, specifically software maintenance process model. Process modeling languages and some other visualization techniques will also be reviewed. The proposed approach will be detailed in Chapter 3. A tool environment that implements our presented approach is described in Chapter 4. This is followed in Chapter 5 with the application of the tool environment, and a discussion and limitation section is also included in this chapter. Related work will be discussed in Chapter 6. Finally, conclusion and future works will be presented in Chapter 7.

Chapter 2 Background

In this chapter, we will review background relevant to software maintenance, process models, process modeling languages and information visualization techniques. This is followed by a review of several existing process modeling tools.

2.1 Software Maintenance

In existing literature, several definitions of software maintenance are found [30,31]. One commonly used definition by the IEEE is:

The modification of a software product after delivery to correct faults, to improve performance or other attributes or to adapt the product to a modified environment [32]

This definition implies that software maintenance is not limited to the correction of latent faults. It usually refers to all the changes that must be made to softwares after they have been delivered. Lientz and Swanson [33,34] categorized software maintenance activities into four classes: *adaptive maintenance*, *perfective maintenance*, *corrective maintenance* and *preventive maintenance.*

Adaptive Maintenance provides enhancements necessary to accommodate changes in the environment in which a software product must operate. These changes are those that are required in order to keep pace with the changing environment. The term environment in this context refers to the totality of all conditions and influences which act from outside upon the system, for example, business rule, government policies, work patterns, software and hardware operating platform [35].

Perfective Maintenance is the modification of a software product after delivery to detect and correct potential faults before they are manifested as failures. It concerns functional enhancements to the system and activities to increase the system's performance or to enhance its user interface [36].

Corrective Maintenance deals with the repair of faults or defects found. A defect can result from design errors, logic errors and coding errors [35]. Design errors occur when, for example, changes made to the software are incorrect, incomplete, wrongly communicated or the change request is misunderstood. Logic errors result from invalid tests and conclusions, incorrect implementation of design specifications. Fault logic flow or incomplete test of data. Coding errors are caused by incorrect implementation of detailed logic design and incorrect use of source code logic. The need for *corrective maintenance* is usually initiated by problem report (PR) drawn up by the end users.

Preventive Maintenance concerns activities aimed at increasing the system's maintainability, such as updating documentation, adding comments, and improving the modular structure of the system [36]. The long-term effect of corrective, adaptive and perfective changes increases the system's complexity [35]. As large software systems continuously changed, their complexity increases unless work is done to maintain or reduce it. This work is known as *preventive maintenance*. The *preventive maintenance* is usually initiated from within the maintenance organization with the intention of making software systems easier to understand and hence facilitating future maintenance work [35].

Among these four types of maintenance, only *corrective maintenance* is considered as 'traditional' maintenance. The other types can be considered as software 'evolution'. The term evolution has been used since the early 1960s to characterize the growth dynamics of software [37]. Software evolution is now widely used in the software maintenance community.

2.2 Software Maintenance Process Models

Software maintenance process domain has been researched for many years. Various process models [6,7,8,9,38] for software maintenance have been proposed. Traditionally, software maintenance was considered as the final activity of the software development process. Even a few years ago, IEEE1074-1997 [39] still represents software maintenance as the seventh step of eight software development steps. Recently, research trend tends to derive software maintenance from the development process. An extensive collection of maintenance process models [6,2,9,40] have been created based on this observation. These models describe software maintenance as a sequence of activities instead of the final stage of the software development process. In particular, the latest version of the international standard, ISO/IEC 14764-2006 [9] is such maintenance-specific process model.

In what follows, we review some of maintenance-specific process models, e.g., quick-fix model, iterative enhancement model, full-reuse model and ISO/IEC 14764-2006 software maintenance process model.

2.2.1 Quick Fix Model

The *quick fix model* [6] is to directly identify the problem in the code and then fix it as quickly as possible. As shown in Figure 2-1, it demonstrates the work flow that the quick fix model usually follows. Ideally, changes should be made to accompanying requirements, design and documentation after the code has been changed. However, due to time and/or cost constraints, changes are often made on the fly, without proper planning, design and documentation. The advantage of this model is that it gets work done quickly with lower cost. The disadvantage is that it does not pay attention to the long-term effects of the fixes. Repeated changes may outdate the documentation and destroy the original design, making future changes more expensive and difficult to carry out.

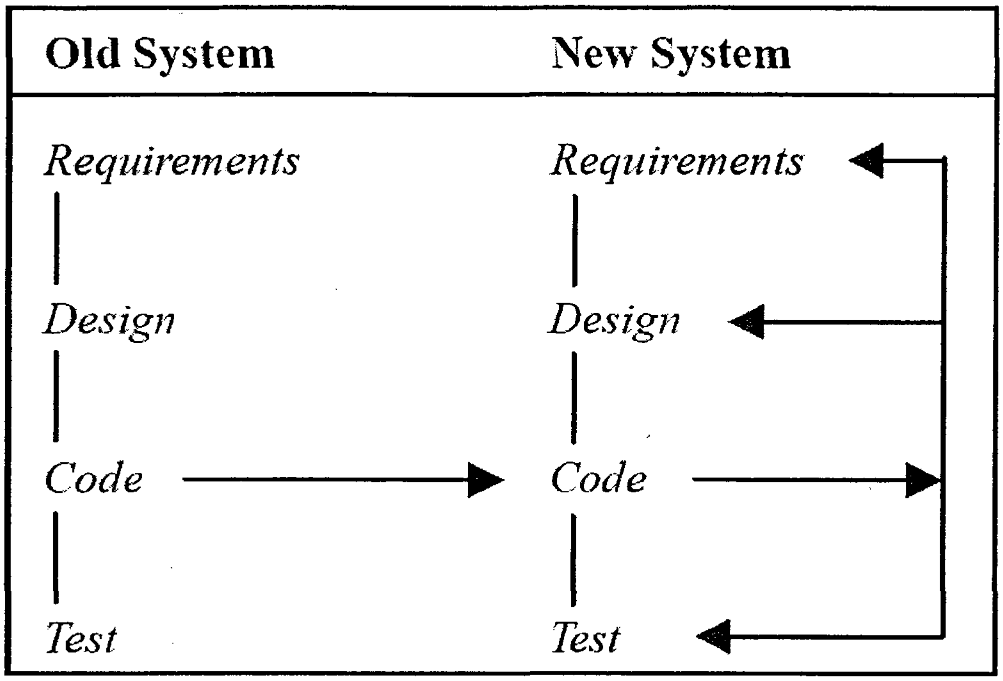


Figure 2-1 The Quick Fix Model [6]

2.2.2 Iterative Enhancement Model

The *iterative enhancement model* [6] is one of the evolutionary life cycle models that considers the changes made to the software make up an iterative process. The idea of this model is that the requirements of a system cannot be understood initially. As a consequence, systems are to be changed in iterations each of which completes, corrects and refines the requirements of the previous iteration based on the feedback collected from users. As shown in Figure 2-2, the process starts with the analysis of the existing system's requirements, design, code and test documentation and continues with the modification of the highest-level document affected by the changes, propagating the changes down to the full set of documents. At each of iteration of the evolutionary process, the system is redesigned based on an analysis of the existing system.

One of the key advantages of the *iterative enhancement model* is that documentation is consistent with the code changed [10]. So it is well suited for systems that have a long life and evolve over time. The drawback of the model is that it is not effective when the documentation of the system is not complete, as the model assumes that a full and update-to-date documentation of the system exists [41].

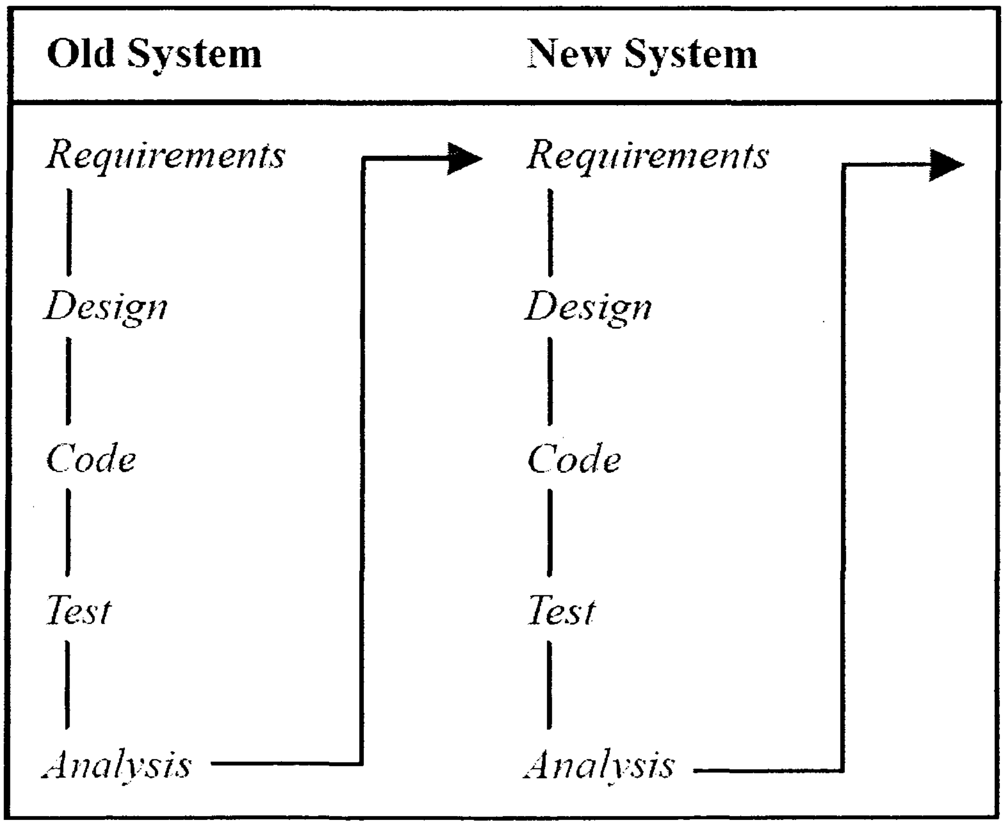


Figure 2-2 The Iterative Enhancement Model [6]

2.2.3 Full-reuse Model

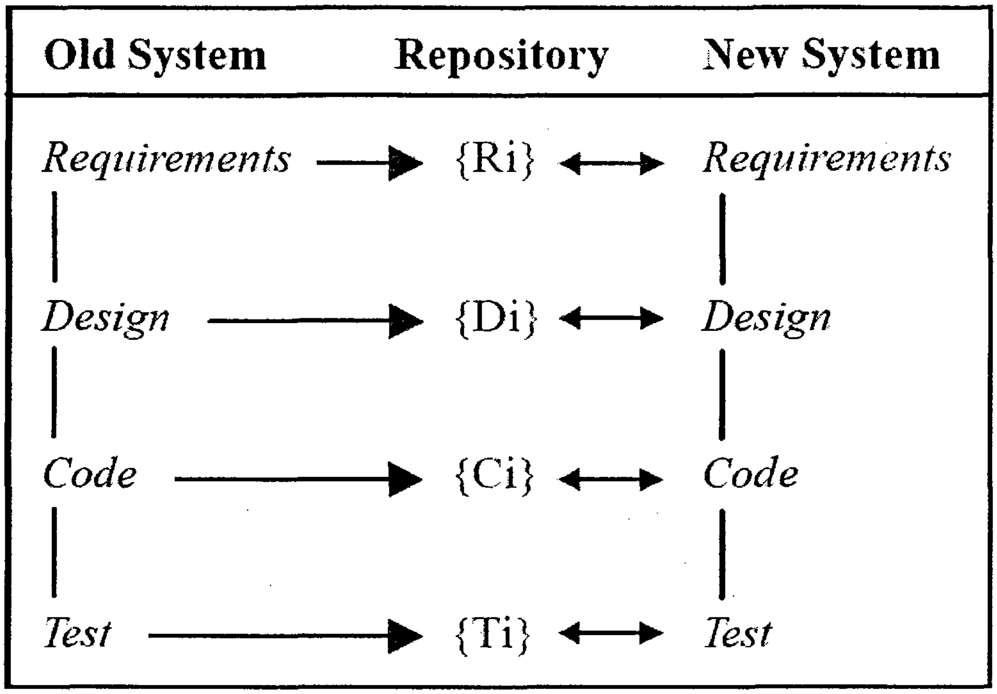


Figure 2-3 The Full-reuse Model [6]

The *full-reuse model* **[6]**, as shown in Figure 2-3, is a particular case of reuse-oriented software development. Central to the *full-reuse model* is that it assumes there exists a repository of software artifacts from the current and earlier versions of the subject system or other similar systems. The *full-reuse model* begins with the requirement analysis and design of a new system by reusing the appropriate requirements, design, code, and tests from earlier version of the existing system. The *full-reuse model* promotes the development of reusable artifacts and encourages their reuse in modification tasks. The effort spent on building reusable artifacts tends to be more costly on the short term, however the advantage may be sensible in the long term, because the accumulation of reusable artifacts of all kinds and at many different levels of abstractions makes future development more cost effective [10].

2.2.4 ISO/IEC 14764 Maintenance Process Model

The ISO/IEC 14764 maintenance process model [9] describes activities, tasks and task-steps necessary to perform software maintenance. The entire process is divided into six activities with respect to the software lifecycle model as shown in Figure 2-4: *process implementation*, *problem and modification analysis*, *modification implementation*, *maintenance review/acceptance*, *migration* and *retirement*. Each of these activities contains a set of tasks, which are further refined by a list of task-steps.

Process implementation: This activity contains tasks for 1) developing plans and procedures for software maintenance, 2) establishing procedures for receiving, recording, and tracking maintenance requests and problem reports, and 3) developing configuration management plans for managing modification to the existing system. According to ISO/IEC 14764, this activity is the start point of the maintenance life cycle. It develops and documents strategies and plans for performing maintenance tasks. Actually, the maintenance plan should be developed in parallel with the development plan.

Problem and modification analysis: This activity is further divided into several tasks. The first task mainly focuses on analyzing the received problem report or modification request to classify the type of maintenance (e.g. corrective, adaptive, preventive and perfective maintenance), and determine its impact and scope (e.g., size of modification, cost involved, and time required, etc). The next task concerns with the verification of maintenance request by reproducing the reported problem on the affected software version. The other tasks of this activity regard the development and the documentation of alternative for change implementation and the approval of the selected option as specified in the contract.

Modification implementation: During this activity, the elements to be modified are identified, and then the development process (e.g., ISO/IEC 12207[4]) is invoked to actually implement the changes. Finally, tests are performed to ensure that changes are correctly implemented and the original unmodified requirements are not affected.

Maintenance review/acceptance: This activity includes tasks to review the modifications to the software system are correct and comply with approved standards using the correct methodology. Approval is obtained if maintenance request is complete and satisfactory. Several supporting processes may be invoked during this activity, including quality assurance process, verification process, validation process, and joint review process.

Migration: When a software system is moved from one environment to another, this activity is invoked. Prior to the activity is really performed, maintainers should

[4] http://www.12207.com

develop a migration plan, notify users of the migration, provide training to the users, assess the impact of the new environment, and archive the data of the old software system. Other tasks of this activity are the parallel operations of the old and new environments and the post-operation review to assess the impact of changing to the new environment.

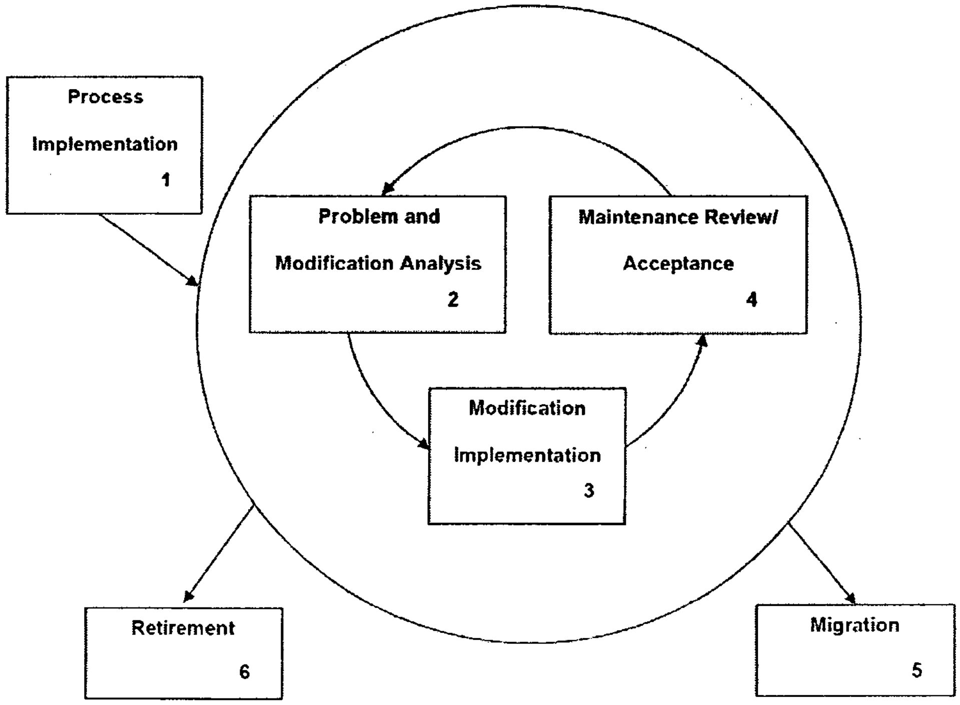

Process Implementation 1
Problem and Modification Analysis 2
Maintenance Review/ Acceptance 4
Modification Implementation 3
Retirement 6
Migration 5

Figure 2-4 ISO/IEC--14764 Maintenance Process Model [9]

Retirement: It is the last activity of the maintenance life cycle. It contains tasks to develop retirement plan and give the notification of the retirement to the users. It also contains tasks concerning about parallel operations of the old and new systems, providing training to the users if it is specified in the contract, and archiving the data of the old system.

2.3 Process Modeling Languages and Their Notations

Process modeling languages and process meta-models are two closely related concepts in process modeling domain. A process modeling language is used to describe a process model [42], while a meta-model is the description of a set of other models [43]. Favre [44] defines a meta-model as a model of a modeling language. This means that "*the task of creating meta-model is the task of creating a modeling language that is capable to describe the relevant aspects of a subject under consideration* [43]". In this thesis, we consider process modeling languages and process meta-models as the same concept and use them interchangeable, although they are not.

As the model-driven development is popular nowadays [45], many process modeling languages [45,46,24,21] have been designed to describe, visualize, control and execute different process models.

Among these modeling languages, the Business Process Execution Language [21], known as BPEL, is a de facto standard for describing the behavior of business processes. Since BPEL is only a modeling language, there is no graphical notation defined in BPEL specification [22]. Business Process Management Initiative (BPMI[5]) proposed a specification, called Business Process Modeling Notation (BPMN) [23], to fill this gap. BPMN defines a set of notations as graphical front-ends, and maps these

[5] http://www.bpmi.org

notations to process elements defined in BPEL processes. Therefore, BPEL combining with BPMN provide full functionalities to graphically model business processes.

The Unified Modeling Language [24], also known as UML, is another graphical modeling language for visualizing, specifying, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems. According to UML 2.0 specification [24], there exists 13 types of diagrams divided into three categories, e.g., class diagram, component diagram, object diagram, package diagram, activity diagram, use case diagram, etc. A diagram is a partial graphical representation of a system's model. Among these diagrams, activity diagram is suitable for business process modeling, which closely relates to the presented research. SPEM [28] which is a meta-model for software development process is closely related to the UML as well. It adopts a lot of UML diagrams such as package diagram, use case diagram, class diagram, activity diagram, sequence diagram, and so on [47], and offers an object oriented approach using the UML notations.

In this section, we will review those modeling languages that are closely related the presented research, e.g., BPEL, BPMN and UML's activity diagram.

2.3.1 BPEL

BPEL is a modeling language for describing the behavior of business processes.

Such a business process can be described in two different ways: either as an *executable process* or as an *abstract process*. An *executable process* models the behavior and the interface of a partner in a business interaction. It specifies the execution order between a collection of activities, the partners involved, the message exchanged between these partners, and the fault and exception handling mechanisms. An *abstract process*, in contrast, is a business protocol only modeling the interface and the message exchange of a partner. It specifies the message exchange behavior between different parties without revealing the internal behavior of any of them.

For the specification of a business process, BPEL provides two kinds of *activities*. An *activity* is either a *basic activity* or a *structured activity*. The set of *basic activities* includes:

Invoke: to invoke a partner;

Receive: to wait for a message from a partner;

Reply: to reply to an external source;

Wait: to wait for some time;

Assign: to copy a value from one place to another;

Throw: to indicate an error in the execution;

Terminate: to terminate the entire service instance; and

Empty: to do nothing.

A *structured activity* defines a causal order on the *basic activities*. It can be nested with other *structured activities*. The set of *structured activities* includes:

Sequence: to process activities sequentially;

If: to process activities conditionally;

While and *RepeatUnit*: to execute activities repeatedly;

Pick: to process events selectively;

Flow: to process activities in parallel; and

Scope: to group activities into a block, to link this block to transaction management, and provide fault, compensation, termination and event handling.

Except basic activities and *structured activities*, another important component in BPEL is *link*. A *link* is used to define the execution order between two concurrent activities in a process flow. A *link* contains a source activity and a target activity. The target activity may only start when the source activity has ended.

2.3.2 BPMN

Since there is no graphical notation defined in BPEL specification [23,22], BPMN is developed to provide a set of notations as graphical front-ends, and maps these notations to process elements defined using BPEL. The basic goal of the BPMN is to provide a set of notations that is understandable by all business participants. Therefore, notations are chosen to be distinguishable from each other, and the shapes of the notations are familiar to their users.

BPMN provides a small set of notation categories so that the users can easily recognize the basic type of elements and understand the diagram. The four categories of elements are as follows: *flow objects*, *connecting objects*, *swimlanes* and *artifacts*.

Flow objects are the main describing elements in BPMN, and consist of three core elements: *events*, *activities*, and *gateways*.

An *event* is represented by a circle (Figure 2-5). *Events* happen during the course of a business process. They affect the flow of a process and usually have a cause (trigger) or an impact (result). There are three types of *events*, based on when they affect the flow: *start*, *intermediate*, and *end*.

An *activity* is represented by a rounded-corner rectangle (Figure 2-6). It is a generic term for work that company performs. *Activities* can be atomic or non-atomic (compound). The types of *activities* are: *task*, *sub-process* and *transaction*.

A *gateway* is represented by a diamond shape (Figure 2-7) and is used to control the divergence and convergence of *sequence flow* which will be described later in this section. Therefore, it can be used to determine branching, forking, merging, and joining of paths.

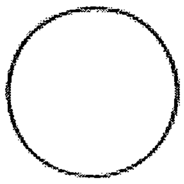


Figure 2-5 Event Notation of BPMN [23]

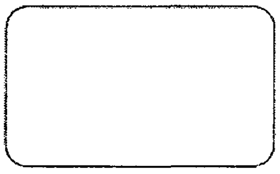


Figure 2-6 Activity Notation of BPMN [23]

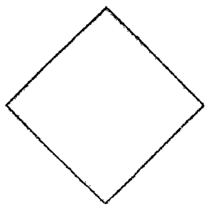


Figure 2-7 Gateway Notation of BPMN [23]

The *connecting objects* are used to connect *flow objects* together in a diagram to create the basic skeletal structure of a business process. There are three *connecting objects* which are: *sequence flow*, *message flow* and *association*.

A *sequence flow* is represented by a solid line with a solid arrowhead (Figure 2-8) which is used to show the sequence that *activities* will be performed in a business process.

A *message flow* is represented by a dashed line with an open arrowhead (Figure

2-9). It is used to show the flow of messages that are sent and received between two separate participants.

An *association* is represented by a dotted line with line arrowhead (Figure 2-10). It is used to associate data, text, and other *artifacts* with the *flow objects*.

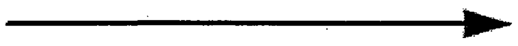


Figure 2-8 Sequence Flow Notation of BPMN [23]

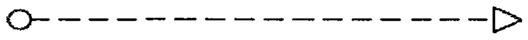


Figure 2-9 Message Flow Notation of BPMN [23]

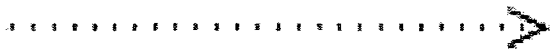


Figure 2-10Association Notation of BPMN [23]

A *swimlane* is a visual mechanism of organizing and categorizing *activities*. It arranges and groups *activities* into separate visual categories according to the responsibilities of those *swimlanes*. BPMN supports *swimlane* with two main types of *swimlane* objects: *pool* and *lane*.

A *pool* (Figure 2-11) represents a major participant in a process. A *pool* contains one or more *lanes*, which likes a real swimming pool. It acts as a graphical container for partitioning a set of *activities* from other *pools*. *Pools* are used when the diagram involves two or more separate business entities or participants and are physically separated in the diagram. The *activities* within separated *pools* are considered self-contained processes.

A *lane* (Figure 2-12) is a sub-partition within a *pool*, used to organize and category *activities* according to function or role. It depicts as a rectangle stretching the width or height of the *pool*. A *lane* can contain *flow objects*, *connecting objects* and *artifacts*.

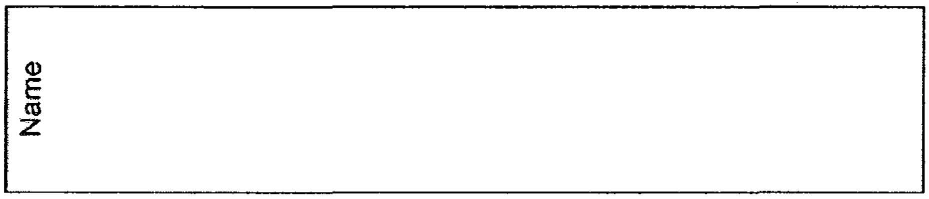


Figure 2-11 Pool Notation of BPMN [23]

Name
Name
Name

Figure 2-12 Lane Notation of BPMN [23]

Artifacts allow developers to bring more information in the process diagram. So the process model becomes more readable. There are three pre-defined *artifacts* and they are: *data objects*, *group* and *annotation*.

A *data object* (Figure 2-13) is a mechanism to show how data is required or produce by *activities*. It is connected to an *activity* through an *association*.

A *group* is represented by a rounded corner rectangle drawn with a dashed line

(Figure 2-14). It is used for documentation or analysis purposes, which does not affect the *sequence flow.*

An *annotation* (Figure 2-15) is a mechanism to provide additional text information for the users of a BPMN diagram.

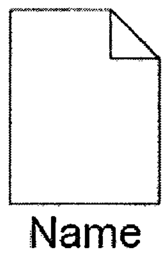


Figure 2-13 DataObject Notation of BPMN [23]

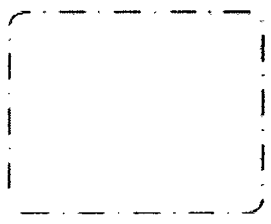


Figure 2-14 Group Notation of BPMN [23]

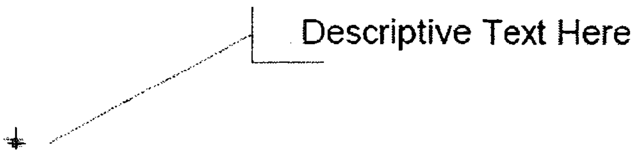


Figure 2-15 Annotation Notation of BPMN [23]

2.3.3 UML's Activity Diagram

UML's *activity diagram* can be used to describe the business and operational workflows of components in a system. An *activity diagram* can show workflows of stepwise activities and actions, with support for choice, iteration and concurrency.

Figure 2-16 depicts a typical business process that is described using UML's *activity diagram.* We will use this figure as an example to describes some of basic notations of an *activity diagram.*

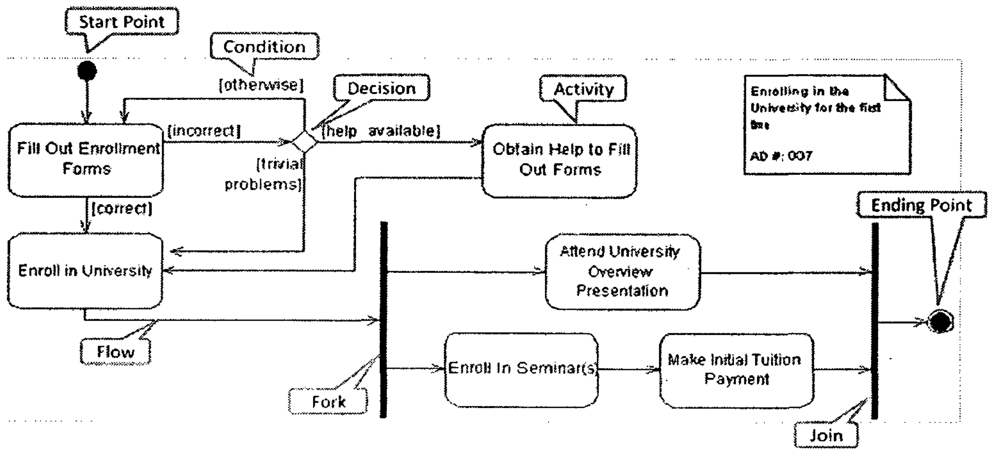


Figure 2-16 An Example of UML's Activity Diagram (Adopted From [48])

A *start pont* is represented as a filled in circle, which is the starting point of the diagram. Every *activity diagram* should have a *start point.*

An *ending point* is modeled with a filled in circle with a border around it, indicating the end of the process. An *activity diagram* can have zero or more *ending points.*

An *activity* is represented using a rounded rectangle. It is typically used to represent activities, such as invocation of an operation, a step in a business process, or an entire business process.

A *flow* is an arrow on the diagram. The direction of the arrowhead indicates the direction of the *flow*.

A *fork* is depicted as a black bar with one incoming *flow* and several outgoing *flows*. It denotes the beginning of parallel activity.

A *join* is depicted as a black bar with several incoming *flows* and only one outgoing *flow*. All incoming *flows* must reach the *join* before processing may continue. A *join* denotes the end of parallel processing.

A *condition* is text information attached to a *flow*. It defines a condition which must evaluate to true in order to traverse a transition.

A *decision* is a diamond with one incoming *flow* and several outgoing *flows*. The outgoing *flows* typically include *conditions*.

A *merge* is a diamond with several incoming *flows* and one outgoing *flow*. A *merge* implies that one or more incoming *flows* must reach it until processing continues.

In addition to the basic notations described above, UML also uses *swimlane* to group *activities* in an *activity diagram*. Figure 2-17 illustrates an *activity diagram* that uses *swimlane* to group *activities* by actors. As shown in the figure, there are three actors: stakeholder, requirement analyst and enterprise architect. Therefore, there are three *lanes* included in the diagram, one for each actor.

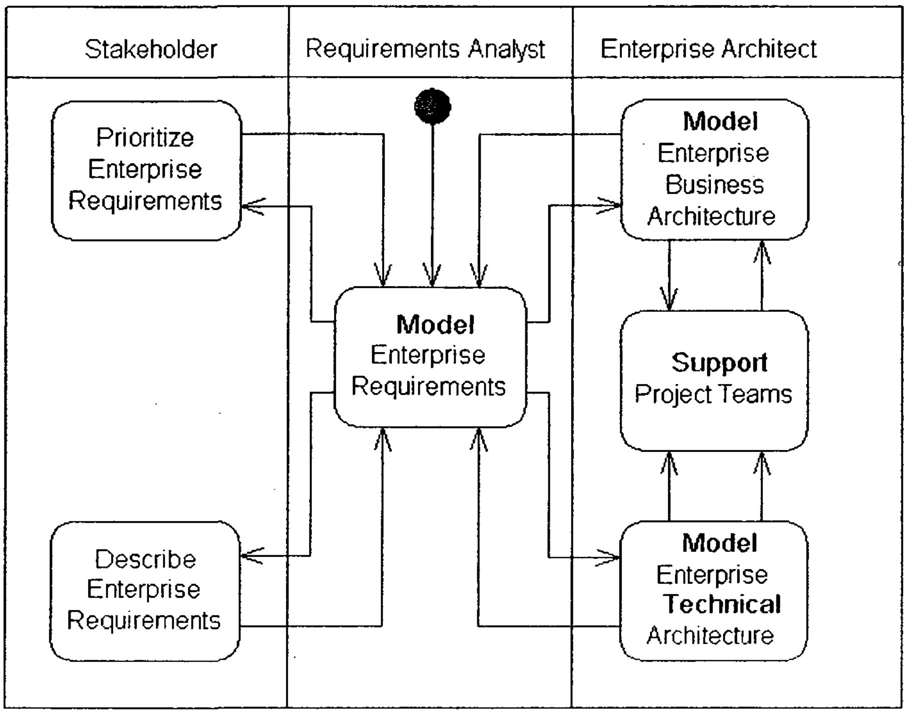

Stakeholder
Requirements Analyst
Enterprise Architect
Prioritize Enterprise Requirements
Model Enterprise Requirements
Model Enterprise Business Architecture
Support Project Teams
Describe Enterprise Requirements
Model Enterprise Technical Architecture

Figure 2-17 An Example of UML's Activity Diagram with Swimlane [48]

2.4 Visualization Techniques

One of the main challenges faced in the process modeling domain is the ability to provide users with meaningful visualization tools. The process model is usually presented by different types of flow charts and diagrams. If the modeling process is simplified in its representation to users, it improves the understanding of the results. This involves the techniques of process visualization. One of the common problems associated with visualization is the relatively small space through which a large amount of information is displayed [49,50]. Because of the problem of needing to display large amounts of information in a limited space, substantial research has been invested to find solutions to fit more relevant data into limited space while reducing irrelevant information. In this section, we will review various types of techniques in the information visualization domain. These techniques can also be applied to process visualization.

2.4.1 Zooming and Panning

The use of *zooming and panning* in visualization is one of the most basic approaches used to display large amount of information. The key to a *zooming and panning* visualization is the notion of what is so-called multi-scale viewing [50]. Information, and its inherent structure, can be displayed at many different space-scales/magnifications. The basic idea of *zooming and panning* visualization is to create many copies of the same 2-D image, one at each possible magnification, and then stacking them up to form an inverted pyramid as demonstrated in the right hand side of Figure 2-18. The vertical axis represents scale/level of magnification; each concentric plane represents a different level of magnification. A viewing window (indicated as *(a)* in Figure 2-18) can be represented as a fixed-size screen, which can be moved through the space-scale diagram, generating all possible views of the original 2D picture. As seen from Figure 2-18 *(d)*, when the viewing window moves in horizontal directions, different parts of the picture can be shown at the same magnification level. This is so-called panning. However, if the viewing window moves in a downward vertical direction, a bigger part of the original picture with less detail will be displayed (Figure 2-18 *(c)*); this is called zoom out. Whereas, if the viewing window moves in an upward vertical direction, a smaller part of the original picture with greater details can be displayed (Figure 2-18 *(b)*), this is called zoom in.

One of the main advantages of *zooming and panning* visualization technique is that it renders an undistorted visualization of a large dataset in a relatively small space, while allowing users to navigate this data using pan and zoom [52].

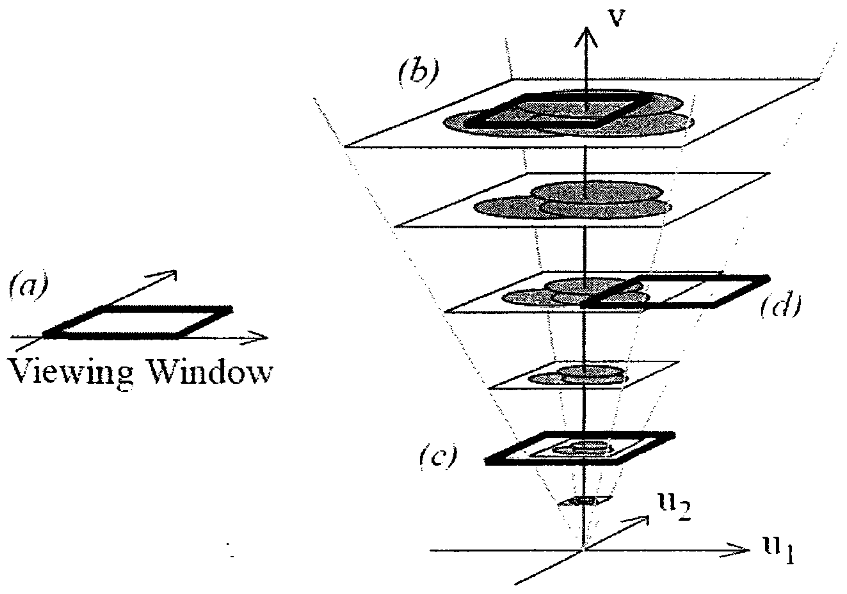


Figure 2-18 The Construction of Zooming and Panning [52]

However, the biggest problem of the *zooming and panning* approach is that the representation may still contain too much information therefore making it too difficult to comprehend and navigate. Many techniques have been devised to ease the navigation between *zooming and panning* positions. Olston and Woodruff [53] summarized six techniques which allow users to rapidly navigate large amount of information.

- Visual Hyperlinks – It is a hypertext style of hyperlink. Like web pages, it allows users to instantly jump from one location to another location.
- Bookmarks – Bookmarks are visual hyperlinks that allow users to bookmark and recall at any time.
- Coordinated Views – A coordinated view shows a different representation of the same data in a main window. It is quite useful for the area where data has multiple alternative representations showing different features [54].
- Overviews – It is also a technique used to help orient the user during navigation. In practice, it is almost always used in combination with a detail view, which is so-called *overview and detail* view, which will be further discussed in Section 2.4.2.
- Filters – Filters allow display of the same set of data in two different graphical representations [55]. For example, suppose we have defined a bar chart filter which can "see" all the tabular data as a bar chart. By applying such a bar chart filter to a canvas which contains some tabular data, the effect is that the bar chart filter will "see" any tabular data as a bar chart, but will see other data in their usual way.
- Magnifying Glasses – a magnifying glass is used to show a specific portion of

the data in greater detail by zooming in. It is useful because it enables users to see different portions of the data in detail without navigation. Actually, fisheye is a kind of magnifying glasses technique.

2.4.2 Overview and Details

Although *Zooming and Panning* approaches can provide support for smooth and rapid navigation among large datasets, it does not provide context support for the users while navigating these large-scale information spaces [49]. *Overview and detail* visualization is one of the solutions to address this problem. This technique is also called multi-view or multi-window arrangement [56,57,58]. As implied by its name, *overview and detail* visualization consists of two or more views. One view is the overview which always displays the entire collection of information, while other views are the detail views. Each of these detail views shows a close-up of a portion of the information in a specific aspect.

Overview and detail visualization technique is considered as one of non-distortion-oriented presentations [49], and has been used for quite some time to visualize both the textual and graphical data [59,60]. It has been proven especially suitable for displaying data that has inherently or implicitly spatial relationships, for example, geographic information systems.

One of the most significant features in *overview and detail* visualization is that it displays overview and detail views simultaneously on the same screen. This technique helps users orient themselves in the large information space, meanwhile still providing them with enough details [61].

Although overview can support users in orienting themselves in large information spaces, switching the focus back and forth between the overview and detail views and the need for reorientation within the overview will still result in a frequent loss of orientation and context [61]. Leung & Apperley [49] addressed this problem by using a visual marker in the overview to indicate the position of the detail view within the overvew.

2.4.3 High Complexity

One important issue in information visualization is the reduction of complexity. Many visualization techniques work well with a small set of data, but they do not scale well [62]. For instance, graph-based representations become cluttered and users get overwhelmed by the sheer number of nodes and relations rendering on the limited space of the computer screen [50,63]. This problem can be solved by filtering [62,64]. By using filtering, only the elements and relations of interest are visualized.

Aggregation [62] is another approach to reduce the complexity by getting higher level of abstraction from a large amount of information. It is an appropriate and effective way to help understanding of information. The basic idea of aggregation is that an entity works as a container representing all of its children and all of the relationships between these children. Recursively, the current entity also represents the entities (grandchildren) and relationships inside each of the children. By doing aggregation, we can get a bigger picture of the information. For example, in Java language, a package structure is an aggregation unit containing a set of related classes. And a class is an aggregation of methods and variables. It is useful for creating views at a higher level of the entire software system.

2.5 Existing Process Modeling Tools

Many tools have been developed for the process modeling domain. In what follows, we will review those that closely relate to software re/engineering process modeling.

2.5.1 EPF Composer

The Eclipse Process Framework composer (EPF[6] Composer) is a process management tool platform and conceptual framework developed by eclipse.org. It is a customizable software process engineering framework supporting a broad variety of project types and development styles. EPF Composer provides an easy-to-learn user experience and simple-to-use features for authoring, tailoring, and deploying of development process frameworks.

[6] http://www.eclipse.org/epf

As shown in Figure 2-19, the most fundamental principle in the EPF is the separation of reusable core method content from its application in processes. Method content describes what is to be produced, the necessary skills required, and the step-by-step explanation describing how specific development goals are achieved. These method content descriptions are independent of a development lifecycle. Processes describe the development lifecycle. They take the method content elements and relate them into semi-ordered sequences that are customized to specific type of projects.

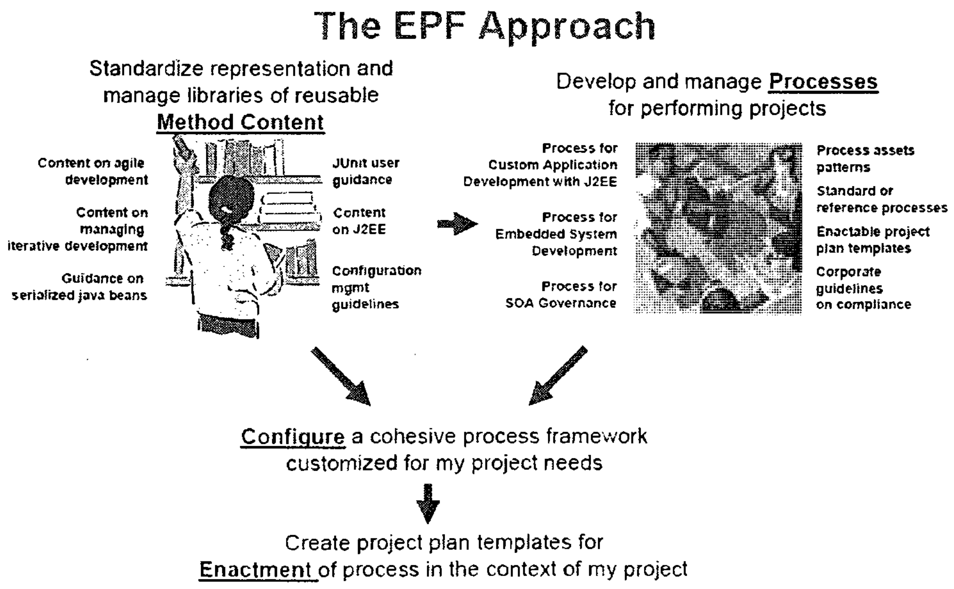

The EPF Approach
Standardize representation and manage libraries of reusable Method Content
Content on agile development
Content on managing iterative development
Guidance on serialized java beans
JUnit user guidance
Content on J2EE
Configuration mgmt guidelines
Develop and manage Processes for performing projects
Process for Custom Application Development with J2EE
Process for Embedded System Development
Process for SOA Governance
Process assets patterns
Standard or reference processes
Enactable project plan templates
Corporate guidelines on compliance
Configure a cohesive process framework customized for my project needs
Create project plan templates for Enactment of process in the context of my project

Figure 2-19 The EPF Approach [65]

Main EPF Composer Features

- EPF Composer stores method contents into knowledge base which allows developers to browse, manage, and deploy. These method contents can be licensed, acquired, and accommodates the user's own content such as, method definitions, whitepapers, guidelines, templates, principles, best practices, internal procedures and regulations, training material, and any other general descriptions of how to develop software.

- EPF Composer provides the facilities required in order to define reusable method content, processes, building blocks, and tools that are used to create project or organization specific processes and methods.
- The elements in the knowledge base can be used for reference and education while building a process. In order to ease the creation of processes, EPF Composer provides catalogs of pre-defined processes for typical project situations that can be adapted to individual needs.
- All content managed in EPF Composer can be published to html and deployed to Web servers for distributed usage.

2.4.2 IBM Rational Method Composer

IBM Rational Method Composer (RMC[7]) is a flexible process management tool built on top of Eclipse platform to provide process authoring, configuration, and publishing capabilities. The main purposes of RMC are: 1) to provide a common management structure for managing process content, and 2) provide developers with the capability of selecting, tailoring and assembling processes for

[7] http://www.ibm.com/software/awdtools/rmc

their concrete development projects. RMC adopted and customized the IBM Rational Unified Process (RUP) process framework as its foundation for building software development processes.

The RUP process framework within RMC includes:

- A *process content library* contains a collection of best practices that are commonly used in RUP projects around the world.
- *Delivery processes* describe a process used to identify what milestones to have in the project, what work products to be delivered by each milestone, and what resources are needed for each phase. These processes can be used out-of-the-box or as a starting point for further customization.
- *Capability patterns* describe a reusable cluster of activities in a common process area that expresses and communicates process knowledge for a key area of interest, such as a discipline. Capability patterns can be used as building blocks to assemble delivery processes or larger capability patterns.

2.4.3 Spemmet

Spemmet [66] is a tool for modeling software processes. The software processes in this tool are described using SPEM meta-model. The tool was developed as a web application and used a shared data storage so that the tool can be available anywhere around the world without installation and supports simultaneous access.

Spemmet offers a set of necessary elements for modeling software processes and it is a flexible modeling solution allows users to use different modeling techniques. However, it does not provide a graphical representation of the process model and only display data in textual form.

2.4.4 APSEE

APSEE [16] is a software framework for software process management which evolved from PROSOFT [67], a formal object-based software development paradigm, to handle the dynamic and evolving characteristic of flexible software process management, process simulation, improvement and reuse. The APSEE software framework has been built based on the APSEE meta-model and APSEE-PML.

The APSEE meta-model is used to describe processes. It includes information about the modeling language components, and is created based on an activity-based paradigm, describing processes as partially ordered collection of activities.

The APSEE-PML is a graphical representation of the process elements described using APSEE meta-model. The main graphical notations in APSEE-PML are summarized by Figure 2-20. As shown in the figure, APSEE-PML includes notations to represent not only fragments, activities and artifacts, but also various connections, such as sequence connection, feedback connection, join connection branch connection and artifact connection.

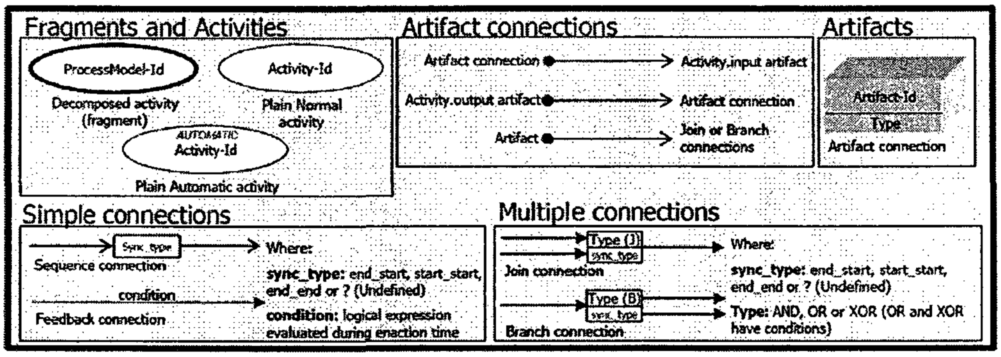

Fragments and Activities
ProcessModel-Id
Decomposed activity (fragment)
Activity-Id
Plain Normal activity
AUTOMATIC
Activity-Id
Plain Automatic activity
Artifact connections
Artifact connection → Activity.input artifact
Activity.output artifact → Artifact connection
Artifact → Join or Branch connections
Artifacts
Artifact-Id
Type
Artifact connection
Simple connections
Sync_type
Sequence connection
condition
Feedback connection
Where:
sync_type: end_start, start_start, end_end or ? (Undefined)
condition: logical expression evaluated during enaction time
Multiple connections
Type (J)
sync_type
Join connection
Type (B)
sync_type
Branch connection
Where:
sync_type: end_start, start_start, end_end or ? (Undefined)
Type: AND, OR or XOR (OR and XOR have conditions)

Figure 2-20 The Notations of the APSEE Meta-model Elements[16]

Chapter 3 Software Maintenance Process Modeller

The goals of this research are 1) to extend the ISO/IEC 14764 maintenance process model to link various resources and knowledge with its process activities and task-steps so that it can provide guidance to maintainers to support their maintenance tasks; 2) to create a meta-model to describe the maintenance process model; 3) to create a set of graphical notations to represent the process model; and 4) All of the above three concepts will be included in a tool environment called Software Maintenance Process Modeller, or SMPM for short.

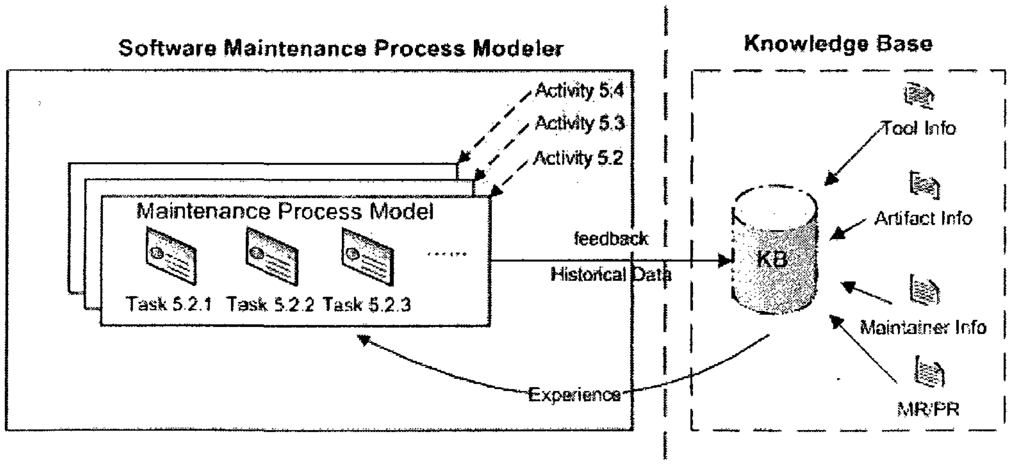


Figure 3-1 The Overview of the Software Maintenance Process Modeller

As shown in Figure 3-1, the SMPM serves as the front-end of the Knowledge

Base (KB). The KB is comprised of ontologies[8], which can be considered as a kind of data storage medium similar to traditional databases. The design of KB has been published in [4], which is out of scope of the presented research. This research mainly focuses on SMPM. In what follows, we will describe in detail the design of the SMPM.

SMPM is built based on a software maintenance process model. We adopt the latest version of IEEE software maintenance process model (ISO/IEC 14764-2006) [9] as the basis for integrating the knowledge and resources in our approach. This will be detailed in Section 3.1.

In order to integrate the software maintenance process model into our SMPM, a meta-model is created to describe the model. Meanwhile a set of notaitons are also created to allow graphically represent the model. Section 3.2 will describe the meta-model and notations.

In practice, a software maintenance task involves many different types of stakeholders all with differing duties and responsibilities. In our approach, these stakeholders are categorized into three different groups: process designer, domain expert and software maintainers. Section 3.3 will describe their duties and responsibilities.

This is followed in Section 3.4 with the description of Context-sensitive support

[8] Ontology http://en.wikipedia.org/wiki/Ontology

in our SMPM tool environment.

SMPM generates and collects historical data while it is runing. Maintainers gain a lot of valuable experience (best practices) in their daily practice. This historical data and experience can be used to enrich the underlying KB, and can be shared with other maintainers. Therefore, SMPM needs to provide an interface to collect these historical data and experience. This will be described in detail in Section 3.5.

Finally, Section 3.6 describes the integration of the software maintenance process model into the SMPM tool environment.

3.1 Maintenance Process Model

Activities are an integrated part of any software maintenance task and require the integration and sharing of available knowledge within the process. Based on this observation, we have decided to adopt the latest version of IEEE software maintenance process model (ISO/IEC 14764-2006) [9] as the basis for integrating the knowledge and resources in our approach. The IEEE standard describes a software maintenance process, which defines activities, tasks and task-steps that are necessary to perform software maintenance tasks. However, it does not specify the details of how to implement or perform these activities, tasks and task-steps included in a process.

Furthermore, a process model is commonly used to specify how a process should be operated over time and which resources should be allocated or involved in its activities. Marshak [68] believes that modeling can be considered as a process of knowledge acquisition about the target business operations. Based on this observation, a maintenance process model should concretize activities, information, and flow, which are embedded in the software maintenance domain, into maintenance tasks with explicit reference context such as organizations, artefacts, tools and maintainers. Although, the IEEE standard mentions that artefacts should be involved in particular activities, it does not give the solution of how these artefacts are related to specific tasks or task-steps as the process is performing.

In this section, we will describe the maintenance process model adoption (Section 3.1.1). This is followed by a section describing how to equip our process model with execution ability (Section 3.1.2). Finally, we will discuss the process flow in Section 3.1.3.

3.1.1 Maintenance Process Model Adoption

As the basis for our software maintenance process model we adopt the IEEE Software Maintenance process model [9], which describes in general the various activities, task-steps to be performed towards a maintenance request. This IEEE Standard can be divided into six sub activities: *process implementation*, *problem and modification analysis*, *modification implementation*, *maintenance review and* *acceptance*, *migration* and *retirement.* In this research, we will limit our scope and focus only on those activities which are most closely related to software comprehension aspects of a maintenance task, e.g. *problem modification analysis, modification implementation, maintenance review/acceptance, and migration.*

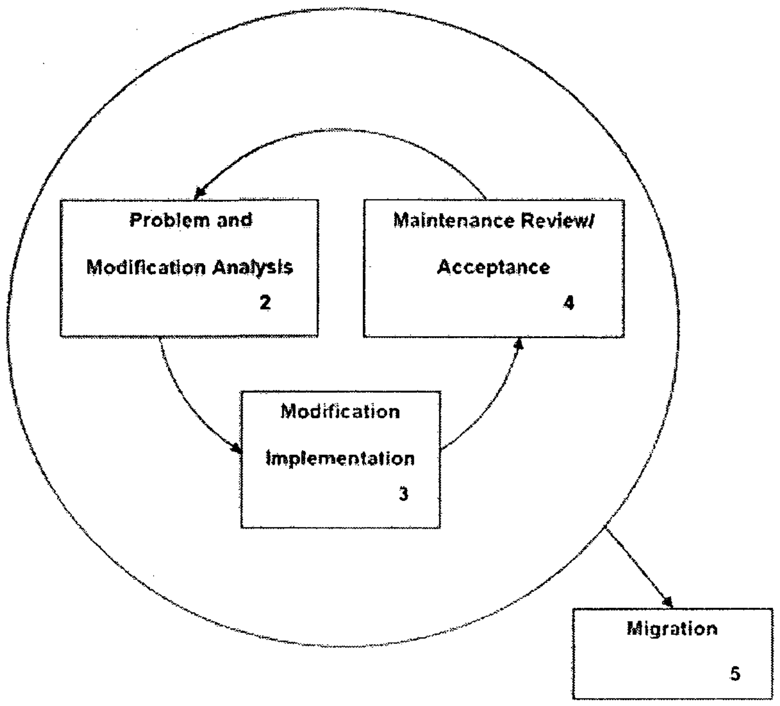


Figure 3-2 The Tailored Software Maintenance Process Model

Figure 3-2 illustrates such a tailored process model used as the basis for the knowledge and resources integration in our approach. The process starts with the activity of *problem and modification analysis*, followed by *modification implementation*, and then *maintenance review/acceptance*. The process completes with the software product *migration*. The arrows between *problem and modification analysis*, *modification implementation*, and *maintenance review/acceptance* indicate that these three activities can be executed iteratively if necessary.

3.1.2 Process Execution Ability

Most of the nowadays process models can be executed. For example, as discussed in Section 2.3.1, a business process model described using BPEL **[21]** may contain *invoke activity* to invoke a remote partner, or contain *receive activity* to receive information from another partner. A process described using SPARK may contain an *executor* to perform actions **[46]**.

In the presented approach, we enhance the process model to have such execution ability by using *query* elements. A *query* is a new process element we add to the maintenance process model. It is linked to a *task-step*. A *task-step* may contain zero or more *queries*, while a *query* may be contained by multiple *task-steps*. The use of *query* equips the process model with ability to extract information from the underlying KB. A *query* element represents 1) a piece of query statements writing using Description Logic language, or DL-language[9] for short. A DL-language (DL), a knowledge representation formalism, is used as a standard ontology language **[69,70]**. The design of DL-based queries is out of the scope of the presented research. Our *query* elements link with a collection of existing DL-based query statements which have been designed by another research team. We are not going into detail of this part; and 2) the results of the execution of the DL-based queries. Usually, the relationships

[9]. Description Logic Language http://en.wikipedia.org/wiki/Description_logic

between the process activities including their task-steps and the resources such as tools, artefacts, maintainers are explicitly specified in the KB [4]. The execution of the DL-based queries returns the resources that are associated with the relevant process *task-step*, which create a link between the process model and the resources. The execution results may provide guidance for maintainers to perform the containing *task-step.*

3.1.3 Process Flow

The original IEEE standard maintenance process only lists *task-steps* to be performed. It does not suggest which *task-steps* are required, or which ones are not. However, we realize that not all of the *task-steps* suggested in the IEEE standard process are necessary to be performed to complete a maintenance task. Therefore, our tailored process model should have ability to allow maintainers to determine if a *task-step* should be executed or not based on the process context they encounter. The same consideration is also applied to queries. Based on this observation, we classify *task-steps* and *queries* into two categories: *mandatory* and *optional.* A *mandatory task-step/query* must be completed in order to being able to continue/complete the process. In contrast an *optional task-step/query* does not have to be executed. It is the maintainer's choice to decide if it is executed or not. Before the process flow can continue with the next process activity, all of the *mandatory task-steps* and *queries* in the current activity must be completed.

Figure 3-3 depicts a sample process showing how the *mandatory/optional task-steps* affect the process flow. The process contains two tasks: *Task A* and *Task B*. All of the steps included in these tasks are *mandatory* except *Step 2*. The arrowhead with solid line represents the process flow. To complete such a simple process, *Task A* and *Task B* have to be performed in order. To finish *Task A*, *Step 1* and *Step 3* must be executed because they are *mandatory*. But *Step 2* is not required to be executed, because it is an *optional* step. The arrowhead with dash line indicates the optional process flow. The process flow cannot move to *Task B* until all the *mandatory* steps in *Task A* are executed. In this example, *Step 1* and *Step 3* must be executed to make the process move to *Task B*. And finally *Step 4*, *Step 5* and *Step 6* must be orderly executed to finish *Task B* because all of them are *mandatory*.

The classification of *task-steps* into *mandatory* and *optional task-steps* implements an algorithm similar to the *pick* activity in BPEL. It provides choices to maintainers to pick one path from multiple possible paths. Taking Figure 3-3 as an example, making *step 2* as optional creates two possible paths which are Step1→Step2→Step3 path and Step1→Step3 path. Maintainers must pick either of them to continue the process flow. This approach is very efficient when several *optional task-steps* continuously linked together. Imagine one *optional task-step* pedicts two possible process flow paths. However, two *optional task-steps* linked continuously pedicts four possible process flow paths. By using this approach, we can describe multiple process flows and still keep the diagram clear and neat.

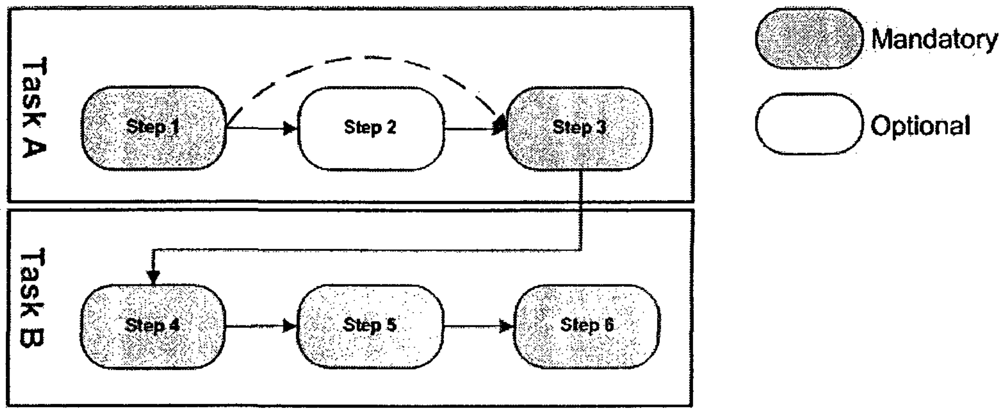


Figure 3-3 A Sample Process Model

3.2 Process Meta-model and Graphical Notations

In Section 3.1, we have introduced the overall process model being adopted and the enhancement made to the process model. In what follows, we will focus on designing a metam-model to describe the process model, which is detailed in Section 3.2.1. A set of notations used to graphically represent the process model will be described in Section 3.2.2. This is followed by Section 3.2.3 with the discussion of layout strategy implemented in the SMPM.

3.2.1 Process Meta-model

The presented process meta-model is used to describe our tailored software maintenance process model described in Section 3.1. In the presented approach, we categorize the process model elements into four categories:

- Basic model element – The basic elements are the basic building blocks used to construct maintenance processes. Most of these elements directly derive from the original IEEE standard;

- Execution model element – It is the *query* element that has been introduced in Section 3.1.2;
- Connecting element – It is used to represent the process flow; and
- Swimlane – It is a layout mechanism used to organize process elements in a process diagram.

In the rest of this section, *basic model element*, *execution model element* and *connecting element* will be presented. And *Swimlane* will be discussed in detail in Section 3.2.3.

Basic Model Element

In our meta-model, *basic model elements* are the basic building blocks used to construct maintenance processes. Most of them are derived from the original IEEE standard process. The following describes these process elements.

A *StartPoint* presents the start of the maintenance process. As illustrated in Figure 3-4, every process diagram should contain one *StartPoint*.

An *EndPoint* indicates the end of a maintenance processs. Every process diagram can only have one *EndPoint* as indicated in Figure 3-4.

An *activity* element models maintenance activities described in the original IEEE standard. According the discussion in Section 3.1.1, there are four activities have been included in the tailored process model, therefore, we only model these four activities in the meta-model, which are *problem modification analysis*, *modification implementation*, *maintenance review/acceptance*, and *migration*. Figure 3-4 shows that a diagram can have one or more *activity* elements.

A *task* describes a specific aspect of concerns that maintainers need to consider while performing an *activity*. As shown in Figure 3-4, an *activity* element can contain one more *task* elements.

A *task-step* is one of the refinements of its containing *task* element. Each of these *task-steps* is an example of what must be performed in order to complete the maintenance process. A *task* element contains zero or more *task-steps* as indicated by Figure 3-4.

A *diagram* represents a software maintenance process. It contains various process model elements that are described in the meta-model.

Execution Model Element

An *execution model element* is an element used to describe the behavior aspect of the maintenance process. Specific to the presented meta-model, there is only one model element belongs to this category, which is the *query*. A *query* element is used to extract information from the KB as described in Section 3.1.2. Figure 3-4 shows that *query* elements are contained by *task-steps*. A *task-step* element can have zero or more *query* elements, each of which specified the details of how to perform its containing *task-step* in order to complete a maintenance process.

A *query* element must be in one of three states: *unexecuted*, *successful* and *failed.* The *unexecuted* state indicates that a *query* has not been executed. The *successful* state means that a *query* has been successfully executed. And the *failed* state implies a *query* has been executed but failed.

Connecting Element

A *connecting element* is used to connect *basic model elements* together in a diagram to create the basic skeletal structure of a software maintenance process. There is only one *connecting element* available in the meta-model which is *connection*. A *connection* element represents the process flow in a maintenance process model.

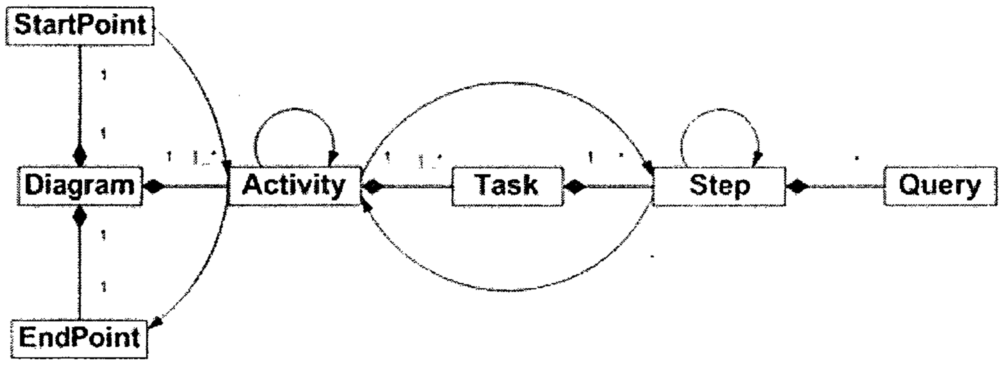


Figure 3-4 The Maintenance Process Meta-model

In Figure 3-4, process flow is represented using an arrow with solid line. According to Figure 3-4, a *SartPoint* is the start of a process. It connects to an *activity*.

An *EndPoint* is the ending of a process, which accepts a *connection* from an *activity*. An *activity* may connect to another *activity*, while *activities* and *task-steps* can connect each other. And a *task-step* may connect to another *task-step.*

A meta-model is a conceptual framework made of rules and elements that allows building models. It contains all concepts and relations included in the subject under study. Therefore, the presented meta-model can be used to create software maintenance process models.

In this research, maintenance process models created using the presented meta-model are further classified into three kinds. They are: *process template*, *domain-specific model* and *process instance.*

A *process template* is a general process model created using the presented meta-model. Usually, a *process template* is a general maintenance solution for maintaining various types of maintenance tasks in an organization. A *domain-specific model* is a process model refined and customized from a *process template* in order to meet specific needs of a particular domain. A *process instance* is a copy of a *domain-specific model.* A *process instance* is a process model ready to be navigated and executed to help maintainers to complete their particular maintenance tasks.

3.2.2 Graphical Notations

The process meta-model has been presented in Section 3.2.1. In the next step we create a set of graphical notations so that process model elements can be mapped onto them. We describe these notations in the rest of this section.

A *StartPoint* (Figure 3-7) is represented as two concentric circles filled with grey color in the middle.

An *EndPoint* (Figure 3-8) is represented as two concentric circles filled with black color in the middle.

As shown in Figure 3-5, an *activity* is represented using a swimming pool with a vertical anchor bar at the left side with two anchors at each end of that anchor bar. The top anchor is a target point that its previous *activity* connects to. This top anchor is also the source point that the *tasks-steps* within the *activity* start from. On the other hand, the bottom anchor is the source point that its next *activity* connects from. It is also the target point that ends the process flow within the current *activity*.

Figure 3-6 depicts a *task* notation, it is presented as an unfilled horizontal bar with a task title on the left side of the bar. Actually, a *task* notation cannot stand alone in a process diagram. Instead, they must be embedded into *activity* notation. We might imagine *task* notations as lanes which are used to constructed a swimming pool which is an *activity* notation.

A *task-step* notation is represented as either Figure 3-9, an unfilled rectangle with bold line as its frame representing a *mandatory task-step*, or Figure 3-10, an unfilled rectangle with regular line as its frame representing an *optional task-step.*

The notation of a *query* varies as shown in Figure 3-11 to Figure 3-16. . A square fully filled in grey (Figure 3-12) represents an *unexecuted mandatory query*, and a square partly filled in grey (Figure 3-11) represents an *unexecuted optional query.* The color is meaningful to *query* notations. Grey color indicates a *query* has not been executed while green color indicates a *query* has been successfully executed. Finally, a red color indicates that a *query* has been executed but failed.

As shown in Figure 3-17, the notation of the *connection* element is represented as an arrow with solid line.

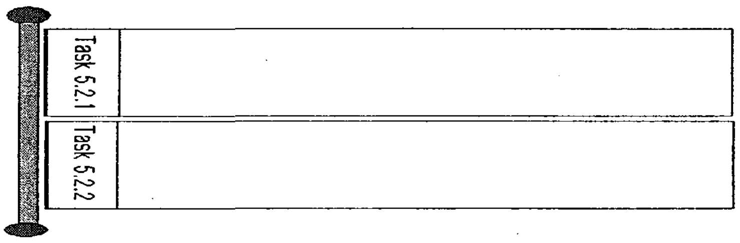


Figure 3-5 Activity Notation of SMPM

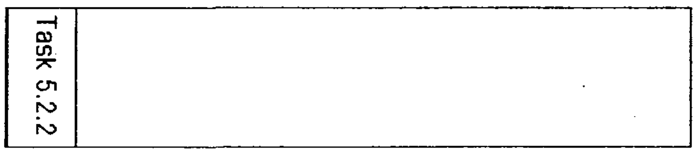


Figure 3-6 Task Notation of SMPM

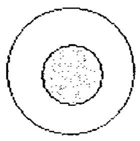


Figure 3-7 StartPoint Notation of SMPM

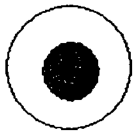


Figure 3-8 EndPoint Notation of SMPM

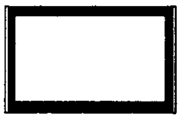


Figure 3-9 Mandatory Task-step Notation of SMPM

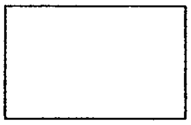


Figure 3-10 Optional Task-step Notation of SMPM

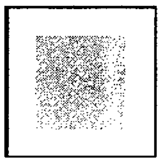


Figure 3-11 Unexecuted Optional Query Notation of SMPM (Grey)

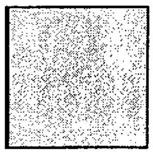


Figure 3-12 Unexecuted Mandatory Query Notation of SMPM (Grey)

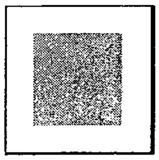


Figure 3-13 Failed Optional Query Notation of SMPM (Red)

Figure 3-14 Successful Optional Query Noation of SMPM (Green)

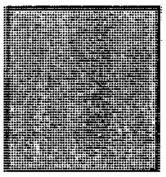


Figure 3-15 Failed Mandatory Query Notation of SMPM (Red)

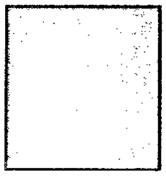


Figure 3-16 Successful Mandatory Query Notation of SMPM (Green)

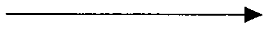


Figure 3-17 Connection Notation of SMPM

3.2.3 Layout Strategy

One of the challenges in process visualization is how to organize the process elements in an appropriate manner in a diagram. This is the work that layout strategy should do. We adopt a swimlane [24,23] mechanism which is popular in BPMN and UML as the layout strategy in our SMPM.

In the SMPM, the swimlane layout arranges process elements into a set of rows called lanes. The attributes of each process element determines which row (lane) it is placed. In our swimlane layout, a lane represents a *task*. The lane header, the vertical bar with label on the left hand side, is the *task* title. Multiple related lanes (implied from their lane headers) construct a pool, which is an *activity* (Figure 3-20). *Task-steps* are embedded into lanes where they belong (Figure 3-19). No *task-step* can be in more than one lane.

Query elements are rendered as small squares linked to a *task-step* where they reside (Figure 3-18). All of the *basic model elements* are connected using *connection* elements. The process element where the *connection* starts from is called the source node. Along with the arrow head the *connection* ends with a target node. To avoid the edge crossings, y-axis offset of a *task-step,* inside the same *activity*, is arranged at the right side of its preceding *task-step*, regardless the *task* to which it belongs. As shown in Figure 3-20, this layout strategy can eliminate edge crossing inside an *activity*.

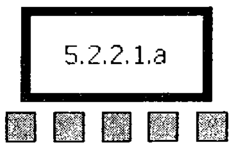


Figure 3-18 Task-step Layout Strategy

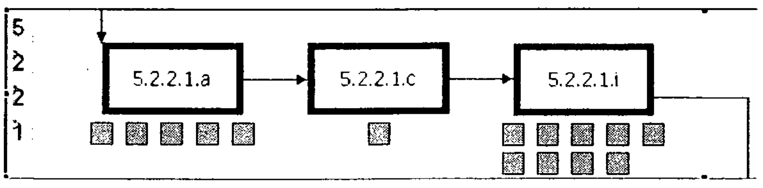


Figure 3-19 Task Layout Strategy

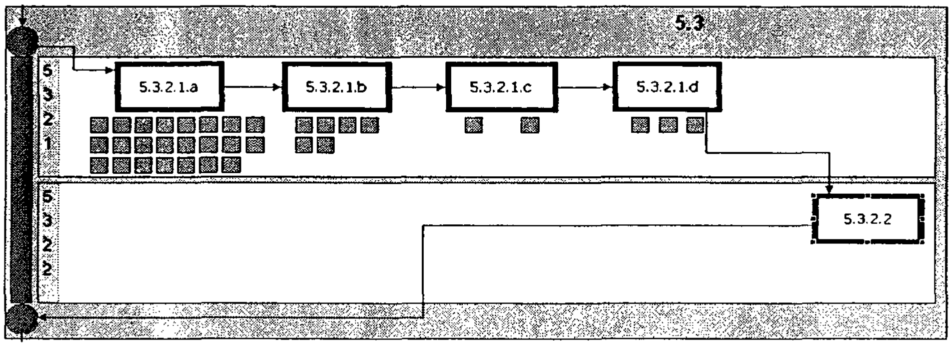


Figure 3-20 Activity Layout Strategy

3.3 Stakeholders and Their Roles

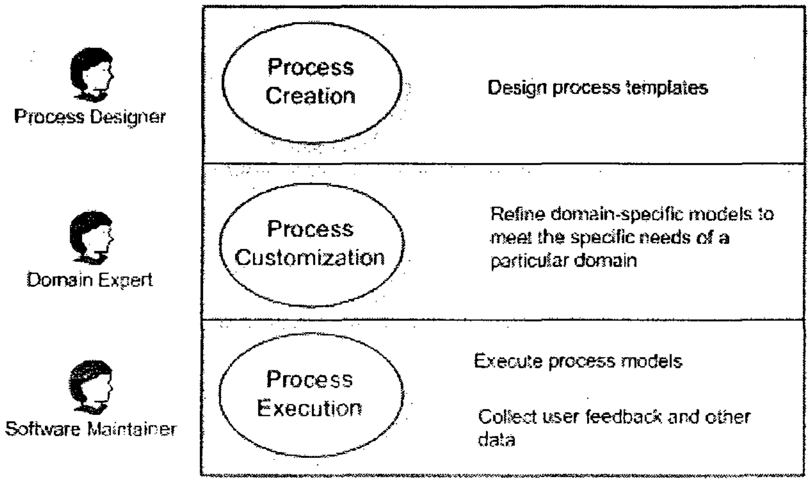


Figure 3-21 User Roles and Their Responsibilities

In this research, the process visualization is considered as a central information hub for all process related information. In particular, different stakeholders (e.g. process designers, domain experts, and software maintainers) may access the environment for different purposes depending on their particular role in an organization. In what follows we introduce three different types of stakeholders, each with a specific role in the organizational context. The user role defines the behaviour, responsibilities and knowledge of individuals involved in a maintenance task.

The stakeholders of our SMPM can be categorized into three different roles as shown in Figure 3-21.

Process Designer Role

Our research makes assumption that stakeholders associated with a *process designer role* are usually senior software maintainers. They are responsible for designing *process templates* based on the organization's maintenance plans and strategies. As discussed in Section 3.2.1, a *process template* is a general maintenance solution suitable for different types of maintenance tasks within an organization. That means a *process template* should not involve any domain specific content.

Domain Expert Role

Domain experts have rich experience and knowledge in their specific field. They are responsible for creating *domain-specific models* from *process templates*, and refining and customizing the *domain-specific models* to meet specific needs of the domain.

In a *domain-specific model*, domain experts can customize some of its attributes. The following summarizes the major attributes that can be changed by domain experts.

- Change the *execution type*: *Execution type* describes whether a process element must be executed or not. As described in Section 3.1.3, the value of an *execution type* can be either *mandatory* or *optional*. This attribute is available to *task-steps* and *queries* elements.
- Change the *context level*: *Context level* defines the constraints when maintainers retrieve knowledge from the KB. There are four *context levels* defined in our process model which will be further discussed in section 3.4. A *context level* can be applied to *query*, *task-step* and *task* elements.
- Management of pre-defined *queries*: *Queries* are process elements used to retrieve information from their underlying KB. Domain experts are allowed to create new *queries*, add *queries* to and/or remove *queries* from a *domain-specific model*.

Software Maintainer Role

Software Maintainers are typically the end users of the process models. They are responsible for executing *process instances* to complete their maintenance tasks. They also represent the main source for collecting feedback resulting from their experience during the execution.

3.4 Establishing Context-sensitive Supports

In this section, we will describe the establishment of context-sensitive support in the SMPM.

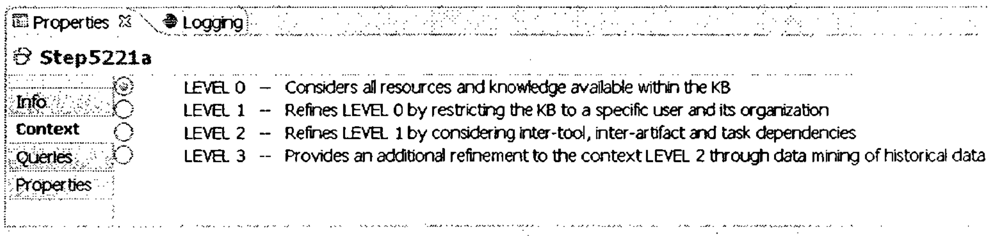
As Rilling et al described in [4], a context-sensitive process can generally be described as the steps involved in identifying any information that might be relevant to characterize the situation of an entity. An entity is typically a knowledge resource that is considered relevant to the interaction between a user and a process. Based on these assumptions, a process can be called context-sensitive if it establishes a relationship between relevant information resources, users and organizational factors that have a direct/indirect impact on a specific task.

Context Level	Description
Level 0	Considers all resources and knowledge available within the KB.
Level 1	Refines level 0 by restricting the KB to a specific user and its organization.
Level 2	Refines context level 1 by considering inter-tool, inter-artifact and task dependencies.
Level 3	Provides an additional refinement to the context level 2 through data mining of historical data collected from previous tasks.

Table 3-1 Context Level Description

Rilling et al [4] divide the context-sensitive into four levels as depicted in the Table 3-1.

As shown in Table 3-1, a process element with *context level 0* is considered no constraints associated with the supporting queries. Meaning all of the resources and knowledge available will be considered during knowledge retrieval. A process with *context level 1* is restricted in scope to only those resources and knowledge relevant to specific users and/or organizations. The most restrictive constraints are imposed to the process with *context level 3*. The inter-resources relationships and information from historical data are important factors for creating the constraints for the processes at this context level.



[✓ LEVEL 0 – Considers all resources and knowledge available within the KB]
LEVEL 1 – Refines LEVEL 0 by restricting the KB to a specific user and its organization
LEVEL 2 – Refines LEVEL 1 by considering inter-tool, inter-artifact and task dependencies
LEVEL 3 – Provides an additional refinement to the context LEVEL 2 through data mining of historical data

Figure 3-22 Implementation of Context-sensitive Support

Our SMPM supports context-sensitive by providing a *context configuration view* to allow change the *context-sensitive level* of process elemenets. Figure 3-22 shows the *context configuration view* in the SMPM. Domain experts can use this view to customize the level of context-sensitive support of process elements. By default, all process elements are set to *context level 0*. Once the *context-sensitive level* of a process element is changed, the scope of information retrieving from the KB will be changed accordingly. Actually, the major portion of work regarding context-sensitive support is done within the back-end KB which is not included in the presented research. The presented research mainly focus on providing a GUI to allow configure the *context-sensitive level* for each process element.

3.5 Data/User Experience Collection

The SMPM will generate various historical data and best practices when it runs. Moreover, maintainers may give feedback and comment when they use the SMPM to help complete a maintenance task. This historical data, best practices and feedback are fundamental to the KB. If this data is not collected, it will be lost. In this section, we will describe how the SMPM supports the collection of historical data / best practices and user feedback.

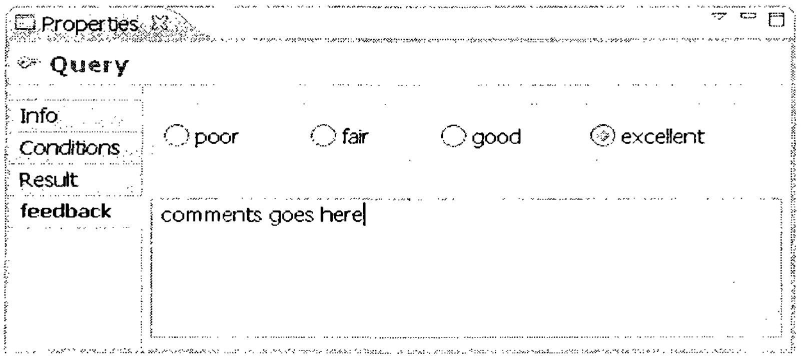
As shown in Figure 3-1, the SMPM will collect various types of historical data/best practices and user feedback while it is running. This information is supposed to be stored in the KB. However, at the time we were developing the SMPM, the team responsible for developing the KB did not deliver APIs to allow "write" operation. As a trade-off, we temperately store this information in the local drive as a regular file. Once the "write" APIs are available, this information will eventually be stored into the KB.

In the SMPM, historical data such as which maintainer uses the tool, when do they use it, which tasks and task-steps have been executed, what is the execution order, etc., are automatically captured and stored.

In the SMPM, we also provide *feedback view*to allow collect user feedback. The *feedback view* (Figure 3-23) includes a comment section and a ranking section which are manually input by *software maintainers*. As listed in Table 3-2, there are 4 ranking levels ranging from "poor" to "excellent". The user selects the ranking based on the support the tool provides during a particular process context.

Level	**Description**
Poor	A process element has no or very limit contribution to the entire process
Fair	A process element has minor contribution to the entire process
Good	A process element has major contribution to the entire process
Excellent	A process element has significant contribution to the entire process

Table 3-2 Feedback Ranking Level



poor fair good [✓ excellent]

comments goes here

Figure 3-23 Design of Feeback View

3.6 Tool Integration

In the previous sections, we have described various aspects of our process model and the techniques for linking the resources and knowledge to the process model elements. In what follows, we will discuss how these various pieces can be integrated in our tool environment.

The advantage of a graphical visualization is the ability to represent semantic rich data at varying abstraction levels. As discussed in Section 2.4.2, the *overview and detail* visualization technique is suitable for visualizing large amount of information. More importantly, it provides context support during the navigation of the information space. Based on this observation, we designed to apply the *overview and detail* technique to our SMPM.

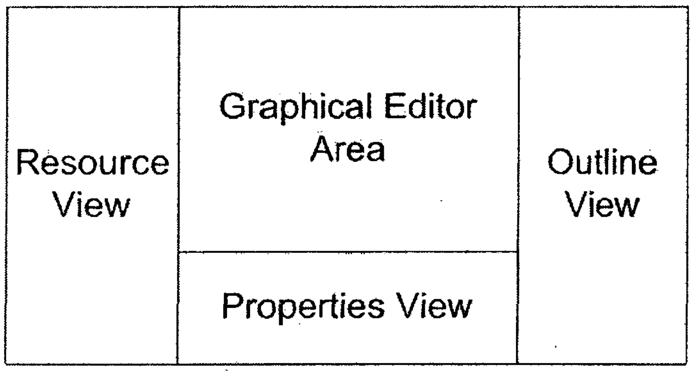


Figure 3-24 The GUI Outline of SMPM

With these considerations in mind, we introduce a visualization approach that incorporates multiple views as shown in Figure 3-24. They are *resource view*, *outline view*, *properties view* and *graphical editor area*. These views are integrated to allow for rendering and organizing different resources and knowledge relevant to a specific software maintenance task.

Resource view: The *resource view* (Figure 3-25) organizes resources relevant to a process model in a hierarchical structure. These resources include *process templates*, *domain-specific models*, *process instances*, users, tools and artifacts. From here, we can create new *process templates*, open *domain-specific mdoel*for editing or select *process instances* for navigation and execution.

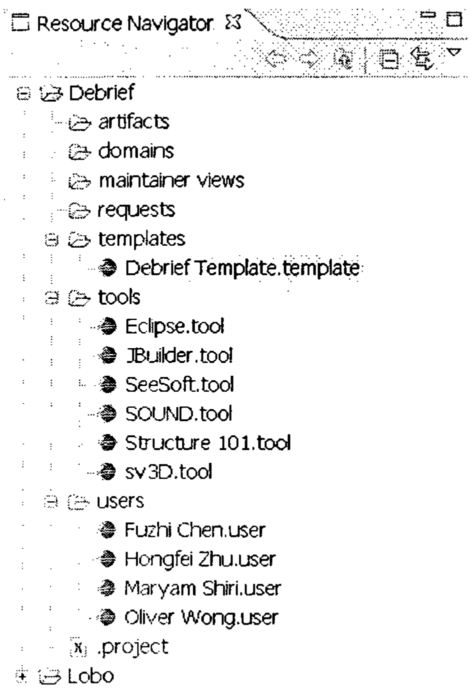
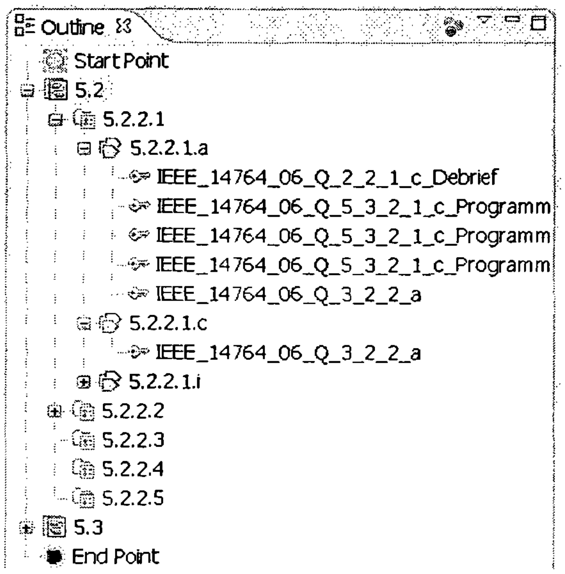

```
Resource Navigator
Debrief
    artifacts
    domains
    maintainer views
    requests
    templates
        DebriefTemplate.template
    tools
        Eclipse.tool
        JBuilder.tool
        SeeSoft.tool
        SOUND.tool
        Structure 101.tool
        sv3D.tool
    users
        Fuzhi Chen.user
        Hongfei Zhu.user
        Maryam Shiri.user
        Oliver Wong.user
    .project
Lobo
```

Figure 3-25 The Resource View of SMPM

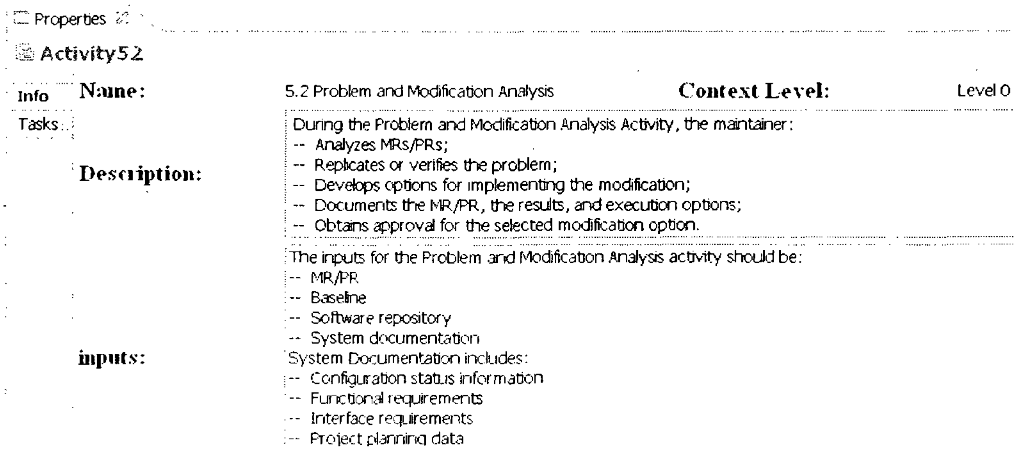
Outline view: The *outline view* (Figure 3-26) displays the process model that is currently under editing in a tree hierarchy.



```
Outline
  Start Point
⊟ 5.2
  ⊟ 5.2.2.1
    ⊟ 5.2.2.1.a
        IEEE_14764_06_Q_2_2_1_c_Debrief
        IEEE_14764_06_Q_5_3_2_1_c_Programm
        IEEE_14764_06_Q_5_3_2_1_c_Programm
        IEEE_14764_06_Q_5_3_2_1_c_Programm
        IEEE_14764_06_Q_3_2_2_a
    ⊟ 5.2.2.1.c
        IEEE_14764_06_Q_3_2_2_a
    ⊞ 5.2.2.1.i
  ⊞ 5.2.2.2
    5.2.2.3
    5.2.2.4
    5.2.2.5
⊞ 5.3
  End Point
```

Figure 3-26 The Outline View of SMPM

Properties view: The *properties view* (Figure 3-27) displays detailed information of a selected process element in either the *graphical editor area* or in the *outline view*. The *properties view* is also a place for *domain experts* to customize *domain-specific models*. This has been discussed in Section 3.3.



Properties

Activity52

Info

Tasks

Name:	5.2 Problem and Modification Analysis	**Context Level:**	Level 0
Description:	During the Problem and Modification Analysis Activity, the maintainer: -- Analyzes MRs/PRs; -- Replicates or verifies the problem; -- Develops options for implementing the modification; -- Documents the MR/PR, the results, and execution options; -- Obtains approval for the selected modification option.		
inputs:	The inputs for the Problem and Modification Analysis activity should be: -- MR/PR -- Baseline -- Software repository -- System documentation System Documentation includes: -- Configuration status information -- Functional requirements -- Interface requirements -- Project planning data		

Figure 3-27 The Properties View of SMPM

Graphical Editor Area: The *graphical editor area* (Figure 3-28) is the central area of the entire GUI. It is used to graphically display software maintenance process models. Depending on the stakeholder using the system, different functionalities are available. The *graphical editor area* provides: 1) *process designers* with a place to visually construct *process templates*; 2) *domain experts* with a means to customize the *domain-specific models*; and 3) *software maintainers* with an interface to execute the *process instances* and collect the feedback.

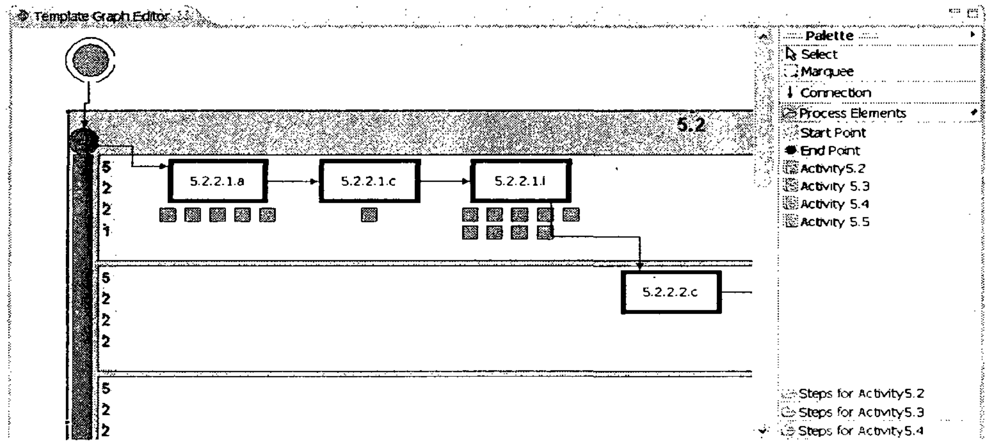


Figure 3-28 The Graphical Editor Area of SMPM

The views described above are organized into perspectives, where a perspective defines a collection of views, their layout, and applicable actions available to a specific stakeholder. As discussed earlier (Section 3.3), no visualization method addresses all the needs of the users, however, the creation of multiple views or perspectives has been shown to be useful in many circumstances. [71,72]. Each view typically focuses on the visualization of information to a given task, or user context by combining different levels of abstraction. Given the fact that in our SMPM tool there are three distinct groups of users of our system, we therefore provide three distinct perspectives. They are *process designer perspective*, *domain expert perspective*, and *software maintainer perspective*.

Figure 3-25 and Figure 3-26 show the *resource view* and *outline view* provided in the STMM tool. Both views are based on a tree structure. Tree structures are a common way of visualizing the hierarchical structure of information [73,74]. In a tree structure, process model elements are represented as nodes, while visual links (lines) in front of nodes visualize the parent-child relationships of these process elements. By applying indentation and meaningful icon to each tree structure item, combining with the "expand-collapse" feature that a tree structure inherently has, a tree structure provides the concept of *level of abstraction*. Furthermore, selecting an item in the *outline view* will also synchronize *with the corresponding element* in the *graphical editor area* and vice versa. Combining the graphical visualization with the tree structure, it provides users with an *overview and detail* approach of visualization. The *properties view* (Figure 3-27) provides detailed information about the selected element in either the *graphical editor area* or *outline view*. The *properties view* provides therefore a *details-on-demand* functionality to our SMPM.

Chapter 4 Implementation

In the previous chapter, we introduced the theoretical and design aspects of our software maintenance process modeller (SMPM). In this chapter we discuss implementation details of the actual tool.

We will first describe the general requirements for implementing this tool (Section 4.1), followed by a description of the overall structure of the system (Section 4.2). And finally we will provide details on the system implementation (Section 4.3).

4.1 System Requirements

Based on the discussion in Chapter 3, we can now derive implementation requirements for our SMPM.

- Requirement #1: Software maintenance process meta-model -- One of the major contributions of this research is to create a meta-model to describe the software maintenance process model proposed in Section 3.1. This will be discussed in detail in Section 4.3.1.
- Requirement #2: Graphical notations -- Graphical representation of process model is considered as another major contribution of this research. Process model elements need to be graphically represented in order to visualize them in our SMPM. This will be further described in Section 4.3.2.
- Requirement #3: Mapping process model elements to their graphical notations

– Basically, in our SMPM, process model elements have one on one relationship with their visual counterparts. It will be further discussed in Section 4.3.3.

4.2 System Overview

In order to enable our process model to provide maintainers with guidance while performing maintenance tasks, we have developed this SMPM tool. Actually, SMPM is not a single program; it is a collection of Eclipse plug-ins[10] and application programs, as shown in Figure 4-1.

Among these Eclipse plug-ins, *process modeller plug-in* is the most important one and it is built on top of the Eclipse Graphical Editing Framework (GEF[11]). The *process modeller plug-in* allows for the visual manipulation of the user defined process models. Another Eclipse plug-in called *navigation tree plug-in* is also developed to illustrate the process models in a tree structure view instead of a graphical view. The reason we need this *navigation tree plug-in* is because it can be conveniently embedded into other developing workbench, such as *Java development perspective*, in order to provide support without losing the context when switching between perspectives. Both the *process modeller plug-in* and *navigation tree plug-in* are created based on the same process meta-model.

[10] Eclipse http://www.eclipse.org

[11] GEF Http://www.eclipse.org/gef

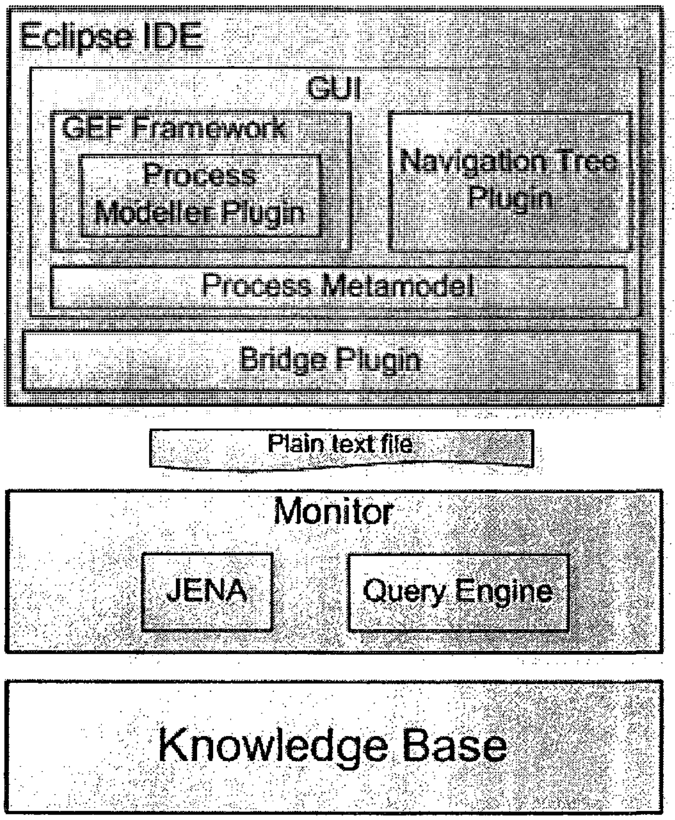


Figure 4-1 System Overview

As shown in Figure 4-1, at the bottom of the figure, there is a KB which is a repository for storing information and knowledge. On top of this KB there is JENA[12], a Java based ontology API toolkit. It allows maintainers to access the KB using Java. At the time we were developing this SMPM, JENA did not support Eclipse Plug-in Development Environment (PDE) so we are not able to directly utilize the JENA toolkit within an Eclipse plug-in. In order to apply JENA to our SMPM tool, a *bridge plug-in* is created to communicate between Eclipse plug-ins and the JENA toolkit. This *bridge plug-in* is a proxy application of JENA in Eclipse PDE. Therefore, if

[12] JENA http://jena.sourceforge.net/ontology

SMPM needs to communicate with JENA, it will directly send the requests to the *bridge plug-in* and then get the result from it. On the other end there is a *monitor* which is a daemon application responsible for executing commands according to the requests received from *bridge plug-in* and then sending the results back to the *bridge plug-in*. Actually this *monitor* is a wrapper program. It executes commands by calling the real JENA API. Plain text files are used to exchange information between *bridge plug-in* and *monitor* daemon program.

4.3 System Implementation

The Figure 4-2 shows the architectural structure of the SMPM. From the figure we can see that the entire system is actually designed using a MVC (Model-View-Controller) design pattern.

- *Model*: The process meta-model acts as the "Model" part in the MVC. It is responsible for describing the maintenance process model elements and their relationships.
- *View*: The graphical representation (notation) is the "View" part in the MVC. The "View" is the graphical and textual elements that are used to visualize the maintenance process model elements.
- *Controller*: The controller is responsible for mapping "Models" to their visual counterparts ("Views"). In the GEF framework, the controller is called EditPart. We will use this term in the rest of the section.

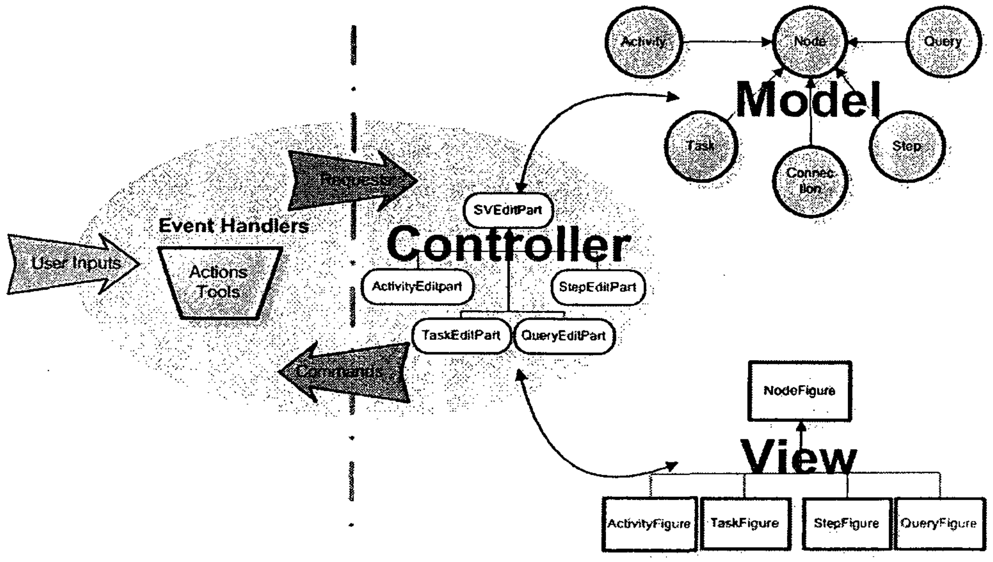


Figure 4-2 System Architectural Structure

4.3.1 Process Meta-model

Software maintenance process model elements are modeled as a set of Java classes. Each of these Java classes contains attributes to describe the properties of process model elements and their relationships. The class diagram (Figure 4-3) illustrates how our maintenance process meta-model is represented using Java classes.

Figure 4-3 shows that the *ModelElement* class (abstract), which is an atomic constituent of the model, is the top super-class in the design hierarchy. It not only provides persistence support to process model elements so that the process models can be saved to a hard disk and reloaded later, but also a notification mechanism used to listen to property change of process model elements and then send notifications to objects which are concerned.

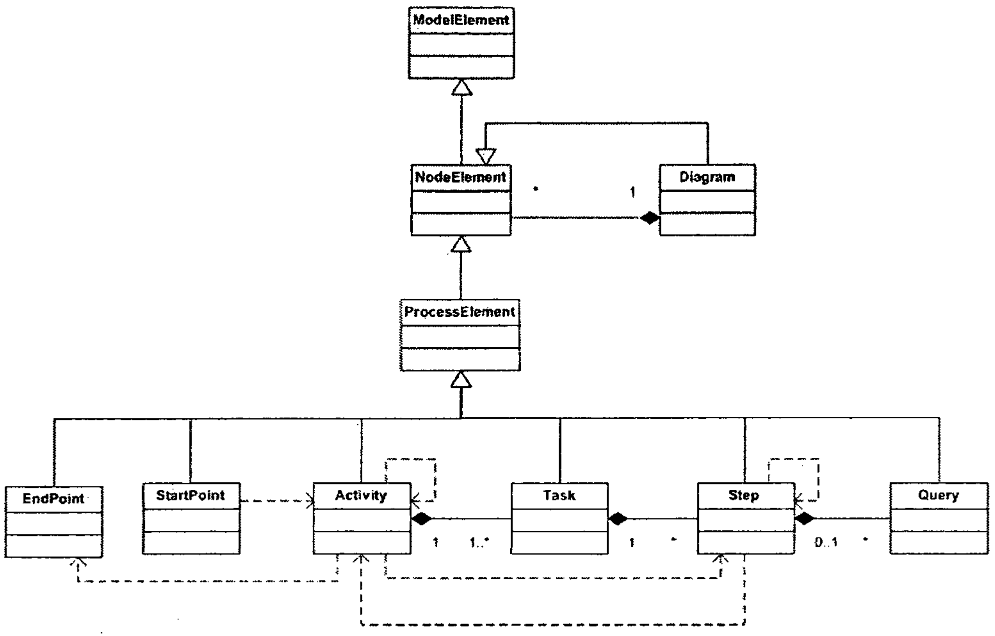


Figure 4-3 Maintenance Process Meta-model Class Diagram

NodeElement class (abstract) is the base class that all process model elements (e.g. *basic model elements*, *execution model element* and *connecting element*) should inherit. It contains some common attributes that all process model elements share (such as, size, location etc.). Actually, all process model elements are either direct or indirect subclasses of *NodeElement*.

ProcessElement class is the super-class of all process model elements. It further describes common properties that all process model elements have (such as element id, name, predecessor and successor elements). All of the process model element classes: *Activity*, *Task*, *Step* and *Query* directly inherit it.

Activity class is the super class of all the *activity* process model element classes. An *Activity* class can have *Task* class instances as its containment elements. An

Activity can have association relations with itself, *StartPoint* and/or *EndPoint* process model elements.

Task class models the *task* process model elements. It contains attributes to describe a *task* (such as description, containing *step* elements, ranking level, feedback, etc.). A *Task* class may take *Step* class instances as its containment elements.

Step class is the parent class of all the *task-step* process model elements. It contains attributes to depict a *task-step* (for example its selected *queries*, available *queries*, context level, execution type etc.). A *Step* class has association relationships to itself and/or the *Activity* in which it is contained.

Query class models the *query* process model elements. This class has attributes necessary to describe a *query*, such as query condition, results, query status etc.

StartPoint and *EndPoint* classes represent the *StartPoint* and *EndPoint* process model elements respectively.

Diagram class represents the *diagram* process model element which consists of a collection of process model elements and their relationships.

Figure 4-4 illustrates the *connection* class diagram. Process flow is represented using *Connection* which is a Java class with two attributes: source element and target element. These attributes represent the source where the process flow starts from, and target where the process flow goes.

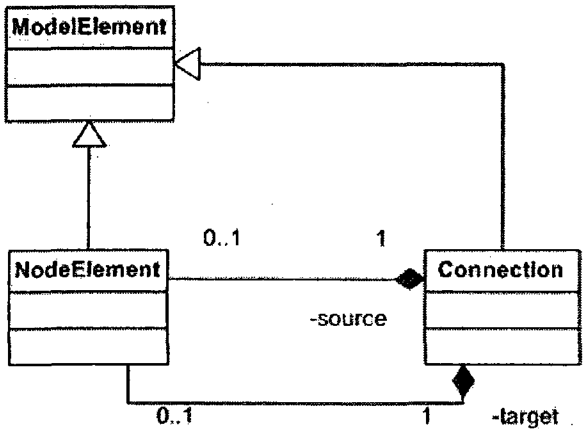


Figure 4-4 Connection Class Diagram

4.3.2 Graphical Notations

The graphical notations are implemented using the Java Draw2D Framework, which is an integrated part of GEF.

As shown in Figure 4-5, the root of the class diagram is the *Figure* class, which is a lightweight graphical component in the Draw2D. This class provides paint events handling the refresh of the diagrams when the appearance or size of a diagram element has been changed.

NodeFigure class extends *Figure* class. It serves as the superclass of all the visible notations in our SMPM (i.e., *ActivityFigure, TaskFigure, StepFigure, QueryFigure, StartPointFigure, EndPointFigure, etc.*).

ActivityFigure, TaskFigure, StepFigure, QueryFigure, StartPointFigure and *EndPointFigure* classes are visible notations used to graphically represent *Activity, Task, Step, Query, StartPoint* and *EndPoint* classes respectively in the process meta-model.

The notation class diagram is designed using a simple factory design pattern. The *FigureFactory* class is the creator class responsible for creating notation instances according to their corresponding process model elements.

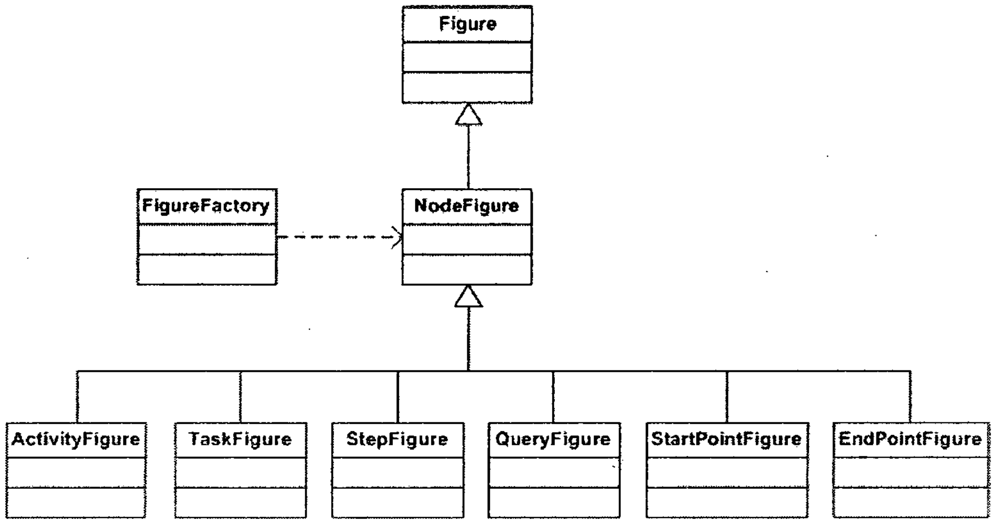


Figure 4-5 Graphical Notation Class Diagram

4.3.3 Model – Notations Mapping

Once the process meta-model and the graphical notations are ready, the next step is to associate the process model with the graphical notations. This is one of the duties that EditParts should perform. The EditPart is a term defined in GEF which refers to the controller in the MVC design pattern. As shown in Figure 4-6, the mapping is basically a one-on-one mapping between process model elements and notations, meaning that the process model elements, graphical notations and the Editparts share the same hierarchical structure. For example, a process model consists of a *Diagram* (process root) which contains process model elements as its children. There is a corresponding *DiagramEditPart* which contains child EditParts with the same parent-child relationship as its model counterpart. This parent-child relationship of EditParts carries over into their graphical notations.

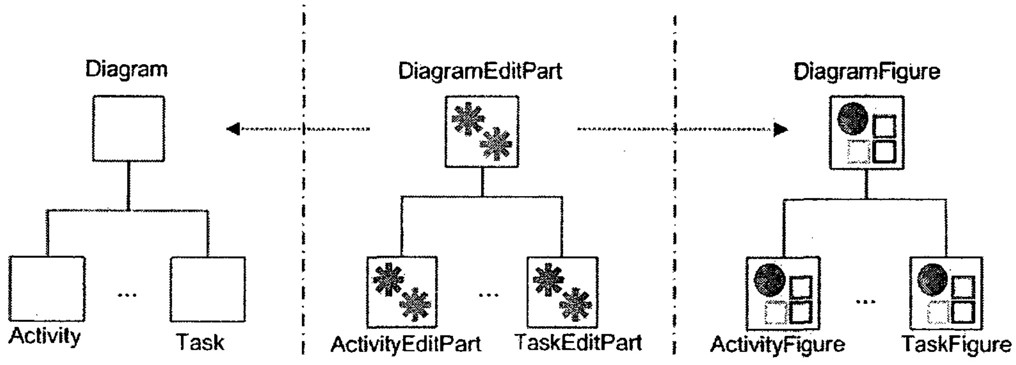


Figure 4-6 Process Model to Graphical Notations Mapping

There are two different types of EditParts implemented in our SMPM. One set of EditParts are implemented for rendering process models in graphical mode, while the other set of EditParts display process models in a tree structure view.

Figure 4-7 illustrates the graphical EditPart class diagram. This diagram is designed using a factory design pattern. *SVEditPartFactory* is the creator class responsible for creating graphical EditParts according to its underlying process model elements. Basically, each of process model elements has its counterpart EditPart classes.

Figure 4-8 shows the tree EditPart diagram. This diagram is also designed as a factory design pattern. SVTreeEditPartFactory is the creator class responsible for creating tree EditParts according to its underlying process model elements. Unlike the graphical EditPart class diagram, we use only one concrete tree EditPart (*ShapeTreeEditPart*) to represent all of the process model elements because of they are of the same shape in the tree structure view.

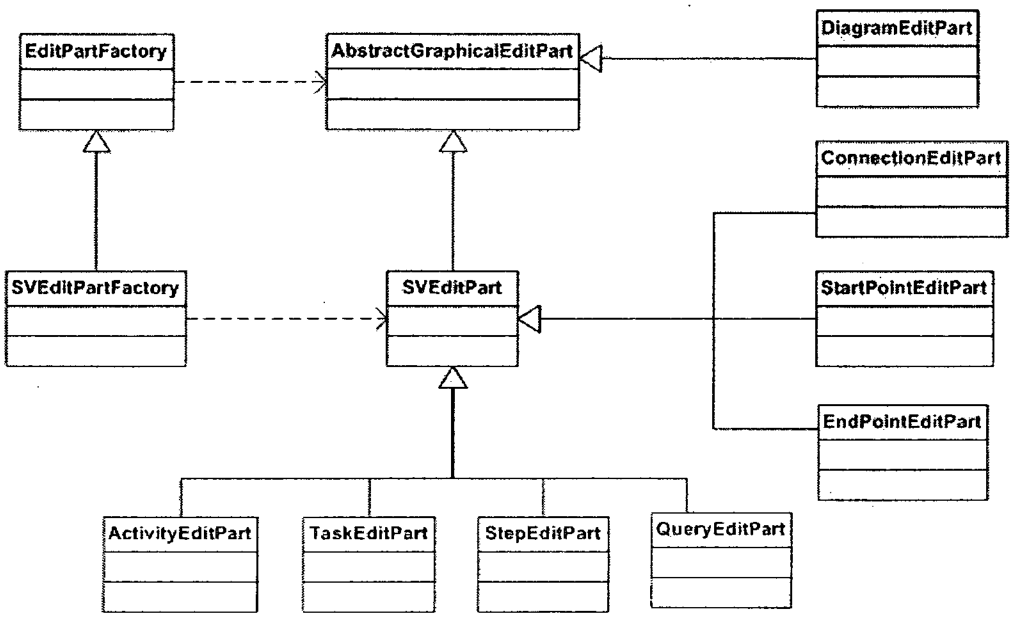


Figure 4-7 Graphical EditPart Class Diagram

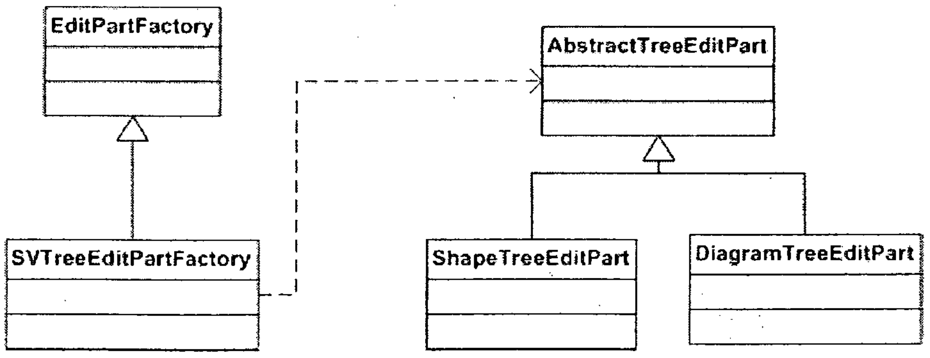


Figure 4-8 Tree EditPart Class Diagram

Chapter 5 Application

In this chapter, we present a set of examples to illustrate the applicability of our SMPM environment in guiding and managing software maintenance processes. The software under study, Debrief[13], is an open source Java application used by navies and companies around the world to analyze and report maritime exercises.

5.1 Process Design

To demonstrate these examples, we design a process of perfective maintenance for Debrief project. It is a generic maintenance process for Debrief consisting of activities of *problem and modification analysis*, and *modification implementation.* This process supports a maintenance cycle inolving tasks of Modification Request/Problem Report (MR/PR) analysis, problem reproduction, modification identification, and modification implementation.

In the rest of this section, we will describe the design of the process for maintaining perfective tasks in the Debrief project. Since all the *tasks* and *task-steps* used in the process have been fully described in the original IEEE standard, we named all these process model elements with their corresponding section numbers in the IEEE standard so that users can easily refer to the IEEE standard for the detail information. For example, a *task-step* with name of "Step 5.2.2.1 a" means that this

[13] http://www.debrief.info/index.php

task-step is derived from *Section 5.2.2.1 a* in the IEEE standard. In addition, there are too many *queries* existed in the process model, we only list those important ones and omit others. The following describes the *tasks, task-steps* and *queries* included in the maintenance process.

Problem Report (PR) analysis: The goal of this *task* is to ensure the feasibility of resolving the requested problems. The *task-steps* and *queries*, as described in Table 5-1, are included to determine if it is possible to solve the requested problems.

Step 5.2.2.1 a: determine if the maintainer is adequately staffed to implement the proposed modification	
Q1	List all programmers/maintainers working on Debrief component substitution
Q2	List all programmers who have previous experience with Eclipse plug-in
Q3	List all programmers who have previous experience with source code
Q4	List all programmers who have previous experience with reverse engineering
Q5	List all maintainers assigned to projects, by name of projects and programmers respectively
Step 5.2.2.1 c: determine if sufficient resources are available and whether this modification will affect ongoing or projected projects	
Q1	List all tools that support software visualization
Q2	List all artefacts available for Debrief
Step 5.2.2.1 i: identify ripple effects	

Q1	Show the classes that students modified in Debrief component/bug case study
Q2	Show classes potentially affected by a change of class "Debrief.Tools.Operations.SavePlotAsXML"
	....

Table 5-1 Task-steps and Queries for Task 5.2.2.1

Verification: It is a *task* to reproduce the problems/errors as described in the problem report. It includes the *task-steps* as listing in Table 5-2.

Step 5.2.2.2 c: install affected version
Step 5.2.2.2 d: run test to verify problem
Step 5.2.2.2 e: document test results

Table 5-2 Task-steps for Task 5.2.2.2

Analysis: This *task* is performed in order to determine which documentation, software units, and version thereof will need to be modified. The *task-steps* and *queries* that need to be executed to complete the *task* are listed in Table 5-3.

Step 5.3.2.1 a: identify the elements to be modified in the existing system	
Q1	List all artefacts available for Debrief
Q2	List all tools that support 5.3.2.1.a IEEE Step
Q3	List all techniques that support the IEEE task-step 5.3.2.1.a
Q4	List all tools and components that support reverse engineering of Java code
Q5	List all tools that support software visualization

	....
Step 5.3.2.1 b: Identify the interface elements affected by the modification	
Q1	List all tools that support 5.3.2.1.b IEEE Step
Q2	List all techniques that support the IEEE task-step 5.3.2.1.b
Q3	List all artefacts available for Debrief
	....
Step 5.3.2.1 c: Identify the documentation to be updated	
Q1	List all artefacts available for Debrief
	....
Step 5.3.2.1 d: Update the software documentation	
Q1	List all artefacts available for Debrief
Q2	List all tools supporting updating documentation
	....

Table 5-3 Task-steps and Queries for Task 5.3.2.1

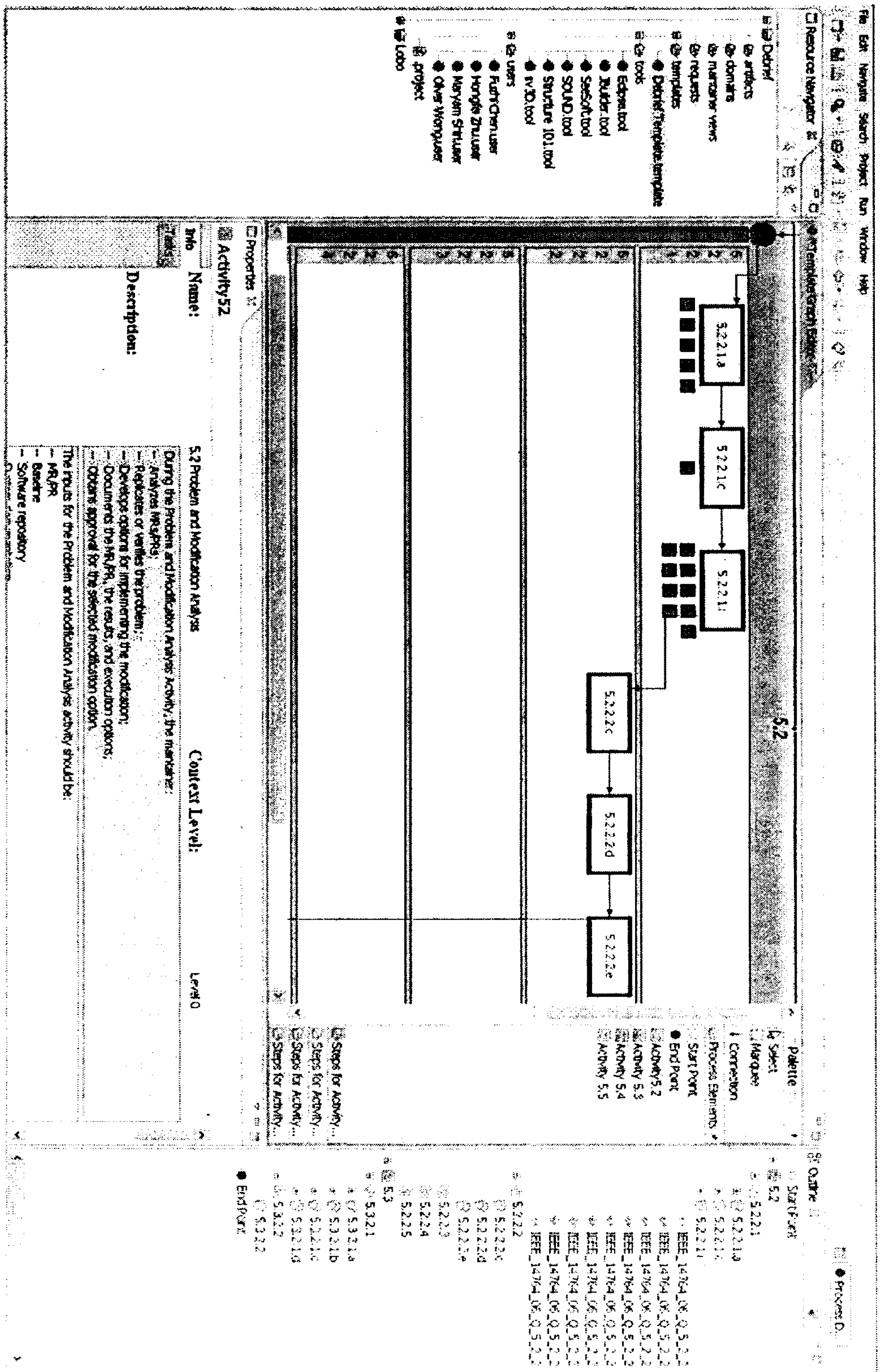
Development Process: During this *task*, the maintainer develops and tests the modification of the software product. It has to be noted that we are currently only focusing on maintenance relevant *tasks* and do not provide guidance on *tasks* related to forward engineering.

5.2 Process Template Creation

After we have the process design in place, we will demonstrate how to create this process and graphically represent it as a *process template* in our SMPM tool environment in this section.

In order to graphically enact the maintenance processes using our SMPM environment, a *process design perspective* (Figure 5-1) has been developed to aid process designers to create maintenance *process templates*.

Example One: In order to create a *process template*, process designers have to first switch to the *process design perspective*, then right click on "templates" folder in the *resources view* and select "create a template" item from the popup menu, this can create an empty *process template*. A *template editor* will be invoked and show up in the *graphical editor area* for editing by clicking on the newly created *process template* in the *resource view*. Process designers design the *process template* by dragging and dropping process model elements from the *palette* to *the template editor* and linking them together using the *connection* element. Figure 5-2 shows a *process template* that has fully modeled the maintenance process designed in Section 5.1.



Activity52

Name: 5.2 Problem and Modification Analysis

Context Level: Level 0

Description:

During the Problem and Modification Analysis Activity, the maintainer:
-- Analyzes MRs/PRs;
-- Replicates or verifies the problem;
-- Develops options for implementing the modification;
-- Documents the MR/PR, the results, and execution options;
-- Obtains approval for the selected modification option.

The inputs for the Problem and Modification Analysis activity should be:
-- MR/PR
-- Baseline
-- Software repository

Figure 5-1 The Process Design Perspective in SMPM

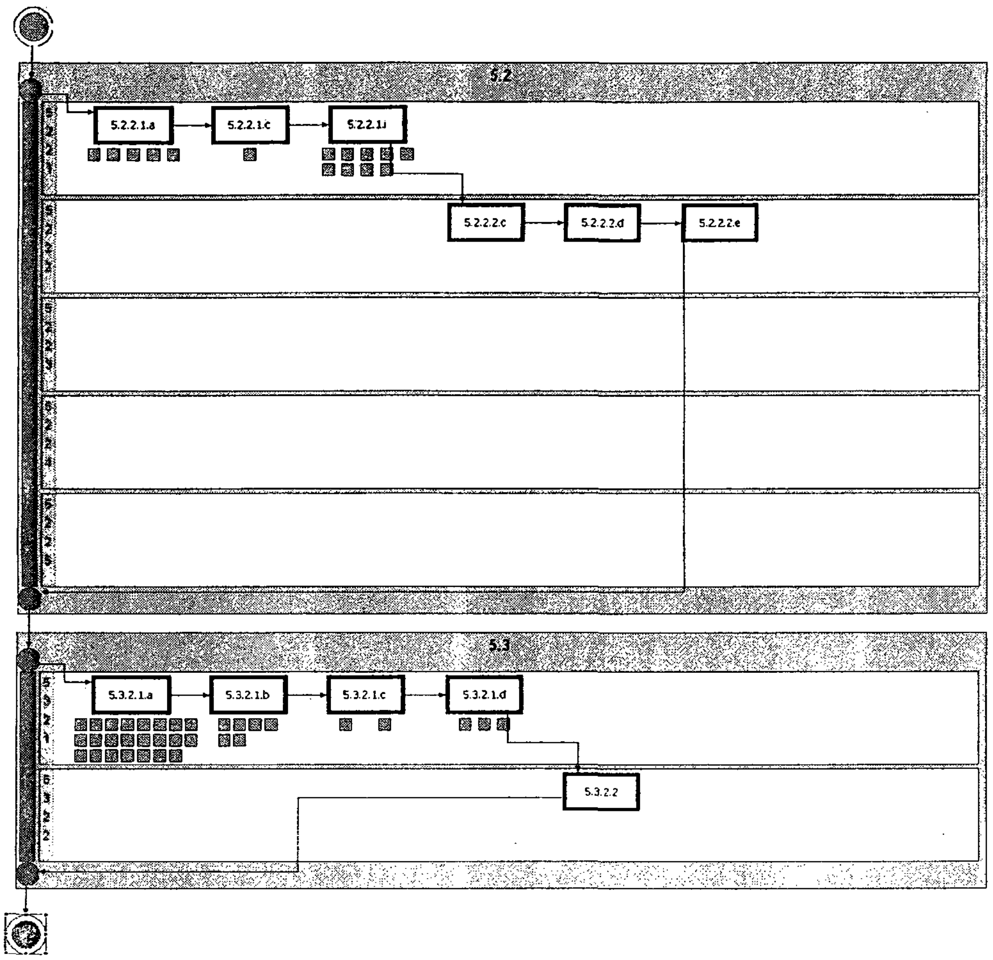


Figure 5-2 The Completed Process Template Graph

Example one confirms that our maintenance process meta-model can be applied to describe software maintenance processes. This example also indicates that the notations can graphically represent the process model elements. It further indicates that our layout strategy can be applied to eliminate the edge (*connection*) crossings in the process template graph.

5.3 Process Customization

The *process template* created in section 5.2 is just a general solution for perfective software maintenance of Debrief project. It can be further customized to meet a specific maintenance context. This is the responsibility of domain experts. In this section, we will demonstrate how domain experts can customize a domain-specific model.

Example Two: When a *process template* is created, the values of the *execution type* of all the *queries* are set by default to be *mandatory*. However, often it might not be compulsory to perform all the *queries* in order to complete a maintenance process. In this case, the *queries* to be executed as part of the maintenance process might depend on a domain expert's experience and the specific maintenance task context. Therefore, we allow the value of the *query*'s *execution type* property to be modified from *mandatory* to *optional*.

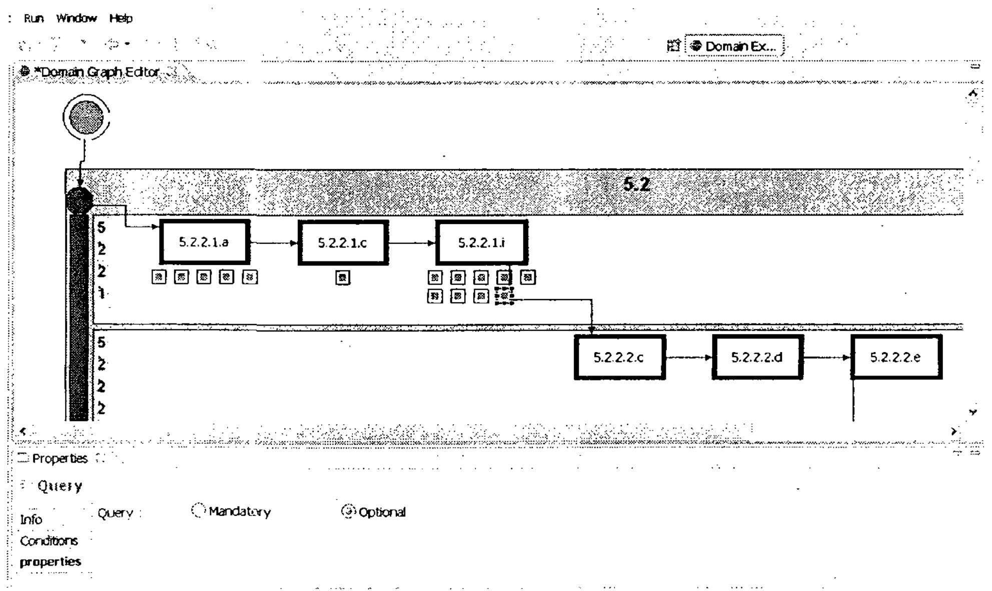
Example Three: When a *process template* is created, the values of *context level* of all the process model elements are set by default to be *Level 0*, meaning there is no constraint being applied to the process model elements, and therefore all the resources and knowledge in the KB are available to them. However, in some cases, for example, there are too many resources and knowledge available for a process model element. We may want to restrict the scope of the availability of resources and knowledge to some process model elements during the information retrieval. We can achieve this by changing the *context level* of those process model elements to a more restrictive level, for example, *context level 1, context level 2 or context level 3.*

Example Four: When a *process template* is created, all of the pre-defined *queries* are selected and available to their corresponding *task-steps*. However, not all of these pre-defined *queries* are necessary or suitable to any maintenance task in any case. Domain experts should be responsible for selecting *queries* and making them available to the process model elements based on their experience and the specific maintenance task context.

In order to further customize a *process template*, domain experts must switch to the *domain expert perspective*. The *domain expert perspective* provides necessary supports for domain experts to accomplish their work. For example, it allows domain experts to change attributes values of process model elements through the *properties view*.

Before a domain expert can customize a process, there is a need to first create a *domain-specific model* from a *process template*. In order to create a new *domain-specific model*, the domain expert selects and right clicks a *process template* from the "templates" folder, and then chooses "create a domain" from the popup menu. The newly created *domain-specific model* will be displayed in the "domains" folder in the *resources view* after the domain expert explicitly assigns it a name. By double clicking on this *domain-specific model*, a *domain graph editor* will be invoked to allow process customization. Selecting the process model elements needed to be customized in this *domain graph editor* or in the *outline view* allows the domain expert to change the attribute values of selected elements.

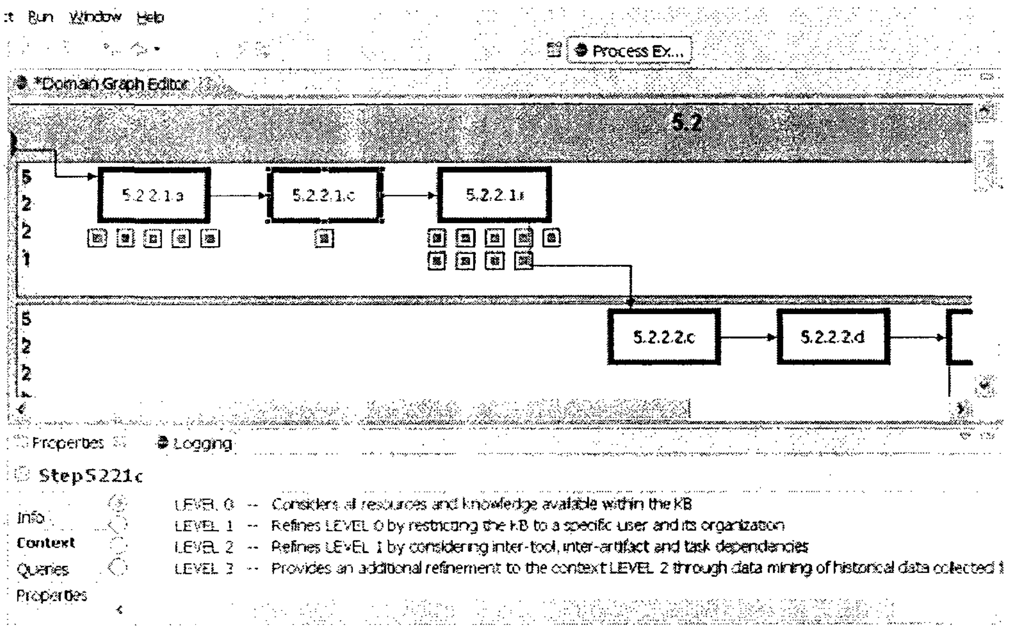
To complete the example two, a domain expert selects *queries* for each of the *task-steps*. Details about the *query* attributes are displayed in the *properties view.* Selecting the *properties tab* in the *properties view*, the domain expert can change the *execution type* of a *query* from *mandatory* to *optional.* Figure 5-3 snapshots the moment the domain expert changing the value of *execution type* of a *query* in the *domain expert perspective.*



Query : Mandatory [✓ Optional]

Figure 5-3 Customization of the Execution Type

To complete the example three, a domain expert selects a *task-step* with its attributes and their details being displayed in the *properties view*. Selecting the *context tab* in the *properties view*, the domain expert can change the *context level* value to this *task-step*. Figure 5-4 shows the GUI in the SMPM to allow accomplishment of this customization task.



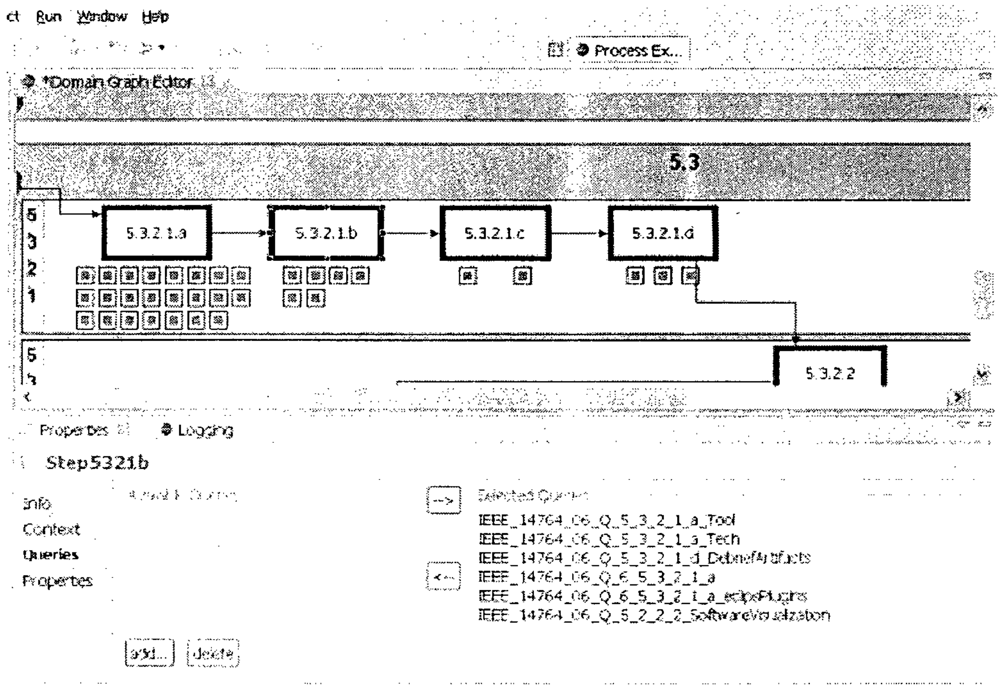
Step5221c

LEVEL 0 -- Considers all resources and knowledge available within the KB
LEVEL 1 -- Refines LEVEL 0 by restricting the KB to a specific user and its organization
LEVEL 2 -- Refines LEVEL 1 by considering inter-tool, inter-artifact and task dependencies
LEVEL 3 -- Provides an additional refinement to the context LEVEL 2 through data mining of historical data collected

Figure 5-4 Customization of the Context Level

To complete the example four, a domain expert selects a *task-step*. Its associated attribute details will be displayed in the *properties view*. Selecting the *query tab* in the *properties view*, there are *Available Queries List* and *Selected Queries List* as shown in Figure 5-5. *Available Queries List* lists all of the candidate *queries*, which are potential *queries* to the selected *task-step*, but they are not selected and therefore invisible to end users. On the other hand, *Selected Queries List* displays the *queries* which have been selected and made available (display) to end users. The domain expert can customize the pre-defined *queries* by registering new *queries* to the

Available Queries List, deleting pre-defined *queries* from the *Available Queries List*, and moving the pre-defined *queries* back and forth between the *Available Queries List* and the *Selected Queries List*. Figure 5-5 displays a GUI interface that allows domain experts to accomplish these customization operations.



IEEE_14764_06_Q_5_3_2_1_a_Tool
IEEE_14764_06_Q_5_3_2_1_a_Tech
IEEE_14764_06_Q_5_3_2_1_d_DebriefArtifacts
IEEE_14764_06_Q_6_5_3_2_1_a
IEEE_14764_06_Q_6_5_3_2_1_a_eclipsePlugins
IEEE_14764_06_Q_5_2_2_2_SoftwareVisualization

Figure 5-5 Customization of the Pre-defined Queries

In the previous examples we have shown that the domain-specific models can be customized, and the SMPM tool environment provides enough support for doing process customization.

5.4 Process Application

A process instance can be applied to guide maintainers while performing a specific maintenance task. A *software maintainer perspective*, which is equipped with abilities to allow *queries* being executed and feedback collection, has been developed in order to provide supports for completing process application.

5.4.1 Process Instantiation

Before one can apply a process instance to guide users through a maintenance task, it must be instantiated based on a *domain-specific model.* In order to instantiate a *process-specific model*, a maintainer switches his/her workbench to the *software maintainer perspective.* In the next step a *domain-specific model* to be instantiated is selected in the "domains" folder found in the *resource view.* Finally, the maintainer has to right click this *domain-specific model* and select "create a new maintainer view" item from the popup menu. The newly created *process instance* will be displayed in the "maintainer view" folder after the maintainer has assigned a name to it. Double clicking the newly created *process instance* will invoke the *comprehension graph editor* to display the *process instance* graphically. Additionally, the hierarchical structure of the *process instance* will be displayed in the *outline view* and the *process instance* is ready to be navigated and applied.

5.4.2 Process Navigation

To navigate a *process instance,* a software maintainer should first open the *process instance* in the *comprehension graph editor.* As illustrated in Figure 5-6, maintainers can now navigate the *process instance* using the *outline view* or *comprehension graph editor.* In our SMPM, there are three different ways to help navigate *process instances.*

The first approach to navigate *process instances* is to use *outline view.* When we select an interested process model element in the *outline view*, its corresponding graphical notation will be centralized and focused in the *comprehension graph editor* as shown in Figure 5-6. A marker will also be added to the interested graphical notation in the *comprehension graph editor* to help oriented it in the graph.

The second approach to navigate process instances is achieved by using horizontal and vertical scroll bars as indicated in Figure 5-6. By moving these scroll bars, software maintainers can easily navigate the entire *process instance* in the *comprehension graph editor.*

The third approach to navigate *process instances* is achieved by dragging the mouse on the *comprehension graph editor.* This is approach is actually the implementation of *panning* visualization technique as discussed in Section 2.4.1.

5.4.3 Process Application

Except process navigation, software maintainers can also apply a process instance to help complete their maintenance tasks.

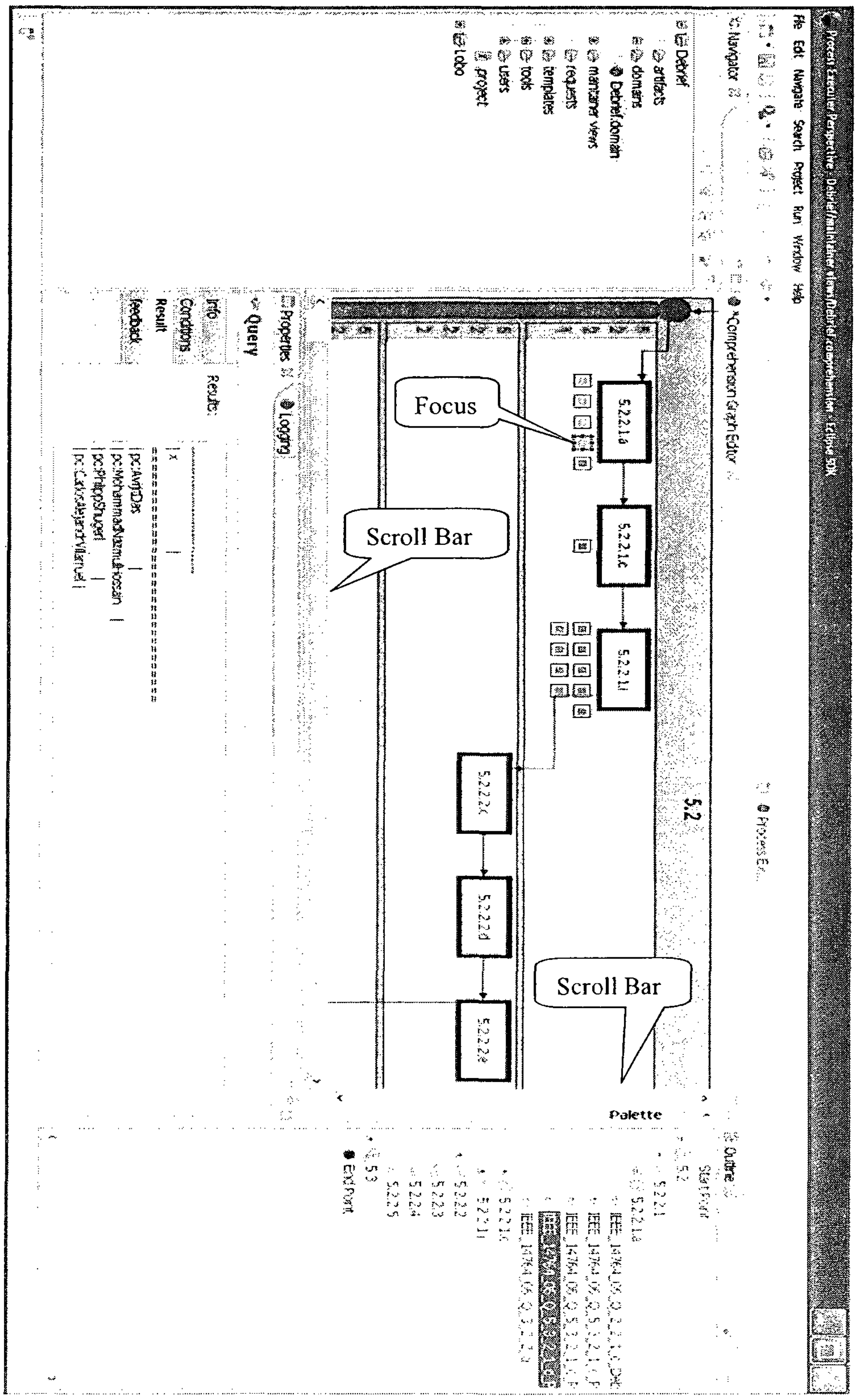
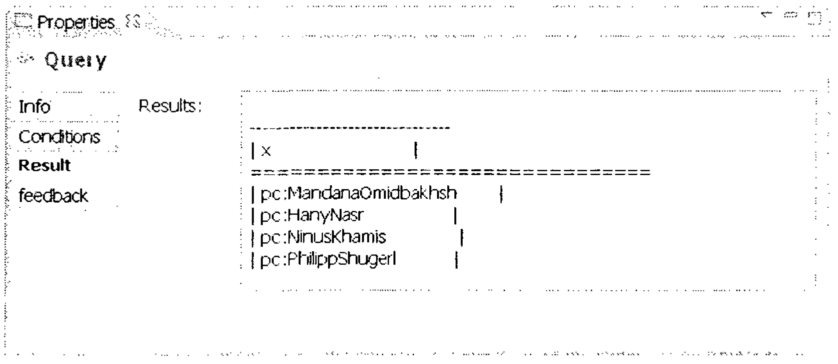


Figure 5-6 The Software Maintainer Perspective in SMPM

Example Five: The first *task-step* of the maintenance process is *Step 5.2.2.1.a: determine if the maintainer is adequately staffed to implement the proposed modification.* This *task-step* is supported by five pre-defined *queries*. Selecting any of these *queries* will result in it being executed and the results will be displayed in the *results tab* under the *properties view*. Figure 5-7 shows the results of the *query: IEEE_14764_06_Q_2_2_1_c_Programmer_Eclipse_Plugins: List all maintainers who have previous experience with Eclipse Plug-ins*. From the results, maintainers know that there are four maintainers/programmers who have experience on Eclipse plug-in development. By reviewing background information of these maintainers/programmers, maintainers can determine if they have enough qualified maintainers/programmers for Eclipse plug-ins development in their organization.



```
-----------------------------
| x                |
=============================
| pc:MandanaOmidbakhsh      |
| pc:HanyNasr          |
| pc:NinusKhamis        |
| pc:PhilippShugerl       |
```

Figure 5-7 Display of Query Results

Table 5-4 lists the results of all of the *queries* attached to *Step5.2.2.1 a*. By analyzing these results, maintainers can answer the question that arises from *Step*

5.2.2.1 a. Similarly, maintainers can execute *Step 5.2.2.1.c* which is the next to the first *task-step*. Step by step, they go through the entire maintenance process.

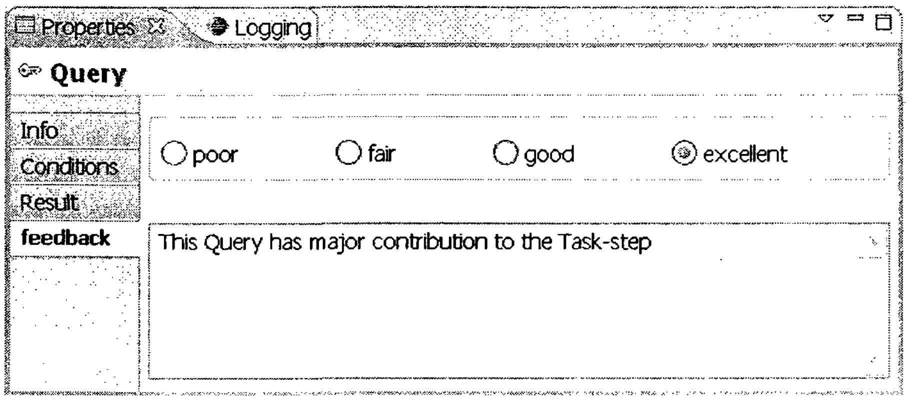
		Results
Step 5.2.2.1 a: determine if the maintainer is adequately staffed to implement the proposed modification		
Q1	List all programmers/maintainers working on Debrief component substitution	4 maintainers
Q2	List all programmers who have previous experience with Eclipse plug-in	4 maintainers
Q3	List all programmers who have previous experience with source code	4 maintainers
Q4	List all programmers who have previous experience with reverse engineering	4 maintainers
Q5	List all maintainers assigned to projects, by name of projects and programmers respectively	4 maintainers

Table 5-4: Results of all Queries of Step 5.2.2.1.a

Example five illustrated that our SMPM can support the visual linking of process model elements with their relevant resources and knowledge through the use of *query* elements. This example also demonstrates our SMPM supported the *queries* execution. By analyzing the results of these *queries*, maintainers received guidance to help complete their maintenance tasks.

5.4.4 Feedback Collection

During the example five, maintainers might give their feedback and comments in the *properties view's feedback tab* (Figure 5-8) which is along with the process model elements.



poor fair good [✓ excellent]

This Query has major contribution to the Task-step

Figure 5-8: Feedback Collection View

Example Six: Again, we take *query*: *IEEE_14764_06_Q_2_2_1_c_Programmer_Eclipse_Plugins: List all maintainers who have previous experience with Eclipse Plug-ins* as example. By reviewing the result of this *query*, maintainers may rank the contribution of this *query* to the containing *task-step*. In this case, the results returned by *query*: *IEEE_14764_06_Q_2_2_1_c_Programmer_Eclipse_Plugins* had significant contribution to answer the question raised in *Step 5.2.2.1.a*. Therefore, it has been ranked "Excellent".

Example six demonstrates that our SMPM tool provides a GUI interface to collect maintainers' experience and feedback.

5.5 Discussion and Limitation

The presented examples demonstrated that our SMPM tool environment can be used to enact tailored software maintenance processes as described in Section 3.1, which adopted from the IEEE standard.

The examples show clearly that the proposed process meta-model can be used to describe maintenance processes. Graphical notations are simple and can represent process model elements in a clear and neat manner. Furthermore, these notations are familiar with users because most of them adopted from existing process standards such as BPMN and UML. In addition, *overview and detail* visualization approach has been implemented in the SMPM.

Section 5.4.2 shows that the *panning* visualization technique has also been implemented in the SMPM tool environment. But the *zooming* visualization has not been addressed. The lack of the *zooming* visualization limits the SMPM to provide *level of abstraction* of maintenance processes in graphical model to its users. It also limits the ability of graphically navigating in very large and complicate maintenance processes.

The examples also show that the proposed meta-model does not have *fork* and *join* constructs, which are popular in some other famous modeling languages such as

BPEL and UML. According to UML, *fork* and *join* constructs are usually used to represent parallel processing activities. Specific to our case, the IEEE standard does not mention any parallel executing activities in the specification. Therefore, our meta-model does not model this kind of behavior.

From the demonstration of examples, we noticed that our SMPM does not support *loop* construct in the meta-model as many other modeling languages such as UML and BPEL do. According to the IEEE standard, activities of *problem and modification analysis*, *modification implementation* and *maintenance review/acceptance* may be called iteratively if necessary, this actually form a *loop* construct. The lack of the *loop* construct limits our SMPM tool environment to describe these repeatedly occurred activities.

The presented examples also show that the SMPM does not provide support for managing the resources knowledge, for example, managing the information of various tools, artefacts and maintainers. This information must be inputted into the KB prior to the SMPM running. Finally, the collected feedback and historical data cannot be stored into the KB. The lack of this feature limits the context-sensitive support of the SMPM. This is due to the fact that the team responsible for developing the KB did not deliver APIs to allow "write" operation to the KB. Once the "write" APIs are available, this information will eventually be stored into the KB.

Chapter 6 Related Work

With regard to process modeling language and meta-model domain, BPEL [21] is a process modeling language specific for modeling business processes. But it does not provide graphical notations to be able to visualize business processes. BPMN [23] is considered as visual process modeling language for modeling business processes. Both BPEL and BPMN are executable languages and can be used to model behavior aspect of processes. UML [24] is a standardized general-purpose visual modeling language in the field of software engineering. Its activity diagram is suitable for describing the business and operational processes. However, it can only describe the structure and flow of processes, thus it is not an executable language. SPEM [28] is a meta-model for modeling processes in software engineering domain. It is a standard closely related to the UML, therefore it is also considered as visual language without execution ability.

The meta-model developed in the presented research differs from above modeling languages and meta-model in the purpose. It is a meta-model for describing processes in software maintenance domain. To be more specific, it is a meta-model describing ISO/IEC 14764 [9] software maintenance process. The meta-model comes with a set of notations, so it is a visual process modeling language. The meta-model contains an execution model element called *query* which can be executed to extract information from the underlying ontology. More importantly, the classification of *task-steps* into *mandatory* and *optional* achieves the similar goal of the *pick* activity in BPEL meanwhile keeps the process layout simple and neat.

Spemmet [66] and APSEE [16] are tool environments for modeling software processes. Spemmt has been built on top of the existing SPEM meta-model. In contrast, APSEE created its own meta-model and visual notations. Both of tools focus on describing the structure aspect instead of behavior aspect of process models. Our approach differs from them in 1) it is a meta-model used to describe not only the structure but also the behavior aspects of process models; 2) it is a process meta-model used to describe software maintenance processes; and 3) the graphical notations of our approach adopt from well known modeling languages such as BPMN and UML that ordinary users are fimiliar with.

The Eclipse Process Framework (EPF[14]) Composer and IBM Rational Method Composer (RMC[15]) are two process management tools closely related to the presented research. EPF supports various process frameworks, such as OpenUP/Basic, extreme programming and Scrum. It can also be used to create new process framework from scratch. But RMC is shipped with the RUP process framework. Both of them are built on top of Eclipse platform with the GEF as the foundation for their graphical notations. Our SMPM tool environment has also been developed based on the same Eclipse platform and took GEF as its graphical infrastructure. But our SMPM differs

[14] EPF http://www.eclipse.org/epf

[15] RMC http://www.ibm.com/software/awdtools/rmc

from them in that it supports only the ISO/IEC-14764 software maintenance process rather than software development processes. Furthermore, RMC constructs reusable building blocks as capability patterns which can be used to assemble processes to meet specific needs of a given project. Our approach represents reusable processes as process templates which can be further customized by domain experts. In addition, EPF represents information and knowledge as method contents, while RMC stores knowledge using a process content library. Our SMPM stores knowledge using ontologies.

Compared with the IBM Rational Process Advisor [75] (RPA), an application developed by IBM Rational for tool integration, our approach differs in its motivation. RPA implements the RUP process and attempt to integrate a selected set of software development tools within the RUP. Instead of tightly integrating the tools with the process model, our approach however focuses on knowledge integration, meaning to integrate the knowledge of resources with process model by creating links between the process model elements and relevant resources. Therefore, our approach is not limited by a set of specific tools.

Chapter 7 Conclusion and Future Works

This thesis presented the design a tool environment for enacting software mainteannce process models. As discussed throughout the thesis, we have created a software mainenance process model by extending the IEEE maintenance process model. We have also designed a process meta-model to describe the software maintenance process model. And then a set of notations have been created to allow graphically represent the maintenance process model elements. Finally, the process meta-model and notations have been implemented into a tool environment called Software Mainteannce Process Modeller (SMPM).

A series of examples have been performed to illustrate how the SMPM manages a software maintenance process, and how a typical maintenance process can be applied to provide guidance for maintainers on performing their maintenance tasks. In particular, we also demonstrated the procedures of using our SMPM to create process templates, customize the domain-specific model to meet the specific needs of a particular domain, and finally apply the process instance to provide guidance for its users' maintenance tasks. These examples verified that our SMPM can be used to enact maintenance processes. These examples also illustrated that our SMPM can support the visuall linking of process model with their relevant resources and knowledge through the use of *query* element. By analyzing the result of the *queries*, maintainers can get guidance to complete their maintenance tasks.

As part of future work, zooming visualization would be implemented to the SMPM to provide level of abstraction while graphically display maitnenance processes.

Another issue for future work is the need to implement the *loop* construct of the meta-model so that it can be used to describe repeatedly occurred activities.

If the "write" APIs are available as discussed in Section 5.5, the feedback and historical data should be stored into the KB instead of storing them in the local storage.

Finally, managing knowledge of various resources such as tools, artefacts and maintainers in the SMPM would be desirable.

Bibliography

[1] M.-A. D. Storey, "Theories, Methods and Tools in Program Comprehension: Past, Present, and Future," in *the 13th International Workshop on Program Comprehension (IWPC 2005)*, 2005, pp. 181-191.

[2] Margaret-Anne D Storey, Susan Elliott Sim, and Kenny Wong, "A collaborative demonstration of reverse engineering tools," in *ACM SIGAPP Applied Computing Review, Vol. 10, Issue 1*, 2002, pp. 18-25.

[3] A. Von Mayrhauser and A.M. Vans, "Program comprehension during software maintenance and evolution," *IEEE Computer*, pp. 44-55, 1995.

[4] Juergen Rilling, Wen Jun Meng, Fuzhi Chen, and Philippe Charland, "Software Visualization – A Process Perspective ," in *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis(VISSOFT)*, Banff Centre, Alberta, Canada, 2007, pp. 10-17.

[5] Juergen Rilling, Wen Jun Meng, René Witte, and Philippe Charland, "Story Driven Approach to Software Evolution," *IET Software*, vol. 2, p. 304–320, August 2008.

[6] Victor Basili, "Viewing Maintenance as Reuse Oriented Software Development," vol. 7, pp. 19-25, 1990.

[7] M. M. Lehman and L. A. Belady, *Program evolution: processes of software change*. San Diego, CA: Academic Press Professional, 1985.

[8] CMMI Product Team, *CMMI® for Development, Version 1.2*. Pittsburgh, PA, USA: Carnegie Mellon, Software Engineering Institute, 2006.

[9] IEEE Std 14764-2006, *ISO/IEC 14764 IEEE Std 14764-2006 Software Engineering, Software Life Cycle Processes, Maintenance, ISBN: 0-7381-4961-6.*: IEEE, 2006.

[10] Gerardo Canfora and Aniello Cimitile, "Software maintenance," *Handbook of Software Eng. and Knowledge Eng.*, 2002.

[11] H. A. M¨uller and K. Klashinsky, "Rigi — A system for programming-in-the-large," in *the 10th International Conference on Software Engineering*, 1988, p. 80–86.

[12] Susan Elliott Sim, Richard C. Holt, and Steve Easterbrook, "On Using a Benchmark to Evaluate C++ Extractors," in *the Tenth International Workshop on Program Comprehension*, 2002, pp. 114-123.

[13] M. Lanza and S. Ducasse, "CodeCrawler - An Extensible and Language Independent 2D and 3D Software Visualization Tool," *Tools for Software Maintenance and Reengineering*, pp. 74 - 94, 2005.

[14] R. Lintern, J. Michaud, M.-A. Storey, and X. Wu, "Plugging-in Visualization: Experiences Integrating a Visualization Tool with Eclipse," in *ACM Symp. on Software Visualization, (Softvis)*, San Diego, 2003.

[15] Stephen G. Eick, Joseph L. Steffen, and Jr., Eric E. Sumner, "Seesoft-A Tool for Visualizing Line Oriented Software Statistics," in *IEEE Transactions on Software Engineering*, 1992 , pp.

957 - 968.

[16] Carla A. Lima Reis, Rodrigo Quites Reis, Marcelo Abreu, Heribert Schlebbe, and Daltro J. Nunes, "Flexible Software Process Enactment Support in the APSEE Model," in *the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, 2002, p. 112.

[17] S. Bandinelli, A. Fuggetta, L. Lavazza, M. Loi, and G. P. Picco, "Modeling and improving an industrial software process," in *IEEE Transactions on Software*, 1995, p. 440–454.

[18] V. Ambriola, P. Ciancarini, and C. Montangero, "Software process enactment in Oikos," *ACM SIGSOFT Software Engineering Notes*, vol. 15, no. 6, pp. 183 - 192, 1990.

[19] S.-C. Chou and J.-Y.J. Chen, "Process evolution support in concurrent software process language environment," *Information and Software Technology*, p. 507–524, 1999.

[20] X. Cr'egut and B. Coulette, "PBOOL : an objectoriented language for definition and reuse of enactable processes," *Software Concepts & Tools*, vol. 18, no. 2, 1997.

[21] OASIS, *Web Services Business Process Execution Language Version 2.0.*: OASIS, 2007.

[22] (2009, June) wikipedia bpel. [Online]. http://en.wikipedia.org/wiki/Business_Process_Execution_Language

[23] OMG, *Business Process Modeling Notation Specification.*: OMG, 2006.

[24] OMG, *OMG Unified Modeling LanguageTM (OMG UML), Superstructure*, 22nd ed.: OMG, 2009.

[25] (2009, June) wikipedia UML. [Online]. http://en.wikipedia.org/wiki/Unified_Modeling_Language

[26] Changyun Li, Jin Gou, Huifeng Wu, and Gansheng Li, "A process meta-model supporting domain reuse," *2005 International Software Process Workshop*, pp. 459-461, 2005.

[27] David Hollingsworth, *Workflow Management Coalition - The Workflow Reference Model.* Hampshire, UK: Workflow Management Coalition, 1995.

[28] OMG, *Software & Systems Process Engineering Meta-Model Specification Version 2.0.*: OMG, 2008.

[29] Inava Rasovska, Brigitte Chebel-Morello, and Noureddine Zerhouni, "A conceptual model of maintenance process in unified modeling language," in *the 11th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2004)*, Salvador, Brazil, 2004, pp. 43-48.

[30] Thomas M Pigoski, *Practical Software Maintenance – Best Practices for Managing Your Software Investment*. New York, NY: John Wiley & Sons, 1997.

[31] N. F. Schneidewind, "The State of Software Maintenance," in *IEEE Transactions on Software Engineering* , Piscataway, NJ, USA , 1987, pp. 303-310.

[32] IEEE Std. 1219, *Standard for Software Maintenance.*: IEEE Computer Society Press, 1993.

[33] B. P. Lientz and E. B. Swanson, *Software Maintenance Management.*: Addison Wesley, 1980.

[34] Burton Swanson, "The dimensions of maintenance," in *the 2nd international conference on Software engineering*, 1976, pp. 492-497.

[35] A. A. Takang and P.A. Grubb, *Software Maintenance Concepts and Practic.* London, UK: Thompson Computer Press, 1996.

[36] H. VAN Vliet, *Software Engineering:Principles and Practices*, 2nd ed. West Sussex, England: John Wiley & Sons, 2000.

[37] N. Chapin and A. Cimitile, "Announcement," *Software Maintenance and Evolution : Research and Practice*, vol. 13, no. 1, 2001.

[38] R. Seacord, D. Plakosh, and G. Lewis, *Modernizing Legacy Systems: Software Technologies, Engineering Processes, and Business Practices.*: Addison-Wesley, 2003.

[39] IEEE Std 1074-1997, *IEEE Std 1074-1997 IEEE Standard for Developing Software Life Cycle Processes.*: IEEE, 1997.

[40] Filippo Lanubile and Giuseppe Visaggio, "Iterative Reengineering to compensate for Quick-Fix Maintenance," *ICSM 1995*, pp. 140-146, 1995.

[41] Kagan Erdil et al., *Software Maintenance As Part of the Software Life Cycle.*: Department of Computer Science, Tufts University, 2003.

[42] Cesar Gonzalez-Perez and Brian Henderson-Sellers, "Modelling software development methodologies:A conceptual foundation," *Systems and Software*, vol. 80, no. 11, pp. 1778-1796 , 2007.

[43] Peter Höfferer, "Achieving Business Process Model Interoperability Using Metamodels and Ontologies," in *the 15th European Conference on Information Systems (ECIS2007)*, 2007, pp. 1620--1631.

[44] Jean-Marie Favre, "Foundations of Meta-Pyramids: Languages vs. Metamodels - Episode II: Story of Thotus the Baboon," in *Language Engineering for Model-Driven Software Development*, 2005.

[45] Jan Kunstar and Iveta Adamuscinova, "The use of development models for improvement of software maintenance," *Sapientiae, Informatica*, vol. 1, no. 1, pp. 45-52, 2009.

[46] Peter E. Clark, David Morley, Vinay K. Chaudhri, and Karen L. Myers, "A Portable Process Language," in *Proc ICAPS Workshop on the Role of Ontologies in AI Planning and Scheduling*, 2005.

[47] Benoit Combemale, Xavier Cr'egut, Alain Caplain, and Bernard Coulette, "Towards a Rigorous Process Modeling with SPEM," in *ICEIS (3) 2006*, 2006, pp. 530-533.

[48] (2009) UML 2 Activity Diagramming Guidelines. [Online]. http://www.agilemodeling.com/style/activityDiagram.htm

[49] Y.K. Leung and M.D. Apperley, "A review and taxonomy of distortion-oriented presentation techniques," in *ACM Transactions on Computer-Human Interaction (TOCHI)*, 1994, pp. 126 - 160.

[50] M. Eichberg, M. Haupt, M. Mezini, and T. Schafer, "Comprehensive software understanding with SEXTANT," in *the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005, pp. 315- 324.

[51] Patrick Baudisch, Nathaniel Good, Victoria Bellotti, and Pamela Schraedley, "Keeping things

in context: a comparative evaluation of focus plus context screens, overviews, and zooming," in *Proceedings of the SIGCHI conference on Human factors in computing systems: Changing our world, changing ourselves* , 2002, pp. 259 - 266.

[52] George W. Furnas and Benjamin B. Bederson, "Space-scale diagrams: understanding multiscale interfaces," in *the SIGCHI conference on Human factors in computing systems*, Denver, Colorado, 1995, pp. 234 - 241.

[53] Chris Olston and Allison Woodruff, "Getting Portals to Behave," in *the IEEE Symposium on Information Vizualization 2000*, 2000, pp. 15 - 25.

[54] M. C. Stone, K. Fishkin, and E. A. Bier, "The movable filter as a user interface tool," in *the ACM SIGCHI Conference on Human Factors in Computing Systems*, Boston, Massachusetts, 1994, p. 306–312.

[55] K. Perlin and D. Fox, "Pad: An alternative approach to the computer interface," in *the 20th International Conference on Computer Graphics and Interactive Techniques*, Anaheim, California, 1993, p. 57–64.

[56] Michelle Q. Wang Baldonado, Allison Woodruff, and Allan Kuchinsky, "Guidelines for using multiple views in information visualization," in *the working conference on Advanced visual interfaces*, Palermo, Italy, 2000, pp. 110 - 119.

[57] C. Plaisant, D. Carr, and B. Shneiderman, "Image-Browser Taxonomy and Guidelines for Designers," in *IEEE Software*, 1995, p. 21–32.

[58] K. Hornbaek and E. Frokjaer, "Reading of electronic documents: the usability of linear, fisheye, and overview+detail interfaces," in *the SIGCHI conference on Human factors in computing systems*, Seattle, Washington, 2001, pp. 293 - 300.

[59] F. M. Monk, P. Walsh, and A. J. Dix, "A comparison of Hypertext, scrolling and folding as mechanisms for program browsing," in *the Fourth Conference of the British Computer Society on People and computers Vol. 4*, 1988, pp. 421-435.

[60] D. V. Beard and J. Q. Walker, "Navigational techniques to improve display of large two-dimensional spaces," *Behaviour & Information Technology, Volume 9*, p. 451 – 466, 1990.

[61] Patrick Baudisch, Nathaniel Good, and Paul Stewart, "Focus plus context screens: combining display technology with visualization techniques," in *the 14th annual ACM symposium on User interface software and technology*, 2001, pp. 31 - 40.

[62] T. Schafer and M. Mezini, "Towards More Flexibility in Software Visualization Tools," in *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT '05)*, 2005, pp. 1- 6.

[63] M. F. Kleyn and P. C. Gingrich, "Graphtrace - understanding object-oriented systems using concurrently animated views," in *Proc. of OOPSLA*, 1988, pp. 191-205.

[64] B. Shneiderman, "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations," in *IEEE Visual Languages*, 1996, pp. 336-343.

[65] Peter Haumer. (2007, Apil) www.eclipse.org/epf. [Online]. http://www.eclipse.org/epf/general/getting_started.php

[66] Tuomas Makila and Antero Jarvi, "Spemmet - A Tool for Modeling Software Processes with SPEM," in *the 9th International Conference on Information Systems Implementation and Modelling, ISIM '06*, 2006, pp. 87-94.

[67] Rodrigo Quites Reis, Daltro José Nunes, and Carla Alessandra Gomes de Lima, "Dynamic Software Process Manager for the PROSOFT Software Engineering Environment," in *Symposim on Software Technology (SoST98)*, 1998, pp. 197-202.

[68] R. Marshak, "Software to Support BPR - The value of Capturing Process Definitions," *Workgroup Computing Report, Patricia Seybold Group, Vol. 17, No. 7*, 1994.

[69] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. P.-Schneider, *The Description Logic Handbook.*: Cambridge University Press, 2003.

[70] V. Haarslev and R. Möller, "RACER System Description," in *In Proc. of Int. Joint Conference on Automated Reasoning, IJCAR'2001*, 2001, pp. 701-705.

[71] C. Knight and M. Munro, "Mediating Diverse Visualisations for Comprehension," *Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Canada*, pp. 18-25, 2001.

[72] M.-A. D. Storey, C. Best, and J. Michaud, "SHriMP Views: An Interactive Environment for Exploring Java Programs," *Ninth International Workshop on Program Comprehension (IWPC'01), Toronto, Ontario, Canada*, pp. 111-112, May 2001.

[73] Jarke J. van Wijk and Huub Van de Wetering, "Cushion Treemaps: Visualization of Hierarchical Information," *the IEEE Symposium on Information Visualization(Info Vis apos;99)*, pp. 73-78, 1999.

[74] Catherine Plaisant, Jesse Grosjean, and Benjamin B. Bederson, "SpaceTree: Supporting Exploration in Large Node Link Tree," *the IEEE Symposium on Information Visualization (InfoVis'02)*, pp. 57-64, 2002.

[75] Jeff Smith, Dan Popescu, and Alfredo Bencomo. (2009, June) IBM Rational Process Advisor. [Online]. http://www.ibm.com/developerworks/rational/library/06/1212_smith-popescu-bencomo/