Chapter 1

Introduction

1.1 Software Process Research

A successful software project requires a combination of technical and managerial solutions, including a well-defined software development process. Indeed, it is widely accepted that software process plays an important role in the development of software products. The main purposes of having a well-defined software process are to (1) promote process understanding and (2) provide guidance to project team members. It also helps with (3) promoting learning in software engineering and (4) providing a basis for software process improvement.

Traditionally, software development highly depends on human decisions. However, as technologies evolve, they are gradually utilized for enhancing this process. With recent advancements in technology, an increasing number of software development tasks are systematically automated by computers while the relative involvement of human decision is considerably reduced. However, for some tasks, especially for many of the sophisticated ones such as process planning, decision making and process quality control, human involvement is still very much necessary. Yet, a research question is raised whether how much of technology can be of assistance in order to systematically automate these tasks without human involvement. For that reason, a great deal of research work has been done towards process development methods and process modeling.

In the past decades, software markets have rapidly grown to a great extent and that adds a new dimension to software projects. Many aspects of software development have changed. Not only the size of software (from huge aircraft control systems to tiny embedded applications on a mobile phone) but also the sizes of companies that develop software vary significantly. Since latest technologies are

widely and equally accessible in this present world, big budget companies no longer have technical advantages over smaller software houses in developing software. This changes the scale of software development dramatically. In the present days, customers vary from a grocery owner who wants to buy a simple software program customized for her store or it could be an airline company who wants a sophisticated flight entertainment software system. Such a situation illustrates a big gap in user requirements which explains a need for different development processes, depending on kinds of applications developed as well as size and culture of software companies. A 'one size fits all' software process is no longer acceptable. Companies must appropriately adopt and adapt processes according to their needs.

The significance of software processes have been confirmed in several examples of software project failures. Inattention to processes can cause several problems to software projects such as lack of control, release defects, integration problems, budget and time overrun etc (McConnell 1998). The significance of software processes is realized by several leading software engineering organizations including the Software Engineering Institute (SEI) at Carnegie Melon University. SEI introduces CMM (Capability Maturity Model) as a conceptual framework used to assess the process maturity, capability and performance of software development organizations. The framework contains five levels. Each level, except Level 1, defines the key process areas an organization should focus on to improve its software process.

Having an appropriate software process for a software project can certainly help reduce problems that may have happened during development. However, creating one from scratch may appear to be inefficient. The famous process models—Waterfall, Spiral and recent agile methods—are all the results of years of software development experience. Many software projects have chosen to follow the trail of successful projects by either reusing their processes or tailoring them to fit their organization needs. Nevertheless, attempts to search for the right process continue.

The aim of this research is to help process designers analyse and understand process models in a systematic manner. With that aim, we have developed a visual notation to express process specification and a method for process analysis. We also

demonstrate the use of the method by analyzing a variety of processes and studying their comparative behaviour.

1.2 Basic Terminology

In this section, we introduce basic software development terminology used in this thesis.

A **software process** (Sommerville 2001), or software development process in full, is a set of activities and associated artifacts for the main goal of producing a software product. A smaller unit of process is task or job. In this thesis, the term *process* is used as a synonym to the terms *methodology* or *method*.

A **process model** is an abstract representation of a software process. A model may represent single or multiple aspects of a process at a certain level. A model can be descriptive or prescriptive. A common process model describes a process in terms of activities. Most models are prescriptive as a descriptive one is hard to construct.

Process analysis is the study of software processes to understand their relationships. The analysis usually begins with a qualitative study and may become more quantitative in later stages.

Software development methodology (Imperial College Department of Computing 2006) is an organized, documented set of procedures and guidelines for one or more phases of the software life cycle. It usually includes a diagramming notation or guideline for documenting the results of the procedure. Examples of methodologies include Waterfall model, Spiral model, Extreme Programming, etc. Elements of methodologies include skills, teams, techniques, activities, process, work products, milestones, standards, quality constraints and team values. Generally, software process does not include suggestion on what tools or methods to use.

Process knowledge is an understanding of software processes gained through experience or study. It includes objects, concepts and relationships. A common form of knowledge is a collection of facts and rules.

1.3 Motivation

Perceiving that time-to-market is critical, a time consuming software processes analysis may be seen as an overhead to software projects. While large scale projects may not consider the analysis as a time-consuming process, this could be difficult for small scale software projects or organizations with limited resources. Following the principles of recently developed Agile processes (Highsmith 2001), such organizations could consider minimizing unessential processes and tools to prevent an ongoing software project from slowing down. Unlike traditional processes which focus on sequential and rigid activities, developers of internet speed software development processes, Agile processes in particular concentrate more on making frequent delivery to the customer. Design is on-the-fly and is only performed as needed. Each development phase occurs in short interval to improve time-to-market. This poses a challenge to put process analysis into practice with this process genre.

The recent development of new process models also put forward the need for process study. Firstly, it creates a question of which process to select or adopt for a particular project. A process engineer or a development team is faced with several related issues: Can they believe the hype generated for certain processes which are in fashion at a particular time? How does a new process compare with other processes? Which process is best suited for a particular project or a particular company? Findings from process analyses can provide crucial information for answering such questions. Certainly, people involved in software development should understand the process used to develop the software product in order to deal with complexity and unforeseen difficulties that may occur during the development process. Moreover, the existence of process diversity could result in confusion with respect to process understanding and process selection. These circumstances lead to the need for process analysis and process classification.

In addition, current software process trends have illustrated the need for software process knowledge to be collected and used in different parts of software development. At the beginning of a software project, process knowledge is required for efficient methodology or process selection. During development, process knowledge can constructively assist in tailoring processes. In order to provide

appropriate process knowledge to software projects, a thorough process analysis needs to be done.

The main benefit of process analysis, in general, is providing a comprehensive picture of the development processes. However, the value offered goes further than the comprehensive picture. The observation made during the analysis will be able to aid in shaping our focus on those aspects of the development processes and decision making that need attention. Alternatively, process analysis can provide constructive lessons for other areas of process research.

To facilitate software process analysis, a notation for process representation should be in place. The language to support process analysis should represent software processes in a consistent way and, at the same time, provide users with ability to analyze processes or methodologies it modeled. A suitable process representation language and an appropriate analysis framework when combined can help process studies and discover useful process insights.

Whereas systematic analysis of processes plays an important role in improving the overall quality of software processes, process analysis has not been brought into attention to such an extent as in other prominent process research areas. Although, process modeling research have produced a number of sound modeling approaches, only a small number of those approaches offer support in process analysis.

A number of process analysis methods focus on examining a particular type of processes. Examples include studies of differences between the prescriptive process model and the actual execution of the model, and comparison of processes in similar domain. Others focus on proving correctness and completeness of the process model by examining model properties. Unfortunately, only a few works have been done regarding studying different kinds of processes. The aim of this research is to create a simple and sound analysis method that allows analysis of multiple processes and classification of them for the study of process diversity and facilitate tasks related to software processes.

Several other process analysis techniques have been proposed in the literature. However, many of them require mathematical backgrounds in order to understand and perform analysis. In contrast, with continuously changing requirements and emerging technologies, an increasing number of non-technical people are involved in software

projects. For example, many recent methodologies (e.g. Extreme Programming) have included clients as a part of the software development project in order to get continual feedback and to generate test data. It is not appropriate to train them to be able to tackle formal process analysis prior to the beginning of a project. We require less formal process notation and approaches to process analysis.

The first challenge in doing so is to provide a systematic approach for process representation that is simple and yet powerful enough to capture essential process characteristics necessary for supporting process understanding and analysis. The primary challenge is to design an analysis method that supports analysis of multiple processes without having to add too much overhead to the software project.

1.4 Thesis Contribution

This section describes research problems and the novel approach taken to address those problems.

1.4.1 Research problems

Before beginning a software development process, the process engineer identifies the process to be followed in order to achieve the intended software product. The process engineer may either design the process from scratch or reuse/modify an established one. Building a process from scratch is mostly an inefficient approach since it is obviously time consuming. In contrast, adopting an appropriate process model or reapplying earlier experiences from completed software processes seems to be the better choice.

Unfortunately, the great variety of established processes does not make the latter choices easier. Some software development processes are designed for specific application domains (Colbert 1989). Several others are developed to address specific development problems (Boehm, B. 1996; Mills, Dyer & Linger 1987; Paulk 2002). Yet others are designed for different project sizes (Abrahamsson et al. 2003). This diversity makes it difficult to choose the right process.

Extensive research attempts are conducted to study and improve software processes in the hope of improving the quality of the developed software. While

criticality of the role of software processes is widely accepted (Fugetta 2000), questions raised in search for an appropriate process (Ambler 2003, 2005; Toth 2005) reflect how little is known about existing processes and their relationships. The problem is how the question gets answered. There is a need for the support to broaden the realization of process diversity and lead to the improvement of software processes as a whole.

Software processes have been studied for decades. The results of studies have provided the software engineering community with various methods and tools to deal with development processes. However, the studies are often limited to particular processes whereas the study of multiple software processes and their insights is rare. These issues when compared to others do not receive a reasonable amount of attention. This problem directs us to focus in creating a systematic method to studying existing processes to understand their principles, discover relationships between different parts of the process and, furthermore, relationships between different processes. The results of process study, which incorporates process knowledge, would provide valuable information to be used in different development phases for software projects. In addition, it would provide insights and answers to many questions regarding software development processes such as inter-process relationship and process selection.

Process selection is a decision-making activity which requires considerable amount of information for making right decision. In a large scale software project, where there is enough finance, software process expertise may be hired during software development to do this job. However, this may not be practical for small to medium size software projects which are currently dominating the software industry. When process expertise or process knowledge is not available, software engineers have to rely on their own knowledge in choosing processes for their software projects.

The situations described above require an effective mechanism that will lead to sustainable process knowledge for any software project. Before reaching that part, the study of software processes and methodologies is required. An appropriate process analysis framework will enable the study of relationships between several processes or methodologies and provide knowledge to be used in several parts of software

development. The selection of an appropriate process could be done by way of process analysis.

Another major problem of software processes is their complexity (Dandekar, Perry & Simplification 1996). Process analysis would provide useful information to software project personnel and promote better understanding to software processes.

An appropriate process analysis method is a step towards providing process knowledge to software process organization. Software processes should be analyzed to provide the source of process knowledge. In this thesis, we contribute a software process analysis method based on process representation and process abstraction.

1.4.2 Research Contributions

The major contributions of this research are:

- ***A new visual process description language***

  The need for a process analysis method to realize process diversity has created the need for process representation. Software processes are often described in informal natural language text, casual diagrams, or a combination of both. Since there has never been a standard for describing software processes, although the process descriptions are easy to create, their ambiguity can be problematic. Frequently, the brief and vague information provided by these descriptions leaves readers behind with questions. In many cases, nicely illustrated graphical descriptions do not provide sufficient information. This obviously shields readers from interpreting the intended meaning of the process author. Even worse, it may lead to incorrect interpretation of the process. An example of this case is the graphical representation of the original Spiral model (Boehm, B.W. 1988) which has been interpreted differently by various researchers. Furthermore, some formatted representation of software process description may require some level of training (Grigorova 2003). These issues indicate a need for a process representation that can convey software process information clearly, and also straightforwardly.

  In this thesis, we define a process description notation called *Order*. The notation is developed with the aim of providing a consistent visual notation for process

descriptions. The notation is intentionally designed with ease of use in mind given that modern software processes encourage non-technical personnel (e.g. customers) to be part of the development process. We demonstrate the notation's use in process descriptions as well as in process understanding through step-wise decomposition.

- ***A novel method for Software Process Analysis***

Proper process analysis will improve the understanding of process elements and their interaction. It can also help people to deal with complexity and unforeseen difficulties that may delay the delivery deadline. Considering all factors affecting the development, process analysis can help software engineers to control, predict, evaluate and, eventually, improve their software processes or select and adapt new ones.

The thesis proposes a novel analysis method for software process descriptions. The method takes a process description, transforms it into a standard form and extracts an abstract view of it. It then measures process properties for comparison. A number of metrics are proposed as part of the analysis method.

- ***An extensive analysis of six well known software processes***

As a further contribution of this thesis, we demonstrate the usefulness of the process analysis method by applying it on six of the well known software process models which come from two families—heavyweight and agile. In the analysis we look at a systematic way of studying what aspects of a process make it belong to a family, and how far does a process confirm or fail to confirm popular notions. We also perform a clustering analysis based on the measured values to determine the closeness of one process to another.

1.5 Thesis Structure

This thesis comprises of seven chapters. The contents and the organization of the chapters are described as follows:

Chapter 1 provides an introduction to the issues in software process analysis research. It describes research problems and related research background. Important

research terminologies are defined in this chapter. Additionally, it explains the motivation for this research and the contributions of the thesis.

Chapter 2 contains an extensive literature review on software process modeling and analysis research. The survey starts with a definition of software development process. Then, software process representations are discussed. Moreover, process analysis techniques, i.e. formal methods and empirical methods, are conferred. Furthermore, possible applications of process analysis which include process validation, process verification, process comparison and classification, process management, and process improvement are described. Finally, the chapter is summarized in the last section.

Chapter 3 discusses the process representation notation, "*Order*". An example of the modeled Extreme Programming process is given as an illustration of Order in aiding step-wise decomposition of process descriptions.

Chapter 4 describes the software process analysis method, SAM. It begins by restating the detailed research problems. The main elements of SAM are discussed and the analysis of the Waterfall process is used as an example to illustrate the method.

Chapter 5 describes the application of the process analysis method in real processes. It defines the metrics used in the analysis. Six representative processes are used, namely, Waterfall model, Spiral model, Extreme Programming, Scrum, Crystal Clear, and Feature-Driven Development. A comparative analysis and several interesting findings are discussed at the final sections of the chapter.

Chapter 6 discusses the analysis results based on software process characteristics, emphasizing on process structure and process behavior. Then, a process classification is attempted, using manual analysis and software-based analysis.

Chapter 7 presents the conclusions and contributions of this thesis. It also suggests future research directions.

Chapter 2

Literature Review

2.1 Introduction

This chapter surveys the literature related to the research problems stated in Chapter 1. Software process, process modeling, process analysis and their relationships are discussed in this chapter.

In order to develop an appropriate process analysis method, two components are critical: a model to represent the software processes and techniques for analyzing the models. In process research, software processes are viewed as a model that can be formally presented, modified, managed and improved. Process models also facilitate other tasks that require the understanding of software processes. Such tasks include process selection and adoption, process verification and validation, process improvement, and process management. Likewise, the selection or development of approaches to software processes should consider the benefits beyond their intention. Based on this concept, the conceptual model is likely to be more appropriate for designing a process analysis method.

Software process analysis research has contributed to software process community by providing approaches and tool sets to study and analyze software processes. The aim of such research is ultimately to make a contribution to software process improvement. Typical analysis concerns examining processes for validation or verification purposes: to discover problems, to investigate areas for process improvement, and/or to gain a better understanding of the models.

This chapter is organized as follows. In Section 2.2 we discuss the concept of software processes, its significance and key issues that can be improved through process analysis. Next, Section 2.3 addresses the significance of process representation and the support for process analysis. This is followed by Section 2.4

which presents common analysis techniques and the application of process analysis. Next, process analysis research and its contribution to software engineering community are presented in Section 2.5. It discusses how process analysis benefits to those areas and what issues should be considered. Finally, Section 2.6 gives the conclusions including a discussion of the issues in software processes that needs to be addressed.

2.2 Software Development Processes

According to Humphrey (Humphrey 1989), a software process is a total set of software engineering activities needed to transform a user's requirements into software. Some user requirements and operational settings may not be known at the beginning of software development. Obviously, developers would like to know them in the early phases. However, in reality, new requirements are often identified after the development has started (Berry 2002). These previously unknown requirements are highly unpredictable and, thus, very difficult to handle (Cugola & Ghezzi 1998). Even established written requirements can change during the software process life cycle. This could add enormous complexity to the development.

While software requirements are difficult to manage, a well-defined software process is not (Heimann et al. 1997; Min, S.-Y. et al. 1997). A software process provides a software development team with appropriate practices to deal with new requirements and appropriately tackle changes. Therefore, it is essential for the software development team to understand the employed software process (Yu & Mylopoulos 1994). For that reason, users need to learn about software process itself as well as how to deal with other factors affecting the process. As a result, reasonable effort has been put into studying software processes in order to analyze, manage and improve them.

Software processes encompass activities, such as technical, managerial and various kinds of supporting processes, from gathering requirements or user expectations to delivering a quality product (Dalcher 2005; Derniame, Kaba & Grah 1999). A software process, apart from representing the flow of development activities, also represents communication and interaction between people at various stages of the process cycle. The scope of a software process includes the entire life

cycle of software production (Singh 1996). In this thesis, we focus our study to the development process where developers lead most activities. The typical development process consists of the following activities; requirement specification and analysis, design, implementation (or coding), testing and delivery (IEEE/EIA 1998).

Software process activities may be presented at different level of detail (Jacobs & Marlin 1994; Nguyen & Conradi 1994). Generally, a process is defined in coarse grain manner and later refined (Verlage 1997). In addition, software practitioners and researchers have put in efforts to understand and improve software processes. Certainly, appropriate supports are necessary at different stages for achieving effective implementation of a software process (Lott 1993). At the beginning of the production, a uniform and comprehensive understanding of in-use software process to everyone in software development teams should be established. Later in the process, efficient tools are required to provide vital and useful information for process implementation, as well as, support cooperation and communication to smoothen the implementation. Ultimately, qualitative and quantitative data collected during the implementation can be used to validate whether the process is implemented according to its original goals, and process improvement can be planned.

In the next section, we present a survey on software process representation which will lead to the selection of an appropriate representation format.

2.3 Software Process Representation

Process representation (often referred as process modeling) plays an important role in process understanding and also at later stages of the development process. One of the challenges in software process modeling is to create a model which is sufficiently abstract but does not decrease the degree of communication. A process can be modeled in various ways. For instance, a goal-based process is modeled based on its goals. Goals in software process may be 'create iteration plan', 'hold release planning meeting', 'test unit' etc. An activity model may be analyzed based on artifacts they produce. In this way, it can be determined that the process produces correct artifacts.

The concept of a software process model is described below. Next, some of the main process modeling approaches are presented.

2.3.1 Software Process Models

A software process model is a conceptual representation of a software process. A model may represent selected view points, i.e. activity, goal, interaction etc., of a process in the form of text, graphics or a mixture of both. The model may be constructed as descriptive or prescriptive (Scacchi 2001). A descriptive model provides a description or explanation of a process, whereas a prescriptive model recommends activities that should be performed in a software process. Noticeably, prescriptive models are more commonly available as it is more complicated to collect real software process data to construct a descriptive model. Descriptive models, although are specific to the actual processes examined, provide beneficial information that is more useful to process improvement than prescriptive models.

A process model should be comprehensive, practicable and executable. An appropriate model for software process should be easily understandable and provide instrumentality for process model improvement.

An efficient study of software processes and models can make it easier to retrieve process knowledge, both trivial and non-trivial. Trivial knowledge can be obtained directly from a model. Non-trivial knowledge, however, needs to be discovered. The question is "How can one be exposed to such kind of knowledge?" The knowledge can serve as a basis for many tasks performed on processes. Being able to know better about the process can surely guide or prevent (or at least catch and rectify) errors that may happen during production and make a quality product available at the end of the process. Process modeling approaches are detailed in Section 2.3.2.

Although different models pose different views to software processes, one thing they share in common is providing a systematic way of making software development processes explicit. Carrying out a construction of a software product without having a construction process defined may lead to failure of a project. Thus, having a software development process appropriately defined could help ensure that delivered software products are in control of schedule, budget and quality.

Apart from representing software process, a process model can be used for different purposes. Firstly, a model may be used to guide process performers, control and manage the software process from start until the delivery of the final release. Secondly, a model precisely defined can be used to provide different levels of support to process participants such as simulation, automation and optimization. Thirdly, it can be used to educate people. By providing precise description, one can understand a process better and the communication among team members can be improved. Fourthly, a model can identify how the responsibilities will be allocated and when a process can be redesigned. Last but not least, there are other possible uses of software process models, such as the introduction of a new process in an organization and personnel training/motivation.

A number of process modeling approaches, sometimes referred as process languages, have been developed and the focus is centered on their formalisms, expressive power and their support to software process execution. The benefit of having rigorous process models is that processes can be analyzed and their properties can be examined. Analysis facility is needed to provide justification for correctness, completeness and consistency

Apart from using models to serve what they can do best, they may well be viewed as a repository of process knowledge which can be accessible through an appropriate technique. To obtain knowledge of a specific subject, we need to utilize knowledge embodied in software processes and models. Some parts of knowledge can be accessed directly from the models and some not. Process analysis, which is the main focus of this thesis, can help in drawing process knowledge. Providing appropriate techniques to retrieve such useful information about software processes and models helps in better understanding of software processes.

2.3.2 Process modeling approaches

Software process models vary in representation styles, interfaces and perspectives. Some perspectives of software process may require a particular representation style and an appropriate type of interface. There are different approaches used to model software processes. One paradigm may be preferable over the other in relation to specific requirements, and in some cases, a mixture of paradigms may be used. Appropriate process modeling approaches are, therefore, required to sufficiently

describe key elements of process components with precision, and to be able to deal with particular types of process models.

Early attempts to describe software processes are by using documentation which, despite using well structured narrative descriptions, could be voluminous thus making process information retrieval difficult. Other alternatives to process modeling paradigms have been proposed, e.g. flowchart-like diagrams, graphs, trees, pictorial languages, low-level coding languages, etc.

Some paradigms are borrowed from system modeling. Others are developed specifically for software processes. Software process modeling paradigms, sometimes referred as process languages, can be classified according to the technique used to model software processes. Armenise et al. (1992) proposed a classification scheme based on representation style and interface type, and identified the main approaches to software process modeling based on representation styles, e.g. logic rule, automata, imperative programming language, artificial intelligence, event-trigger based formalisms and abstract data types. Osterweil and Heimbigner (1994) suggested the use of multiple modeling paradigms for software process modeling as an alternative to using a single paradigm.

Understanding modeling paradigms helps one gain knowledge of how a process is modeled and, more importantly, could aid in selecting the most appropriate process modeling paradigm for the modeling task. Included in the following subsections is a brief introduction to main process modeling paradigms along with their advantages and limitations.

2.3.2.1 Rule-based Languages

According to IEEE standard glossary of software engineering terminology, a rule-based language is a nonprocedural language that allows the user to define a set of rules and to create queries or problems based on these rules (IEEE 1990). A rule-based system is a relatively simple model. It incorporates all appropriate knowledge into a bunch of if-then rules that tells what should be done or not done under different situations.

Rule-based process models are represented in terms of rules that describe development processes. Those using rule-based approaches are Articulator (Mi & Scacchi 1990), Marvel (Finkelstein, Kramer & Hales 1992), Merlin (Junkermann et

al. 1994) and EPOS (Jaccheri & Conradi 1993). Articulator is a knowledge-based environment for engineering organizational processes. It uses a rule-based object-oriented knowledge representation method to model organizational resources. MARVEL is a software development environment that models software engineering process activities as rules. In MARVEL, a model is specified using project rule sets. The body of a rule consists of variables, logical condition, an optional activity which may be invoked according to the evaluation of conditions, and a set of effects that states each activity's alternative results. It supports process execution by means of forward and backward chaining over the rule base. MERLIN is a software development prototype that uses a rule-based technique to describe and execute software process models (Junkermann et al. 1994). The model defined in a rule-based language describes development activities and corresponding documents. An activity whose conditions are all true could be performed.

There are differences in these prototypes implementing rule-based techniques on process modeling. Articulator uses a resource-based representation scheme to model organizational processes while MARVEL and MERLIN models software processes based on development activities, and they provide different methods for process execution. A rule-based language offers declarative expressive power. It allows real-world process knowledge to be represented in a consistent format. A major draw back, however, is that it is difficult to view the flow of control of a system specified in rule-based style, textual rule-based modeling language in particular.

2.3.2.2 Petri Nets

Petri nets are often used to model software processes. SPADE (Bandinelli et al. 1994) and FUNSOFT (Deiters & Gruhn 1994; Emmerich & Gruhn 1990) are examples of Petri net based process modeling paradigms. Petri nets, which originated as a graphical and mathematical modeling method for information systems, have been applied to software process models. At the first glance, Petri Nets may look similar to flow-charts, however, unlike flow charts they can precisely specify when each activity is triggered. In fact, they can be used to express mathematical functions of models. In addition to application in software process modeling, Petri Nets have been used in business workflow models, formal methods, hardware design, etc.

Standard analysis techniques of Petri net includes proof of properties and calculation of performance (Grigorova 2003). Although there are success stories in several application domains, Petri nets are not widely used in software engineering, and successful methodologies often suggest alternative modeling languages, e.g., SDL or Statecharts (Denaro & Pezzè 2004).

Advantages of Petri Nets include ability to include mathematical equations and formal analysis facilities to software process models. It is also able to represent asynchronous concurrent activities in software processes very well. However, extension to Petri Nets is required to satisfy the requirements of process modeling. Another major disadvantage of Petri Nets is its complexity. The model can grow too large and may complicate analysis, and, thus, require assistance from a computerized tool. It also appears to lack scalability when applied to a process situated in a distributed environment context.

Petri nets have several disadvantages despite its modeling power, analysis capability and broad recognition in software process research. Firstly, the operational semantics and formalism are difficult to understand for many software developers who lack the required mathematical training (Balser et al. 2004). The primitive nature of Petri net constructs places a burden on process modeler (Berson, E. Souza e Silva & Muntz 1991). Secondly, Petri net size can become very large for real world problems (Hofstede 1993; Petit 2002). Thirdly, there is no standardization for Petri nets (Bosilj-Vuksic, Giaglis & Hlupic 2000). Although Petri nets can be classified into three main classes: ordinary Petri nets, abbreviation and extensions (David & Alla 1994), many versions of these are created and extended to solve particular modeling problems. Because of these disadvantages, computer-based tools to support modeling and analysis of Petri nets are necessary.

2.3.2.3 Process programming languages

Osterweil (Osterweil, L. 1987) proposed the idea of using a programming-like language to model software processes. Since then the idea of executable process models has become a key element of software process modeling (Cass et al. 2000; Lerner et al. 1998; Osterweil, L.J. & Heimbigner 1994; Sutton, Heimbigner & Osterweil 1995). By allowing a process model to be encoded as a process program, a model becomes executable.

APPL/A is an early example of a process programming language extended from Ada (Sutton, Heimbigner & Osterweil 1995). APPL/A is a language aimed at managing relations between software objects and processes. The language paradigm is a mixture of procedural and reactive programming. However, some aspects are represented using rule style. Software developers who have a background in programming languages may find it easier to learn and use these languages.

Little-JIL is a recently developed process definition language that describes the coordination of activities carried out by external entities (Lerner 2003; Lerner et al. 1998). The language employs a graphical representation of process steps. It uses programming and finite state machines for checking of process model properties. Finite state machines written in their process algebra called FSP is translated into a Labeled Transition System for Analysis by a model checker (Lerner 2003).

Despite the advantage of executable capability of process programming languages, the low-level coding style of such languages makes the creation of understandable process models particularly hard. Like software requirements, certain software process requirements may not be known before hand and, therefore, process changes may be difficult. In addition, low-level coding languages may limit creativity of process designers as it is hard to see the control flow of the process, and could cause problems in understanding and communication. Graphical notations can overcome this problem better.

2.3.2.4 Graphical representation

While process programming languages enable process execution, its textual form is a significant disadvantage. Against these languages, graphical representations such as diagrams or graphs are introduced to produce conceptualized models. It could increase understandability and facilitate communication and information exchange (Allgood et al. 1994). Carefully selected graphical representations can readily convey significant amounts of information (Kellner 1989). The use of such visual illustrations makes it easy for non-experts to provide graphical description of a process.

Graphical representations of software processes is nothing new. Simple flow charts haved been used to describe processes. Some of the early and most commonly used graphical tools are PERT (Sommerville 1992) and Gantt charts (Liu &

Horowitz 1989) which are commonly used for process management (Hazeyama & Komiya 1992).. A PERT chart, also known as a Network chart, is mainly used for project planning and management. It represents a model of boxes and arrows linking tasks to show dependencies. A Gantt chart is a type of bar chart that represents a project schedule.

Recently developed process representations lean towards incorporation of graphical representations to its formalism in order to improve their communication capabilities. Although mixed with graphics, formal approaches may not become easily understood and may cause difficulties in other parts of development processes (Min, S.Y. & Lee 2000). When the strong emphasis is on end-user, a graphical representation to process modeling should be incorporated.

Each process modeling paradigm poses different requirements to process modelers. A single paradigm may not be able to very well capture important aspects of software processes. A mixture of paradigms such as formal and informal methods or other combinations of two paradigms in software process modeling may resolve the problem regarding formality of representation language, and could be attractive to process modelers and non-technical users (Abeysinghe & Phalp 1997)

2.4 Process Analysis Techniques

A software process can be quantitatively and/or qualitatively analyzed. Software processes are often analysed for completeness, correctness, consistency, etc. Completeness analysis can help in detecting incomplete process specification. An example is a process with no input or missing output branch. Consistency analysis can help determine inconsistency in process specification. Correctness analysis can determine whether a software process is correctly modeled. For example, it can resolve whether all process properties are given correctly.

Analysis methods may include data collection, analysis and conclusion of findings. Most stages require human participation. For example, data collection may involve interviews, observation and data recording. Then the analysis may be performed by reviewing and interpreting collected data. Those steps unquestionably add extra time and cost to the analysis process and, consequently, software development. Some researchers have proposed the use of formal methods and others

empirical studies. The following subsections discuss literature on software process analysis.

There are several approaches to software process analyses and a number of tools provide support to them. The analysis tools should be able to provide useful information to process designers to assist them in process learning and process improvement. A process analysis capability may be provided as a part of the software process modeling approach or the process engineering environment. In fact, only a small number of modeling approaches offer process analysis. On the other hand, most software process engineering environments provide different ways of process analysis to validate completeness, correctness and consistency of software processes by analyzing process model properties. These analyses are generally targeted in detecting and preventing common and invisible errors and at the same time improving quality of the software process.

A number of techniques are employed in software process analysis. Subsections below briefly discuss how each technique facilitates process analysis.

2.4.1 Formal Methods

A Formal method involves a strictly defined logic system and notation. The most common formal methods include formal specification and formal verification. The significant advantage of formal methods is preciseness and robustness in its specification that is useful for verification purpose (Hall 1990). Formal methods have been applied to several fields of software development research (Plat, Katwijk & Toetenel 1992). In software process research, the techniques have been applied to various stages of software processes; particularly process modeling and process analysis. A formal specification facilitates formal analysis of software processes. Types of analysis are based on formalisms used to specify processes (Min, S.-Y. et al. 1997). The common formal methods used in software process modeling and analysis are Petri Nets (see Section 2.3.2.2) and process programming (see Section 2.3.2.3).

Semi-formal methods, especially UML (Pender 2003), provide an alternative to fully-formal approaches such as Petri nets. Various examples of UML application in software process modeling and other areas illustrate its recognition (Dobán &

Pataricza 2003; Jager, Schleicher & Westfechtel 1999; Lopez, Franch-Gutierrez & Ribo-Balust 2004; Nitto et al. 2002). UML, although widely used in industry, is very vast. The vastness results in requirements for a particular set of skills for effective production of UML models (Hailpern & Tarr 2006). This may discourage people from learning (Hegde 2007) and using it. While the industry has spent loads of financial and human resources on learning UML and UML tools, the benefit in using formal methods as well as UML still remains a question of whether it comes from the formalization process or the end result (Wing 1990).

One obvious problem with the formal approach is that the model is harder to understand and, therefore, can affect communication among stakeholders. Formal methods often require training before the model can be fully utilized and understood (Pan 1997).In modern methods such as agile processes where customer interaction is critical, opportunities for training of customers to understand formalisms are almost non-existent.

Software processes unquestionably need some level of formality. However, not every project can afford investment in learning and using formal methods, either financially or personally. Formality introduces precision while limiting the number of people who can participate. An informal method or a mixture of methods may not be able to provide a similar level of precision. However, it encourages involvement from stakeholders other than those from technical background. There are also other factors that affect the selection of approaches used in software development. Small businesses and organizations have found the difficulties to implement software standards adopted by large organizations, (Fayad, Laitinen & Ward 2000; Leung & Yuen 2001). Small projects need processes and methods that can be easily applied.

2.4.2 Empirical Studies

Empirical studies deals with evidence. Empirical studies may be case studies (Kirk & Tempero 2005), experiments (Garcia et al. 2005; Raffo et al. 1999), discussion groups (Coleman 2005), etc. The study can be done in both quantitative and qualitative research. Statistical methods can be used to analyse quantitative data, whereas it is difficult to do statistical analysis on qualitative work.

The studies carried out in software processes area look at how aspects of process, product and people affect each other (Kirk & Tempero 2005). The fact that empirical studies are based on real evidences supports the reliability of the studies. However, it is often found that several studies done on the same object do not produce consistent results (Kirk & Tempero 2005). Data collection can become costly and labor-intensive (Wolf & Rosenblum 1993). In the work proposed by Wolf and Rosenblum (1993), process event data is captured from the process under development and analyzed. Relationships between event data are discovered. The design of process improvement is then guided by findings from statistical analysis. The kind of empirical analysis that requires data collection from ongoing development (Coleman 2005) will be useful in some cases. For instance, when the cost of conducting the analysis is not high or when the finding can reduce a great deal of the cost for future enactments.

Collecting expert knowledge is an alternative to field data collection. With the help of computer and network technologies, the cost of data collection can be reduced. Lindvall et al. (2002) held workshops in which experts discussed and recorded their findings on Agile processes. Expert experience and opinions are highly valuable. However, gathering a number of experts is not trivial and repetition of the events depends on their availability.

There are other studies which are based on periodical evidences (Abrahamsson et al. 2003; Keller et al. 1993; Nguyen & Conradi 1994; Smith 2001). Data retrieved for those studies reflect the factual data and, thus, are adequate as a constructive source. Real data, though genuinely collected according to the purpose of the study, are specific to intended objective of the study and, hence, narrow the analysis. In contrast, studies where data from difference sources are gathered are likely to broaden the perspective as well as the results of the studies.

2.5 Application of Process Analysis

This section looks at works that stretch the benefit of process analysis further to other tasks such as process validation, process verification, process comparison and classification, process management and process improvement.

2.5.1 Process Validation

A process is considered valid if the behavior exhibited corresponded to its specification (Cook & Wolf 1998; Sommerville 2001). Process validation may involve data collection to create a model of actual execution and analysis of it together with the corresponding process model to see if they have the same pattern.

Validation should show that the process serves its intended purpose. For example, if a process X is modeled to produce artifact Y, when the actual process is executed, X should produce artifact Y according to its model. If not, the process validation would report that the process does not work correctly.

Cook and Wolf (Cook & Wolf 1999) applied a neural network technique and Markov method to software process discovery, and created a method for validating software process models by comparing specifications to actual execution. Two process streams are produced: one generated from the model and the other discovered from process execution. The comparison of two streams is done using the string distance: Simple String Distance (SSD) and Non-linear String Distance (NSD) methods adapted from Levinshtein distance. This technique depends on the data collection of actual process execution.

In other work, Johnson and Brockman (1998) validated models for predicting process cycle times based on execution histories. The focus of their work is not exactly validation. It is rather estimation concerning control flow.

Process validation is a tough job to any type of software processes as the majority of validation costs are incurred after implementation (Sommerville 2001). Added to the cost is the intensive labor required for data collection. Therefore, process validation can be very costly to certain types of software processes.

2.5.2 Process Verification

Process verification is the procedure to check if the process matches its specification (Christie 1993). Verification and validation should not be confused (Sommerville 2001). The availability of tools and research publications in process modeling has inspired a great number of process verification activities based on process models (Gruhn 1991).

When a process model is constructed, its properties may be verified to check if they are accurate. Validation usually occurs after an implementation of a process while verification occurs before the implementation begins. It is necessary to check the correctness and completeness of process specification before executing it.

Brockers and Gruhn (1993) proposed automatic verification of software process model properties. Cobleigh, Clark and Osterweil (2000) performed verification of properties of process definition. Atkinson and Noll (2003a; 2003b) proposed an automated approach to detecting errors in processes. Errors are discovered by analyzing flow of process resources specified in a process programming language.

Other works include the finite state verification system which was applied on the process definition written in the Little-JIL process definition language of an auction example (Lerner 2003). The verification is usually based on low-level representation of processes such as process programming paradigms. Examples of such languages include APPL/A and Little-JIL (Lerner et al. 1998; Sutton, Heimbigner & Osterweil 1995).

Automation of process verification rests heavily on the concept of rigorous process modeling (Gruhn 1991). In some application domain where criticality of the product developed is crucial and the development process must be strictly defined, formalization and automation of process representation and verification is considered important. Considering benefits and tradeoffs, the project manager will be able to justify the appropriateness of approaches required for his software project.

It is possible to apply process verification concepts at a higher level than checking low-level process model properties. Traditional and iterative processes, for example, can be seen as kinds of process specifications. By using verification concepts, one can verify whether a given process matches its specification.

2.5.3 Process Comparison and Classification

Classification means grouping similar things together. Each group is distinguished by at least one characteristic that is unique. Classification aids in discovering relationships among entities and among classes of entities (Prieto-Diaz 1987). Classification reflects what people would like to know about software processes. The

idea of using classification with software processes is to understand and classify process characteristics. When that information is available, it becomes valuable knowledge that can be reused. One can analyze and make use of it for other purposes, particularly software process improvement.

Attempts to compare and classify software models and processes have been around for decades (Scacchi 1998). Process comparison and classification are often performed manually. Automated approaches to these applications can speed up the classification process and save time. However, human judgment is often required to verify the results. Classification is an important subject in data mining. Some of the classification techniques in data mining are applied to analysis of the development process (Goel & Shin 1997).

Ellmer and Merkl (1996) proposed a classification process based on natural language descriptions. Neural Networks are implemented to support process reusing and tailoring based on process descriptions consisting of a process definition document and a process model. This work resulted in a software process library containing process knowledge which can be used further in software project but there is no discussion of how the knowledge will be used. Although it reduces the process modeling effort, the results do not visually present similarity, difference and relationship among classes.

Another effort to classify software processes is proposed by Inchaiwong and Sajeev (2006). The paper presented a software analysis approach based on faceted classification to illustrate benefits of software process analysis. The analysis begins with questions that are used to determine attributes of software processes and its value. Each process is represented as a tuple of these attributes. The tuples were assessed and analyzed. A limitation to this work is that the scale of measurement is ordinal. The assessment of such a measurement is difficult to justify.

Recently Ocampo and colleagues (2005) proposed a method for commonality analysis of software processes. Process descriptions are analyzed and validated if they are similar. A reference model is then produced. The comparison is performed automatically based on Levinshtein distance, and process engineers manually recheck process descriptions to verify the comparison result. Processes need to be formalized into an intended format before comparison begins. This work aims to

select best practices as a base for establishing a reference model and the analysis is targeted among processes in the same domain. Another work by Podorozhny and colleagues (2003; 2005) is an automated approach to analyzing software processes based on comparison of artifacts produced. The comparison was based on the specification language Little-JIL (see Section 2.3.2.3). Similar to Ocampo's work, this system assumed that compared software processes are in the same problem domain and have a similar purpose.

As stated earlier, a great number of research attempts involve either studying a single process or processes in similar domain. The analysis and study of multiple processes appeared to have been overlooked.

2.5.4 Process Management

Some means for systematic analysis of process execution and related environment factors are necessary in order to make reasonable and rational decisions during process management. Some argue that managerial decision making needs to be based on process analysis (Min, S.-Y. et al. 1997).

Management plays a substantial role in software process domain to support planning, scheduling, resourcing, and software production cost control, and at the same time improving the organization's capability to produce better products. Bad management influences everything including quality, timing, budget and product reliability. Software process management focuses on the support of the development process in order to reduce the development time, improve product quality, and also to handle changes that occur during the development period, such as requirements, tools, teams and deadlines.

Many project management tools have been created to help project teams to manage software production. Management tools are developed to support the developers and leaders in communication, rule the process, process tracking, automatically management of resources, team guidance and document generation. In addition, the process management tools are used to define the process definition. The tools that provide various kind of support to software development processes is referred as Process-Centered Software Engineering Environment (PSEE) (Garg & Jazayeri 1995). a process modeling tool is often a part of PSEE. Some examples of

environments that provided management tools on the process definition level are ALF (Canals et al. 1994), Marvel (Barghouti 1992), Merlin (Junkermann et al. 1994), SPADE (Bandinelli et al. 1994) and Process Weaver (Fernstrom 1993). Conversation builder is a process support tool that has a main focus on process managerial support using various types of conversations throughout the development cycle using message servers for messaging issues between developers and managers (Junkermann et al. 1994).

Several process management tools have been built to take advantage of the Internet technology since development teams often require collaborative work. Web-based process support systems can improve the coordination and information exchange of virtual teams (Kaplan et al. 1992). MILOS is an example of a web-based process support system that combines process modeling, process management and the workflow engine (Bowen & Maurer 2002; Maurer et al. 1999).

PSEE should be the future direction of any systematic approach to software processes to provide appropriate support to those development processes which directly affect the quality of the software product.

2.5.5 Process Improvement

Software production unquestionably does not follow a fixed pattern. It needs appropriate customization and adaptation for individual companies and even individual projects. Software processes have associated management and technical activities and they need to be controlled in order to obtain a quality product. Following a defined process, that integrates uses of people, tool, tasks, method and documentation, can guarantee a level of quality. However, management failures and poor processes can affect software production (Flowers 1996). To address this, many businesses and organizations employed models such as CMM for their process improvement program (Emam & Briand 1999). CMM (Capability Maturity Model) was developed by SEI (Software Engineering Institution) to measure the capability level of organizations with respect to software process and define the requirements of quality and levels of achievement.

Process improvement methods have been researched and studied. SPICE and IDEAL are other examples of recognized improvement methods (Fugetta 2000).

An ultimate challenge of software process analysis is to support the process improvement. The concept of process improvement exists in all areas that require effective processes (Ward, Fayad & Laitinen 2001). To improve a process, it requires assessment, elimination of unnecessary processes, restructuring, as well as identification of effective and useful options. Different approaches to process improvement yield differences in costs and benefits (Emam & Briand 1999). Process analysis can certainly offers helpful information that leads to potential benefit to process improvement.

2.6 Summary

The previous sections of this chapter presented related works in process modeling languages and process analysis areas. This section concludes the chapter by revisiting these areas.

In relation to process modeling languages, the main modeling styles: Rule-based, Petri nets, Process programming and Graphical languages were presented.

A software process can be seen as a creative process which can be benefit from graphical tools. Recent process modeling languages are a mixture of graphical notation and selected process modeling styles to combine formality and visualization. Graphical representation to process modeling is encouraged if end-users are regularly involved.

With respect to process analysis, common process analysis techniques were presented and followed by tasks that could benefit from process analysis. Formal methods are often used in process analysis research in regard to its ability to provide analysis functions. However, it will require training and learning time. The other method to process analysis involves collection of process evidences, comparison and analyses. Some data collection methods may be costly and labor intensive.

The availability of process representation languages and different process analysis methods helps process engineers and stakeholders choose and adapt software processes according to organizational and project specific needs. To accomplish this efficiently, they need appropriate support that can assist them in performing routine tasks of process selection and adaptation and help them design, select and analyze processes. In order to select the most suitable process model and

technique for a project, the software engineers need to consider many critical issues such as the appropriate life-cycle, the balances of activities, the effort expectation, and the tolerance in project planning. (Lindvall et al. 2002)

However, traditional methods of analysis pose some problems. Although it offers in-depth analysis, it does not provide much insight into processes on what occurs between each phases (Ocampo, Bella & Mumch 2005). Despite unique challenges posed by software process analysis, not much work has been done in this area.

There are also several problems that interrupt software process analysis research. Yaung (1992) indicates the lack of technical papers on specific techniques for process analysis. Current practices are mainly qualitative (Yaung & Raz 1992) (Zakarian & Kusiak 2000), subjective and inadequately defined.

This literature review chapter has laid down a foundation for this thesis. In the next chapter we introduce our process modeling notation, named *Order* before we use it in subsequent chapters for process representation.

Chapter 3

Order: A Process Notation

3.1 Introduction

A software development process is a prearranged set of software development activities. The ultimate goal of a software process is to deliver the software product as specified in the contract (IEEE/EIA 1998). An example of development activities expressed in natural language is quoted below.

> "The developer shall develop plans for conducting the activities of the development process. The plans should include specific standards, methods, tools, actions, and responsibility associated with the development and qualification of all requirements including safety and security. If necessary, separate plans may be developed. These plans shall be documented and executed." (IEEE/EIA 1998:16)

A unique set of development activities result in a unique development process. The uniqueness of software process can be determined by examining activities occurred during the development process and identifying its pattern. In order to achieve such systematic study of development activities, software processes must be systematically represented. This indicates that the need for expressiveness is one of the critical considerations of software process notations.

Software processes used in the industry vary from the traditional waterfall model to the modern agile processes such as Extreme Programming (XP) and Crystal methodologies (Paulk 2002). A key indicator of the level of maturity in the Capability Maturity Model (CMM) (Olson, Reizer & Over 1994) is the extent to which an organization usually uses a defined process. This has increased the role of software processes in the industry, as well as the importance of defining it correctly and completely. The existence of modern processes, especially Agile processes, and rapid acceptance of these processes from practioners and researchers has produced a significant impact to the way modern day software is developed. The study by Syed-

Abdullah *et al.* (2006) has shown that Extreme Programming (XP) has positively increased the enthusiasm level of software developers. This new discipline also drives the changes in other areas of software development, for instance requirement engineering (Schwaber 2002) and project management (Coram & Bohner 2005). The results-based approach has been taking over the traditional process-based approach (Royce 2002). The influence of these modern processes to the software engineering community is, therefore, made clear.

Undoubtedly, software process is a complicated subject no matter what state it is in; either pre- or post-development. Although it is true that software is an easier artifact to be modified during the development process, it can be very costly (Cugola & Ghezzi 1998). Similarly, modification to software processes particularly during the development is likely to be costly too. The pre-development approach can potentially prevent problems to software projects and, thus, put software project in safe hands.

To support the pre-development approaches, software processes must be modeled. The main purpose of process modeling is to promote understanding and communication among people working in the project. However, all personnel do not need to understand the process in its minute detail. Ironically, a process specified in a natural language in great detail may fail to deliver the messages to the people who use the process. What is required is a way to specify processes systematically at various levels of detail from the macro-aspects to the micro-aspects.

3.2 *Order*

In this chapter, we describe a visual process specification language called *Order* and demonstrate one of its uses as a tool for step-wise decomposition of processes. Several process modeling languages are previously reported in Chapter 2. Compared to them, *Order* provides a visual representation of a software process using the metaphor of *production line in a factory* (Inchaiwong & Sajeev 2004, 2006; Sajeev & Inchaiwong 2002). Visual representations based on appropriate metaphors are easy to use. The success of the desktop metaphor as an operating system command language is perhaps the most prominent illustration of this theory.

Order is later used to prepare software processes for analysis in Chapters 4 and 5. The main characteristics of *Order* are simplicity and modularity. *Order* addresses

simplicity by presenting a sufficient set of activity constructs. Modularity is addressed in step-wise decomposition (Sajeev & Inchaiwong 2002).

Order uses the metaphor where a software project can be seen as a factory that produces software and associated documents. A factory uses several kinds of *production point*s to create the product. Each production point in a factory contains resources and information required to produce a product in demand. A *component* produced by one production point may be delivered to another production point. The final production point could deliver the final product to a customer. Several kinds of production points in *Order* represent software development activities that result in products—intermediate or final.

Each production point has a *rulebook* for specifying rules and constraints that the point should obey, and a *suggestion box* to provide guidelines; as with suggestions, generally, they are not mandatory on a production point. Each artifact used in a software factory has a property page listing its attributes. *Order* provides three kinds of process control—sequential (arrow), selection (diamond) and fire (trigger).

The next section introduces the syntax of *Order* notation. Firstly, the syntactic constructs for modeling main process elements will be explained in Section 3.3. Secondly, the semantic concepts are followed.

3.3 The Software Factory

3.3.1 Production Point

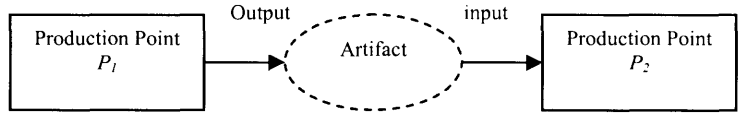
A software project can be viewed as a software factory () where software and associated artifacts are produced. A factory uses several kinds of production points to create the product. A production point is where the development activity is performed. It represents a unit of process activity. *Order* provides three kinds of production points: shop floors (), assembly points (), and workers (). The process designer decides which production point to use. Generally, a Shop Floor represents a complex activity involving both technical and business personnel. An Assembly Point represents a moderate activity which may involve technical or business personnel. A Worker represents a simple activity which may be performed by one or a few people.

Practically, a worker may be used for simple tasks, shop floors for complex activities, and assembly points for those in between. For example, pair programming in XP is a worker activity. Requirement specification in Waterfall, on the other hand, is a complex activity represented in Order as a shop floor. The availability of the different icons help process designers provide a visual signal of the complexity of an activity while specifying a process.

3.3.2 Artifact

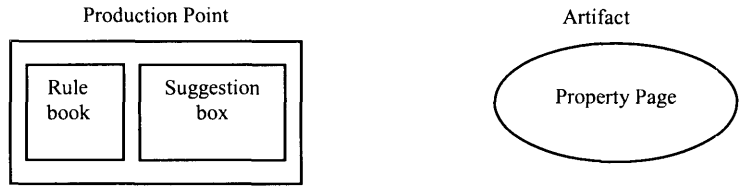
An artifact is an entity required as input at a production point or is a product of a production point. An input artifact of a production point may be an output artifact of another production point (alternatively, it may be sourced from outside the factory) (Figure 3-1). There are three kinds of artifacts: (1) a final product which is the final artifact from the factory () (2) an intermediate part that is a constituent of the final product () and (3) an *auxiliary* () which is an artifact that is not part of the final product, but will help build and/or maintain the product.. For example, in an automobile factory, a wheel is a product since it is part of the car under production, where as a road built to test-drive the car is an auxiliary. In a software project, a test plan, for instance, is an example of an auxiliary where as a user manual is part of the deliverables.

A production point has a number of bays () for receiving parts and sending produced artifacts. (Bays are similar to ports used in some message passing languages).. When there is no ambiguity, the bay icon will not be shown explicitly in the model.



The dash line indicates that an Artifact may be explicitly shown or hidden.

Figure 3-1 Data flow model in *Order*



(a) For Production points (b) For Artifact

Figure 3-2 Specification of attributes in *Order*

Each artifact has a *property page* that lists its characteristics. Each production point has a *rulebook* ( ) and a *suggestion box* ( ). The *rulebook* specifies constraints and rules that the production point needs to follow. The *suggestion box* gives guidelines that the production point may use (Figure 3-2). For example, the rulebook of the painting shop floor in an automobile factory may specify that the shop floor must be dust free, which needs to be obeyed. On the other hand, the painting shop floor may have a guideline in its suggestion box to give priority to cars that require metallic paint; which it may or may not adhere to always.

The rulebook has a predefined template for rules that often occur, and provision for any additional rules that the process designer may want to specify. The rule books and suggestion boxes contain constraints and rules expressed in natural or formal language. For formal specification, *Order* uses Object Constraint Language (OCL) (Object Management Group 2003). OCL is a formal language associated with UML. For example, in OCL, we can specify that a Production Point can start production when at least one of its inputs is ready as follows.

```
Context ProductionPoint::start():integer
pre: self.inputSet -> (exists i : i.ready)
```

(Read this as: a precondition for starting a production point is that there exists an input i among the set of inputs for the Production Point, such that i.ready is true.)

3.3.3 Start and Stop Criteria

Two of the fields in the predefined template of the rulebook are for *start* and *stop* criteria. The start criteria specify the conditions that must be satisfied before the production point can start its operation, similarly, the exit criteria specify the conditions the production point needs to satisfy for the production to complete. A default start

criterion for a production point is that all its input artifacts are available to it, and a default stop criterion is that all output artifacts are produced.

To help with formal reasoning, the rulebook is divided into two sections. One is to give the rules in natural language, and the other formally as logic expressions using pre-conditions (start criteria), post-conditions (stop criteria) and invariants (Figure 3-3). Processes used in the production of safety critical software systems probably would want to use the formal specification of rules, whereas other software houses may be happy with the natural language listing of rules.

(a) Sequencing control (b) Decision making control

Figure 3-3 Control flow model in *Order*

A production point is automatically switched on when its start criteria are satisfied. It terminates when its stop criteria are satisfied. This means that that there is no special notation needed to indicate concurrency. All production points that have their start criteria satisfied can run concurrently. Sequencing, on the other hand, is achieved by specifying appropriate start criteria, usually, graphically.

A production point will restart itself when new items appear in its input bays satisfying its start criteria. A production point can also be restarted by using an explicit trigger (). Whenever the trigger is generated, the production point will start again provided that its start condition is also satisfied.

3.3.4 Other aspects

Additional attribute to production point or artifact may be defined. For example, a role in *Order* may be listed as a property of a production point or artifact. There is no special notation to indicate it. A process designer when drawing a production point or artifact has to specify this to its property.

3.4 Process Decomposition

In this section, we describe how *Order* can be used to specify a process model. *Order* achieves the specification through step-wise decomposition. This helps a process designer to design a process by capturing the high-level aspects first, and then repeatedly decomposing the high level components to required detail.

The visual syntax for process decomposition is a rectangle (Figure 3-4(a)) to which the production point that is decomposed is attached. In other words, the decomposition is shown within the rectangle. Bays (connector end points) attached to the border of the rectangle are used to connect the decomposition to external connectors. An example of process decomposition in the next section will illustrate this point.

A *flattened* decomposition will remove the rectangle and bay(s) and show the result as a single layered structure.

(a) (b)

Figure 3-4 The visual syntax for process decomposition: (a) Decomposed process and (b) Iterative process

Often, certain sub processes need to be performed several times. The number of iterations may vary from project to project. *Order* shows an iterative process by a soft-edged rectangle (Figure 3-4(b)) over the icons. The range (e.g. 0..10) of possible iterations is a property in the rulebook of an iterative production point. Each instance of the iteration can be referred to using an index-integer.

3.4.1 A Case-Study

In this Section we illustrate the step-wise decomposition of the Extreme Programming (XP) process using the *Order* notation.

An XP process has three major stages, *Release planning*, *Iteration and Acceptance testing* (see Appendix A for the process model of XP). The Release planning stage results in a collection of *stories* which when implemented completes the

project. (Stories capture the requirements in a way similar to user cases (Cunningham & Cunningham 2007).)

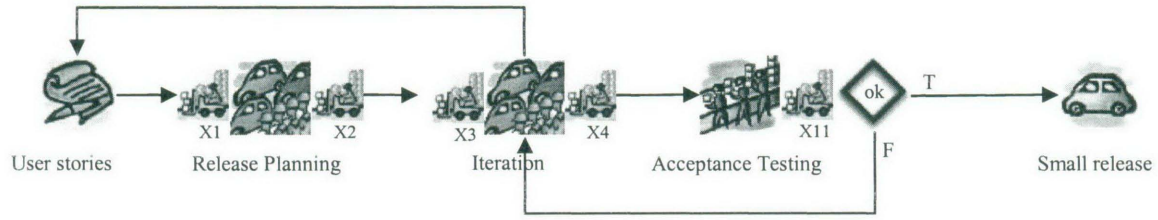
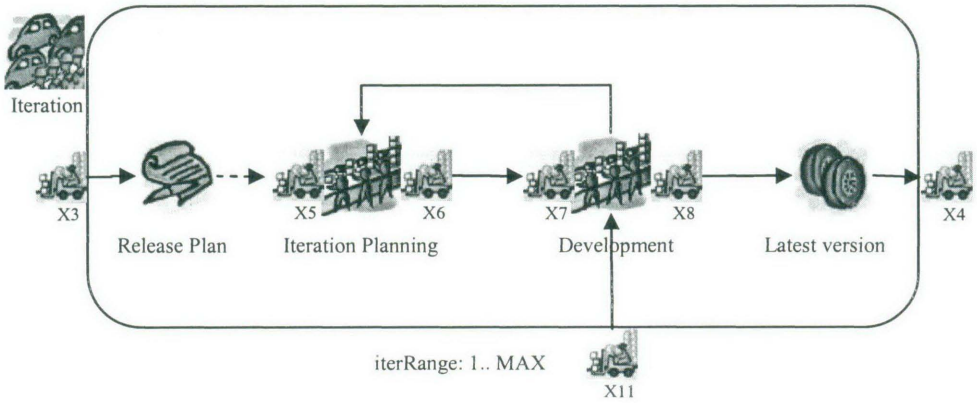


Figure 3-5 The high level view of XP process



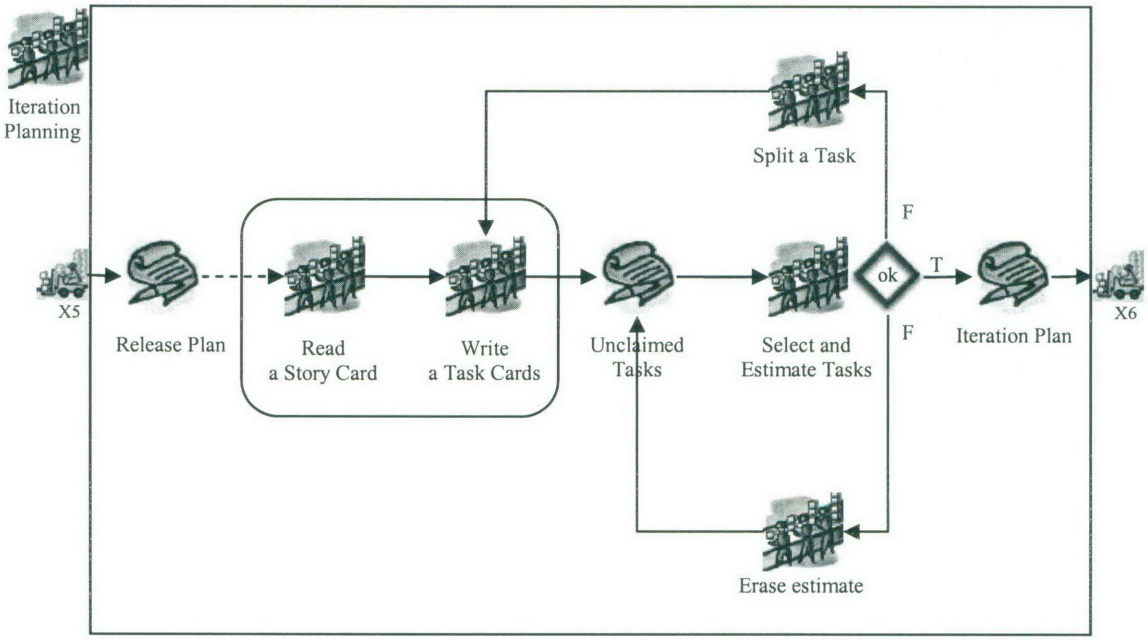
Note: *iterRange* is the range of possible iterations referred as an index-integer.

Figure 3-6 Iteration decomposition of XP process

The top-level view of XP is shown in Figure 3-5 which identifies input bays (X1 and X3) and output bays (X2, X4 and X11). Figure 3-6 shows further details of the Iteration production point. Comparing Figure 3-5 and Figure 3-6 shows that X2 is where the output from *Release planning* come into the *Iteration* phase and X4 is where the output of the *development* phase goes out of the *Iteration* phase. The process designer may select to show or hide a selected artifact. To minimize complexity, the artifacts in subsequent decomposition are hidden.

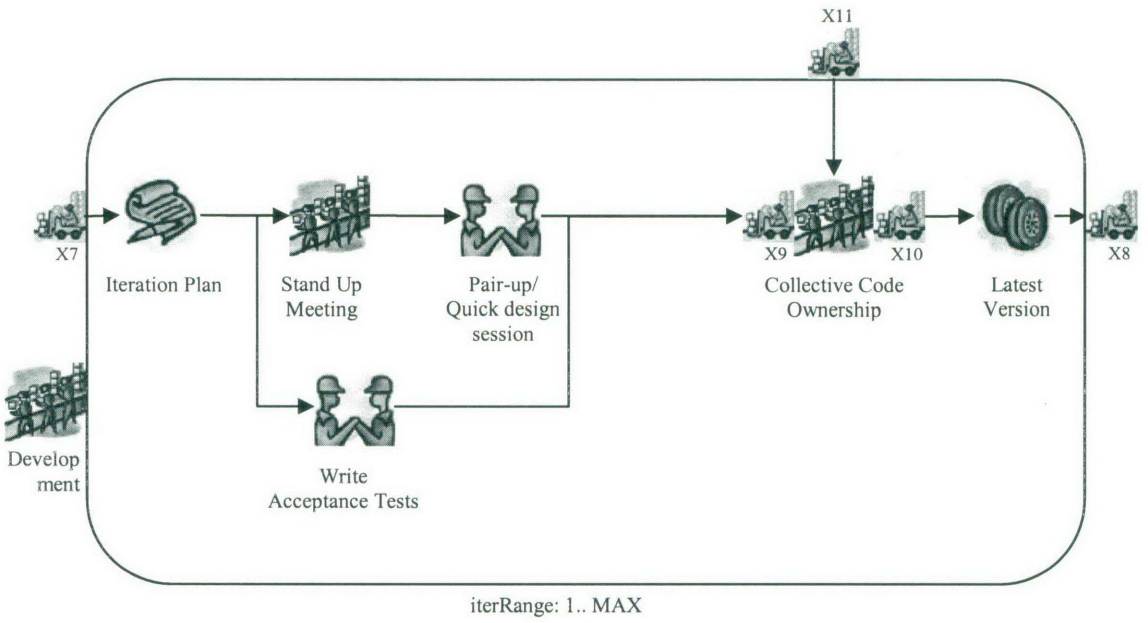
The bays will be used in subsequent decomposition. In each *iteration* of the *Iteration* planning phase, a plan is prepared for implementation. The *Iteration planning* is divided into a number of simpler tasks. This aspect of XP is shown in Figure 3-7.

There are certain rules relevant to each *Iteration*. This is recorded in the Rulebook of *Iteration planning* production point. A rule is that the union of all the identified tasks constitutes the plan. The rule specified in the *development* phase, shown in Figure 3-8, is that programming is done in pairs.



Note: *iterRange* is the range of possible iterations referred as an index-integer.

Figure 3-7 Iteration planning decomposition of XP process



Note: *iterRange* is the range of possible iterations referred as an index-integer.

Figure 3-8 Development decomposition of XP process

Figure 3-9 contains a few notations that have not been used so far. One is the dotted arrow from *Test Plan* to *Create a Unit Test* activity. This indicates that *Create a Unit Test* activity cannot start before the *Test Plan* is ready (an important requirement in XP) even though, the input to the *Create a Unit Test* activity is the Task document and

not the *Test Plan*. Without the dotted arrow, a unit test can be created as soon as the *Task Cards* is ready, and concurrently with test plan preparation; however, that will be a violation of XP process.

Another notation is the diamond symbol attached to *Unit Testing* with a condition *ok*. If *Unit Testing* is ok, *Continuous Integration* will be activated and the next unit test will be developed, if not, it sets a trigger (the switch symbol) to reactivate the *Pair Programming* activity. Likewise, if *Continuous Integration* is not ok, then *Pair Programming* may be reactivated.

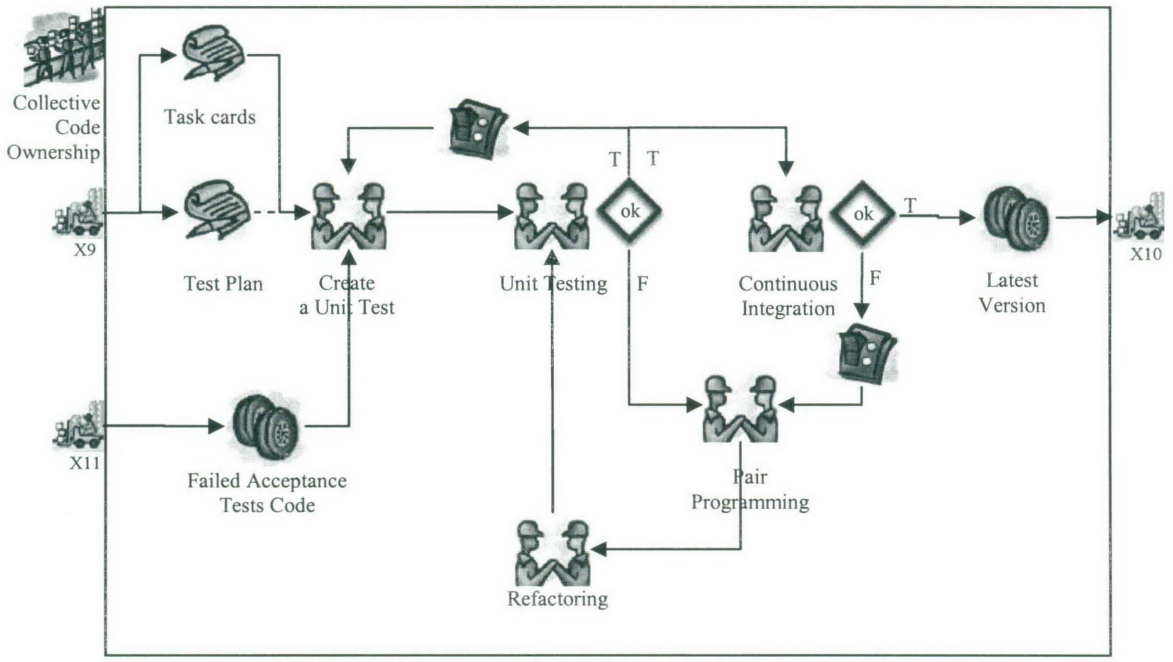


Figure 3-9 Collective Code Ownership decomposition of XP process

Also, it needs to be noted that there is no distinctive design phase in XP; this is because XP does not make any significant distinctions between design and coding; however, if needed, one can decompose *Pair Programming* to show the to and from movement between design and coding until the activity is complete.

There are other aspects of XP that can be represented using *Order*. One feature of XP is that more stories can be added any time to cater for *requirements creep* during development. This can be indicated in Figure 3-5 by connecting an external trigger to the *Release Planning* production point. The trigger causes *Release Planning* phase to generate more *stories* which then automatically restarts the *Iteration* production point.

XP also requires that every *story* must be business-oriented, testable and estimable (a criteria for the Rulebook of *Analysis* production point), and that a story is

typically the amount of *use cases* that fits in an index card (a statement for the suggestion box of *Release Planning*).

Another feature of XP is the involvement of customers in choosing the stories to be implemented in each iteration, as well as in writing acceptance tests. This can be written into the suggestion box of the *Iteration Planning* production point (Figure 3-7). Alternatively, one can decompose *Release Planning* to show details of the "planning game" where estimates of cost of stories and development speed of teams are given to customers as input to selecting the next subset of stories, which in turn are converted to smaller grained tasks by programmers.

3.5 Summary

Order provides a simple graphical notation for process specification and analysis. Its process decomposition facilities allow specification of processes in a top down manner, which makes it easy to think about macro-level phases first, and later the micro-level details. From a comprehension point of view, if senior managers are interested only in process steps at the highest level, they don't need to study detailed decompositions. Similarly, an engineer working on the implementation task probably needs to know the details of the task implementation process, and not necessarily details of, for instance, Problem Analysis.

Unlike other graphical process languages such as Little-JIL where a process decomposition is shown as a tree of substeps, *Order*'s decomposition into separate 'diagrams' do not clutter the precious screen real-estate. On the other hand, *Order* provides a flattening feature to see all process activities together on one screen when required.

Another benefit of decomposition is that the internal details of a production point can be changed independently as long as it satisfies the rulebook of the production point and constraints of bays attached to it. This gives the flexibility for adapting and modifying process activities without producing unexpected consequences in other activities. In other words, the rulebook and the attached bays form the external interface of a production point for applying principles similar to Meyer's (1997) design-by-contract.

As the number of stages and connections increase, visual specification can become cumbersome because of limitations of screen real-estate. As shown above, *Order* handles this problem by allowing modularization of process specification. A process may be specified as an interconnection of stages at high-level, and then each stage may be decomposed into separate diagrams showing its details. Apart from keeping each diagram small and manageable, this approach has other benefits such as reuse of process modules in different process specifications and separation of low-level details from high-level abstractions. However, a difficulty with modularization is that defects in the process design may be spread over different diagrams and may not be visible when they are looked at one at a time.

Chapter 4

Software Process Analysis Method

4.1 Introduction

Modeling of software processes was originally developed in the 1960s. However, at that time, it did not receive sufficient attention from software developers. The importance of software process models has become widely recognized only during the recent decades. Appropriate selection of modern software processes has the potential to tackle complex software development activities as well as considerably smoothen and maintain the development schedule. According to case studies, software process selection can ultimately lead the project to either success or failure (Ambler 2003).

It has been about 40 years since the most widely known *Waterfall* lifecycle is introduced to the software engineering community. This "*ideal*" lifecycle has created large numbers of scholarly feedback and arguments. As a result, many software development lifecycles and methodologies including variations in the waterfalls methods are introduced in response to outstanding criticisms, as well as various emerging software development needs. The significance and the diversity of software development lifecycles and methodologies have created a situation where a thorough study and analysis of various processes is needed to provide researchers and developers a better understanding and realization of process characteristics and their differences. To simplify the terms, we use 'software process' to represent 'software development process', 'software development lifecycle', 'software development methodology' or 'software development method'.

Why software process diversity should be realized? There may be a situation that a project manager who is looking for a process with a specific characteristic may want to know which software process he or she should select for his/her software project. In order to accomplish that, he must be aware of process diversity and be able to determine which software process is most suitable for the project. Another instance is when a

process manager knows the kind of software process she would like to use, but would like to know what other choices are available, in order to see whether there is a better candidate software process with similar characteristics. In such a case, an appropriate software process analysis method could be able to provide the project manager with answers to such queries/requirements thereby potentially reducing the time required for appropriate process selection. The knowledge gained during the analysis can also be useful in tailoring a software process to the project's needs.

Despite the importance of the issues discussed above, the majority of research contribution is dedicated to software process modeling and support, software metrics and software development methodologies. Unfortunately, software process analysis even though known to be the area that lays an important foundation for others, has received relatively less attention.

A number of software process analysis and comparison methods have been recently published. Some methods offer rigorous comparison of software process and/or models while other attempts are informal. A background on comparative analysis of software processes and related work is previously depicted in Chapter 2.

This chapter describes a method for the study and analysis of software processes. In this research, the main purpose is to help developers and researchers to realize the commonality and diversity that exist among software processes. Therefore, the context of the study is limited to the general models of software processes. Specific issues such as tailored processes for particular software projects are outside the scope of this study. To illustrate this idea, software process P tailored to project A is considered as a different process to P when tailored to project B.

The software process analysis method along with examples of process analysis is presented in subsequent sections. Section 4.3 defines components used in the software process analysis method. Then, section 4.4 demonstrates process elaboration, normalization, abstraction, and an example. Finally, the method is concluded in section 4.5.

4.2 Research Problems

Various issues need to be addressed in selecting a proper method for analysis of software process models. Firstly, in the case of software process comparison, various process descriptions often use different terminology to denote the same thing. For

instance, user requirements are called "User Stories" in Extreme Programming while it is called "Features" in Feature-Driven Development. Indeed, although User Stories and Features actually refer to the same object, the differences in terminology might lead to confusion. Even worse, sometimes the same term may have different meanings when used in different processes.

Another major issue is the difficulty of comparing process descriptions of different levels of detail. Obviously, it is logically incorrect to compare a suburb with a state, using the same measurements. In the same way, comparing one process description at one level of detail with another at a different level of detail could give us misleading results.

Software process descriptions available in references are often expressed in textual English descriptions and/or original diagrams which make it difficult to compare and analyze, because formats of descriptions or diagrams are of different standards. Process modeling is used to transform software process descriptions into a standard form recognized by the analysis method. When software process descriptions are mapped into a standard form via process modeling, they are ready to proceed with analysis. Additionally, process modeling alone is beneficial for improving process understanding and communication. When combined with an appropriate process analysis method, it would provide accurate insights into software processes, and enable the software development community to gain deeper understanding of software processes and, therefore, be able to manage them better.

In order to address issues discussed above, a three step procedure is employed. The steps comprise of *elaboration*, *normalization* and *abstraction* which are described in detail later. These steps prepare process models for analysis, introduce the use of common terms, and establish the base comparison level. The usage of process models and common terms is the basic principles supporting these steps. The basic components that constitute the three step analysis method are explained in following section.

4.3 Related elements

4.3.1 Process Representation

Frequently, process descriptions and diagrams used for such descriptions in the literature create problems to readers. Firstly, a reference might provide diagrams to describe activities of a software process without sufficient explanation of how each diagram is organized or how to interpret it. This problem should not be overlooked as it could later become a major problem that affects process understanding and communication. Secondly, different references may use different types of diagrams causing misinterpretation. For example, we have found different interpretations of the classic Spiral Model across several references (Boehm 1988; Chocano 2004). The difference in these interpretations can ultimately lead to completely different development processes. Thus, there is a need for a standard representation of the software processes to facilitate their analysis.

Software process model based on Order notation

A software process p is an instance of a software process model. Each software process model may have more than one software process instances. A software process model is a pair of sets $M = (N, A)$ where N is a set of production points and A is a set of directions. An arrow, a, is a pair (n_x, n_y) where n_x is the starting production point and n_y the ending production point.

A production point, n, is a concept in Order notation (See Appendix A). It represents a basic unit of software process activity. Each production point is attached with a unique label which contains a process term, t. A label is a short description of a process activity.

$M = (N, A)$ where

$N = \{ (n, t) \mid n$ is a type of production point, t is a process term$\}$

A = $\{a \mid a$ is $(n_x, n_y)\}$

$a = (n_x, n_y)$ denotes a direction from a production point n_x to another production point n_y; $n_x \in N, n_y \in N$

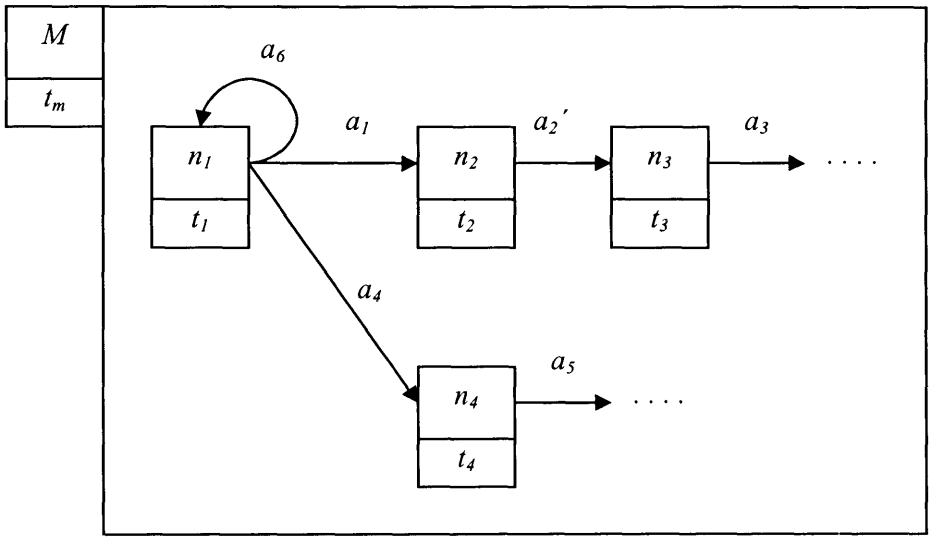


Figure 4-1 Process model structure

The process models, as appearing in Figure 4-1, are represented in *Order* notation. Details of the notation can be found in Appendix A.

4.3.2 Base References

There is little or no standard practices on how a process should be described. It is common to see the same process model being described by different authors using different notations. One reference may use UML diagrams to describe a process while others may use the authors' own notation. Some others may use flow charts and other custom representations.. Many of these custom representations are not defined accurately and therefore create ambiguity in process descriptions. This may lead to a different interpretation of a software process. This is a significant problem in studying process models. To address this problem in our analysis method, we prioritize different references to a process model and give preference to a reference written by the original process author or experienced developers and select one or in some cases more

references as the base reference for process analysis. In this chapter and the next, for each process model we study we indicate the base references used.

4.3.3 Process Dictionary

A software process contains information regarding activities, objects, conditions, time and other related information. This information is often presented as a model which consists of simple diagrams and narrative text structured in a uniform way. Several software process modeling approaches have been proposed. Some of them vary existing modeling approaches to address various aspects of software processes (Bandinelli et al. 1994; Bondavall, Majzik & Mura 1999; Deiters & Gruhn 1994; Finkelstein, Kramer & Hales 1992; Jager, Schleicher & Westfechtel 1998; Junkermann et al. 1994) while others design an entirely new set of notation (Baldi et al. 1994; Cass et al. 2000). Regardless of different approaches and representations, all process models are developed to help improving communication among people and providing a framework for project management, planning, measurement, automation, etc. Although the main purpose of a process model is to promote understanding among people, it can be challengingly used as a tool in process comparison. For instance, people participating in software development may communicate with each other in a number of ways and they may model their process or set up a process metaphor differently.

As briefly mentioned earlier, several software processes use specific buzzwords to define activities, objects and other artifacts. For example, Extreme Programming uses 'Spike' to refer to a simple program developed to explore potential solutions to tough design questions. This term may have a different meaning in another software process. Other terms may have a specific meaning to one process and have no meaning anywhere else. Obviously, this situation can cause confusion and is considered an obstacle to process understanding. Another interesting example is, while a term 'user story' or, in short, 'story' refers to a functionality of software in the Extreme Programming model, it may refer to a level of a building in a construction context. This term can have completely different meanings depending on the environment. To solve this problem, we use a process dictionary to collect, define and link typical process terms.

The process dictionary contains collections of terms used in software process models and their connections to other terms. The purpose of the dictionary is not only to introduce a set of terms that can be universally used across several process descriptions but also to minimize the repetition of terms. With an association from a process dictionary, processes can be efficiently described using common terms therefore a balanced and systematic comparative analysis can be made. Table 4-1 illustrates a portion of process dictionary.

Id	Type	Section	Description	Synonym	Related
Design description	Document	Designing	A document that describes the design of a system or component. Typical contents include system or component architecture, control logic, data structures, input/ output formats, interface descriptions, and algorithms.	design document; design specification	product specification
System requirements review	Process	Specifying	A review conducted to evaluate the completeness and adequacy of the requirements defined for a system; to evaluate the system engineering process that produced those requirements; to assess the results of system engineering studies; and to evaluate system engineering plans.	software specification review	software requirements review

Table 4-1 Entries in the process dictionary

The process dictionary, *D*, contains the definition of process terms and their relation to other terms. Initially, the dictionary contains terms taken from IEEE Glossary of Software Engineering Terminology (IEEE 1990) and IEEE/EIA 12207.0-1996 (Standard for Information Technology-Software life cycle processes) (IEEE/EIA 1998). New terms are added to the dictionary by the process analyst as new process definitions are considered. The definitions of new terms are taken from the base references.

The dictionary is a collection of entries, *e*. Each entry composes of a unique term referred as common term t_e and a set of meanings of the term $Meaning(t_e)$. Each term is related to the other in some degree of similarity. In other words, a term is associated with a set of synonyms S_e and a set of related terms L_e. Note that in most cases, a term will have only one meaning. However, as described earlier, because process designers often use their own terminology, some terms will have different meaning in different models, hence the need for a set to represent meaning.

Determining similarity and other relation among terms is a complicate matter. It can be performed manually on a small set of data and automatically on a large set of data. Although similarity measure can be automated, human judgment is still required to verify the end result.

$e = (t_e, Meaning(t_e), S_e, L_e)$, where

$\forall e_1, e_2 \in D : t_{e_1} \neq t_{e_2}$

$T = \{t_e\}$
$S_e \subseteq T$
$L_e \subseteq T$

Semantic relation and measurement

Synonyms and related terms to term t_e are determined by the degree of similarity between *Meaning(t_e)* to *Meaning(S_e)* or *Meaning(L_e)*. As *Meaning* is a set, one meaning from each set is considered at a time. The degree of similarity is called *closeness*.

Theoretically, for two software process terms, t_1, $t_2 \in T$, *closeness* C, or C(*Meaning(t_1), Meaning(t_2)*), indicates the closeness in meaning between two terms. C(*Meaning(t_1), Meaning(t_2)*) is a value between 0 and 1.

C	Meaning
0	When there is no relation in meaning between selected meanings of t_1 and t_2.
1	When selected meanings t_1 and t_2 are identical.
0<C<1	When selected meanings are related but not identical.

Table 4-2 Meaning of Closeness

We use the term 'closeness' in software process context to refer to quantification of term-similarity. In order to quantify the closeness of software process terms, we adapted an open source software called Reqsimile. Reqsimile is a Java application based on research led by Johan Natt och Dag (Dag 2005). The similarity measure used by this program is based on Cosine similarity. Although originally aimed to use with software requirement sets, Reqsimile can be adapted to use with software process terminology comparison. In this research, Reqsimile's database is altered in order to reflect the content of the Process Dictionary described earlier. Upon selection of a term in a process model, ReqSimile can efficiently determine the closeness to all terms in the dictionary.

To prepare for the analysis, process specific terms are considered and added to the dictionary if required.

- Each process term is checked for its existence in the dictionary. If it exists and its meaning is identical to the meaning defined in the dictionary, no update is required.
- If the term exists in the dictionary but there is no match in meanings, a new meaning is added to an existing term.
- If the term is not in the dictionary and cannot be substituted by any terms in the dictionary, it will be added to the dictionary as a new term.

Then, its connection to other terms in the dictionary is determined.

The algorithm for developing the dictionary is as follows:

for each process model, *p*
  for each term, t_p in p {
    let the meaning of t_p in *p* be *m*
      if there exists an entry *e in D* such that
        if t_p equals t_e
          if there is an element m_e in *Meaning*(t_e) such that *C*(*m*, m_e) equals 1
            break; // the entry is already in dictionary
          else if for all m_e in *Meaning*(t_e), *C*(*m*, m_e) equals 0 { // new meaning
            $Meaning(t_e) = Meaning(t_e) \cup \{m\}$
            break; }// go to next term in p
          else
            break; // not possible
        else { // t_p is not equals to t_e
          if there is an element m_e in *Meaning*(t_e) such that *C*(*m*, m_e) equals 1 {
            $S_e = S_e \cup \{t_p\}$ // add t_p as a synonym for entry e
            break; } // go to next term in p
          if there is an element m_e in *Meaning*(t_e) such that $0 < C(m, m_e) < 1$ {
            $L_e = L_e \cup \{t_p\}$ // t_p is a related term to t_e
            break; }

$D = D \cup \{t_p, Meaning(t_p), S_e, L\}_e$ } // add t_p as a new term to the dictionary

continue; // to explore relations with other terms }

Note that the conditions in above algorithm still require justification by the process designer, particularly in adding new terms to the dictionary.

Root terms

To establish the baseline for comparison of process terminology, a concept called *root term* is introduced which sets the foundation for process terms in the dictionary. Root terms represent main activities of a software development process. Every other term is either directly or indirectly related to at least one of the root terms, either by being close in meaning or by being a part of it. Thirteen root terms are taken from the development process defined in IEEE/EIA12207 international standard (IEEE/EIA 1998). (See Table 4-3)

The IEEE standard provides a common framework for software life-cycle processes from conceptualization of ideas through retirement. Activities that may be performed during the life cycle of software are grouped into primary processes, supporting processes, and organizational processes. Each process is further divided into activities and tasks, respectively. The development process (IEEE/EIA 1998) defines the activities of the organization that designs and develops the software product. Table 4-3 lists 13 root terms defined in IEEE/EIA12207 international standard. An additional root term, *supporting processes*, is added to represent other activities such as supporting processes or organizational processes since in many software process specifications there are "non-core" activities defined to support the main process activities.

Frequently, software process definitions do not make a distinction between system requirement and software requirement, or system architectural design and software architectural design, etc. Hence, it is more practical to classify these root terms into six groups (see Table 4-3) for analysis purpose.

Group	Root term
Planning	1. Process Implementation
Specifying	2. System requirements analysis 3. Software requirements analysis
Designing	4. System architectural design 5. Software architectural design 6. Software detailed design
Coding and testing	7. Software coding and testing 8. Software integration 9. Software qualification testing 10. System integration 11. System qualification testing
Delivering	12. Software installation 13. Software acceptance support
Supporting processes	14. Supporting processes

Table 4-3 Six groups of software process activities

Let R be the set of all root terms.

$R \subset T, r \in R$

R = *{Process implementation, System requirements analysis, System architectural design, Software requirements analysis, Software architectural design, Software detailed design, Software coding, Software testing, Software integration, Software qualification testing, System integration, System qualification testing, Software installation, Software acceptance support, Supporting Processes}*

A root term is defined based on the following condition.

1. It is not close to any other roots. That is,

   $\forall r_1, r_2 \in R \mid C(r_1, r_2) = 0$

2. Every term is directly or indirectly connected to at least one root term.

   $\forall t \in T \mid \exists r \in R \mid 0 < C(t, r) \leq 1$

4.4 Software Process Model Analysis Method (SAM)

Software process model analysis method (SAM) is introduced to promote better understanding of software processes including similarities, differences and relations among them. Knowledge gained from the analysis can be used in process selection, process tailoring, process adoption, process re-design or other areas. Related works are previously discussed in Chapter 3.

SAM consists of three main steps: (1) Elaboration, (2) Normalization, and (3) Abstraction. All of these elements are essential for performing process model analysis in SAM. The following subsections discuss each element.

4.4.1 Elaboration

There are two common approaches to defining processes, i.e. top-down and bottom-up. In a top-down approach the highest level is first described and later refined in greater detail. In a bottom-up approach the individual elements are defined first and then linked together to form a bigger picture. Moreover, a combination of both approaches is often used.

In the Elaboration step, the process definition is mapped to include all details specified in the base references. Ideally, a base reference is the standard reference for the process definition. However, in reality, such standard definition for software processes is not available. Hence, our options are limited to choosing the most authoritative reference(s) available. For example, base references for Extreme Programming process are chosen from those written by Kent Beck, the author of Extreme Programming.

In this chapter, we use the classical Waterfall model to illustrate how a software process model is prepared in the elaboration step. The base reference used for the waterfall process model is Royce (1970). In the reference, the process is initially described in big picture terms. Figure 4-2 illustrates the top-level view of the sequential Waterfall model in Order notation. The terms used to denote activities are directly retrieved from the original publication of Royce without any modifications (Royce 1970). In fact, Royce explained in his publication that there is the possibility of going back to redo the previous phase by the use of feedback. However, many publications do

not address this side of the model and presents it as a sequential model of software development, and thus the name Waterfall.

To prepare for the analysis, the model is described in a top-down fashion. In each iteration, more details from the base reference(s) are added until no further elaboration is possible. The final elaboration is determined either when all details in the reference are captured and/or when each element in the elaborated model is atomic. An atomic element is the smallest element which cannot be elaborated further. Atomicity is decided based on the base reference. An element is determined as atomic when there is no more description or explanation provided by the base reference. Figure 4-3, Figure 4-4, Figure 4-5 and Figure 4-6 show the elaboration of preliminary program design, program design, testing and operation activities. The final outcome of elaboration, which captures the whole Waterfall model is given in Figure 4-8.

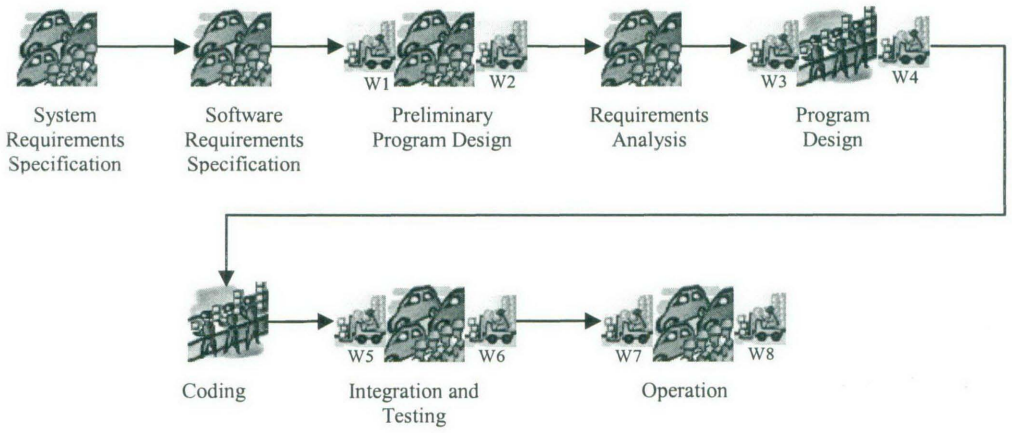


Figure 4-2 A top-level view of the Waterfall model (Royce 1970).

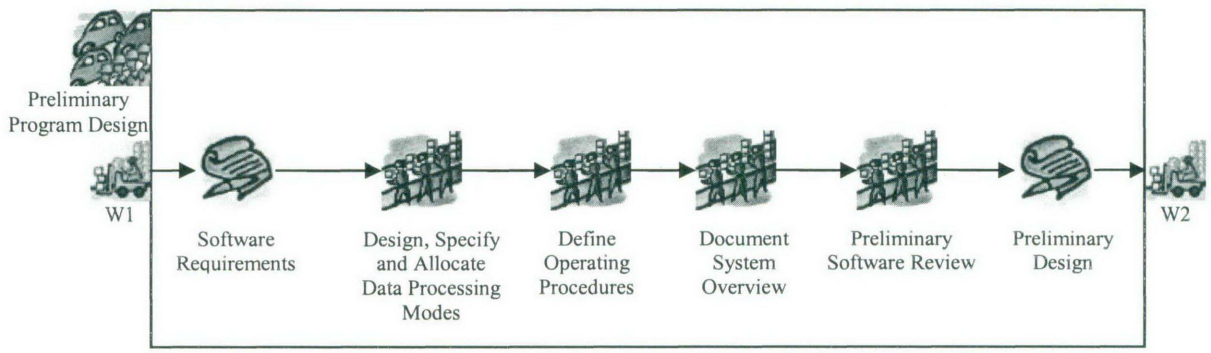


Figure 4-3 Waterfall's Elaboration: Preliminary Program Design

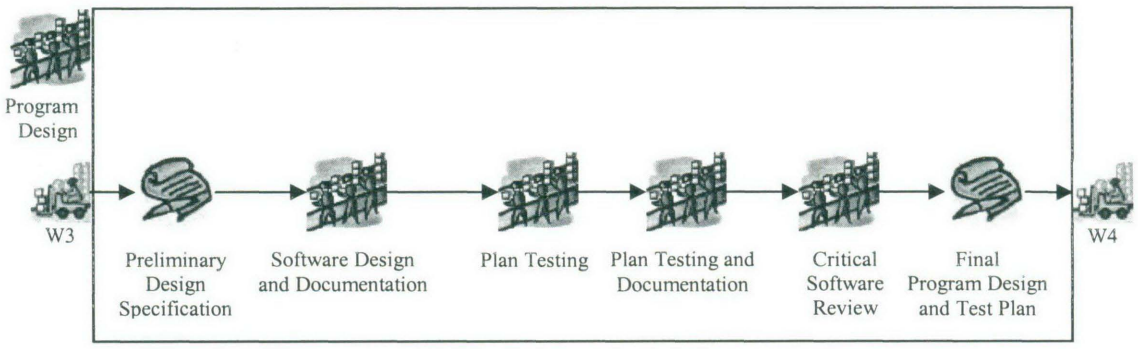


Figure 4-4 Waterfall's Elaboration: Program Design

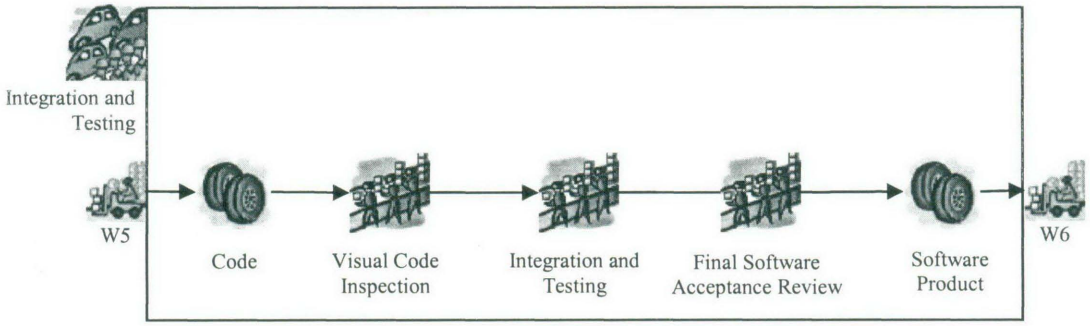


Figure 4-5 Waterfall's Elaboration: Integration and Testing

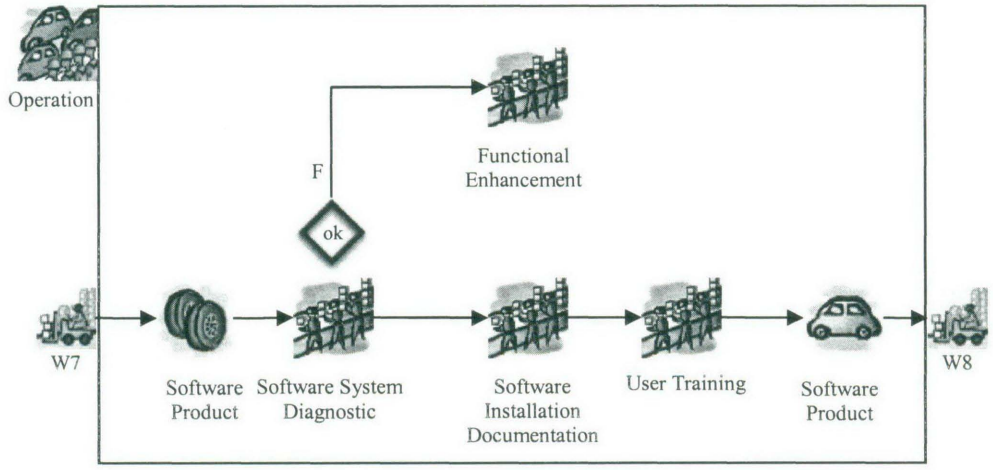


Figure 4-6 Waterfall's Elaboration: Operation

4.4.2 Normalization

Generally, a process term denotes a process activity in a natural language phrase. Each term has a meaning, and in some cases, more than one meaning. Similar to a common dictionary, process terms may have relationships to other terms in the dictionary. In

other words, a process term may relate to another term by being a synonym or by having a similar meaning. In contrast, a process activity may be referred by different terms. For instance, two terms 'software requirements specification' and 'write user stories' are actually related in software process literature. It may not be logically possible to notice the similarity at first glance. In order to efficiently judge whether these two terms are related, their descriptions need to be systematically compared. However, comparing process descriptions of different processes poses as much challenge as comparing two sentences written in different languages. The comparison would have been straightforward if there were a universally accepted standard for process descriptions (similar to the Backus Naur form for specifying programming language syntax).

Similarity between software processes can be efficiently analyzed when processes are represented by the same rules. This requires all processes to be modeled and process elements to be defined using the standard terminology. The need for a standard terminology has become important with the increasing rate of new software development processes and variation of existing processes. Such diversity has led to creation of new terms and adoption of new meanings to existing terms, which in turn causes difficulties in process analysis and comparison. Overcoming this challenge requires us to develop a mapping of process terms to standard terms.

The term normalization is used in this thesis to indicate the process of mapping particular process terms to the standard terms in order to minimize their great variety of meanings which is caused by software process diversity. To assist with normalization, the Process Dictionary and similarity measures are employed.

As part of normalization of a process model, non-standard terms that are specific to a model, e.g. user stories in XP or sprint in Scrum, are replaced by their corresponding standard terms from the dictionary such as user requirements or iteration. When all specific process terms in the process model are replaced with standard dictionary terms, normalization is complete and the process is ready for further analysis.

Initially, the Process Dictionary contains the collection of standard terms from the IEEE Standard Glossary of Software Engineering Terminology (1990). The glossary includes definitions of terms and relationships to other terms in the field of Software Engineering as well as software development processes. Terms are excluded if they are narrowly restricted to one group; multi-word terms whose meaning could be deduced

from the definitions of the component words; or terms whose meaning could be directly inferred from English dictionary. When a process term is considered, it may be added as a new entry to the Process Dictionary or as a new related term to an existing term depending on the result of the similarity measure. Indeed, number of entries in the Process Dictionary grows as new terms are found.

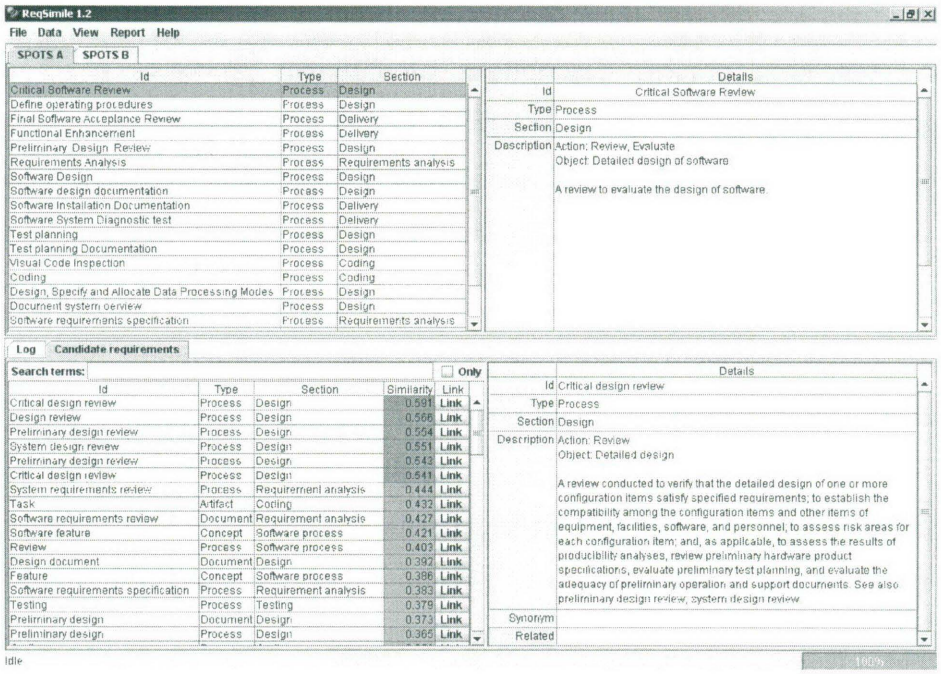


Figure 4-7 Reqsimile queried for similar terms for each process specific term from the Process Dictionary

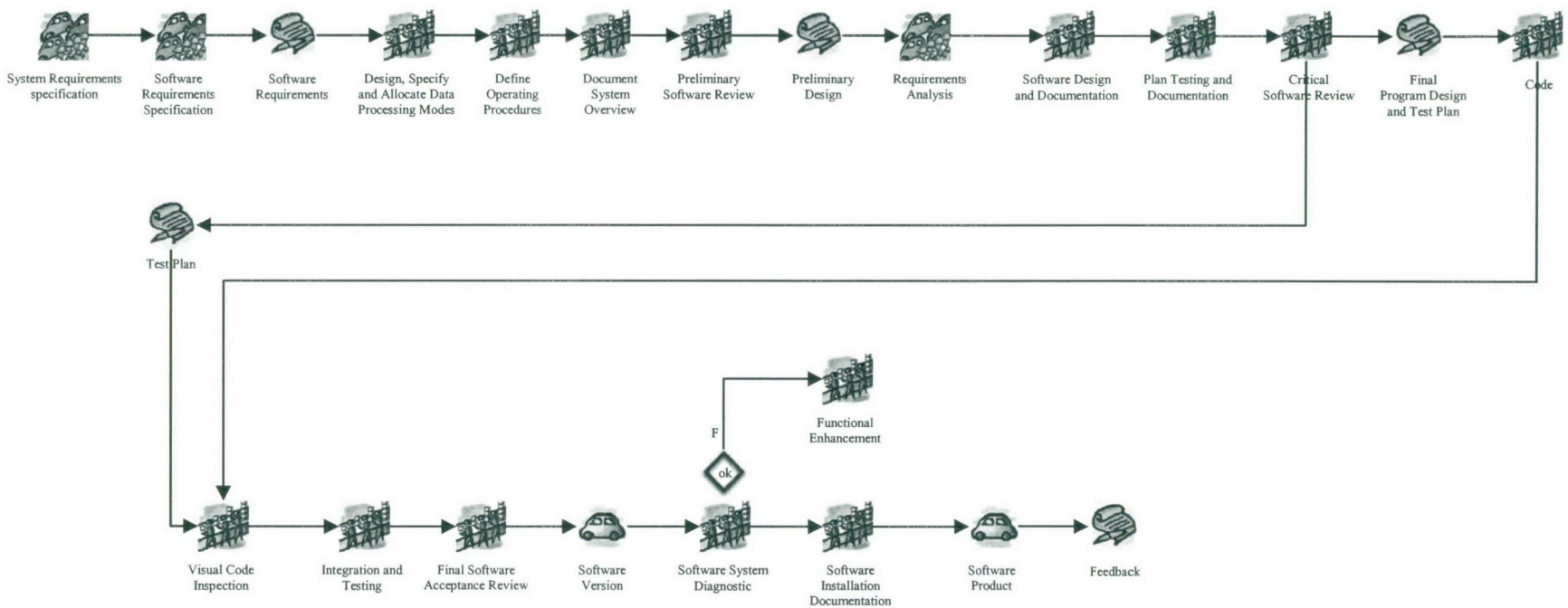


Figure 4-8 Elaboration of the Waterfall model

When a process term is chosen for normalization, it will be replaced with a corresponding standard term from the Process Dictionary. Results from similarity measure determine which standard term has the closest meaning. **Figure 4-7** shows process terms and similar terms determined by Reqsimile. The upper half windows contain process specific terms and the bottom half windows contain standard terms from the Process Dictionary. When a process specific term is selected in the top-half window, terms similar to it are displayed in the bottom-half window, sorted by their similarity value. The similarity value ranges from 0 (no relation) to 1 (identical). The windows on the right side display details of the selected term.

The term with highest similarity value is automatically chosen as the normalized term. Table 4-4 shows normalized terms chosen for the Waterfall process model based on similarity values obtained from Reqsimile which calculates similarity based on selected data fields. In this case, type and description of terms are the most appropriate candidates for terminology comparison as different process terms may be used to mean the same thing. Likewise, the same terms may be used differently when presented in different processes. Manual intervention by the process designer may be required when the term highlighted by Reqsimile as the closest is not appropriate. This can happen, for instance, if a process developer has used a term differently from its commonly understood meaning. Table 4-5 show the final normalized terms for the waterfall model; the manual corrections are notified in bold.

Waterfall process terms	Normalized terms
System requirements specification	Requirements specification
Software requirements specification	Software requirements specification
Document system overview	Installation manual
Design, Specify and Allocate Data Processing Modes	Design
Define operating procedures	Design
Preliminary Design Review	Critical design review
Requirements Analysis	Requirement specification
Software design and documentation	Design
Test planning	Supporting processes
Test planning Documentation	Document
Critical Software Review	Critical design review
Coding	Coding
Visual Code Inspection	Inspection
Integration	Integration testing
Final Software Acceptance Review	Review
Software Installation and Documentation	Product support
User Training	Product support
Software System Diagnostic test	Testing
Functional Enhancement	Document

Table 4-4 Normalized terms based on Reqsimile

Waterfall process terms	Normalized terms
System requirements specification	Requirements specification
Software requirements specification	Software requirements specification
Document system overview	**Document**
Design, Specify and Allocate Data Processing Modes	Design
Define operating procedures	Design
Preliminary Design Review	**Preliminary design review**
Requirements Analysis	**Requirement analysis**
Software design and documentation	Design
Test planning	**Develop plan**
Test planning Documentation	Document
Critical Software Review	Critical design review
Coding	Coding
Visual Code Inspection	Inspection
Integration	**Integration**
Final Software Acceptance Review	Review
Software Installation and Documentation	**Installation and checkout phase**
User Training	**Software acceptance support**
Software System Diagnostic test	Testing
Functional Enhancement	**Operation and maintenance phase**

Table 4-5 Normalized terms based on process designer's judgment

4.4.3 Abstraction

The completion of the normalization step provides us with process models that use the same terminology which is a significant step towards making comparisons easier. However, there can be wide variations in the size of normalized diagrams from one process to the next, which needs to be addressed. Therefore, the next step is to combine all similar activities to provide an abstract view of each process model. Through abstraction, we aim to establish the baseline for comparison by relating normalized elements to their root elements.

Each node in the normalized model is first replaced by its corresponding root term (see Figure 4-9). Then, a detailed abstract model is constructed as in Figure 4-10. Multiple occurrences of the same root term are compressed to one node in the abstract model (see Figure 4-11). The thickness of the node indicates the number of occurrences of the corresponding root term. Similarly, the thickness of a link indicates the number of links in the normalized model between nodes that are translated to the same pair of root terms. For example, in Figure 4-11, the link from Designing to Coding & Testing is twice as thick as the link from Coding & Testing to Delivering because there is only one connective activity from Coding & Testing to Delivering whereas there are two activities from Designing to Coding & Testing.

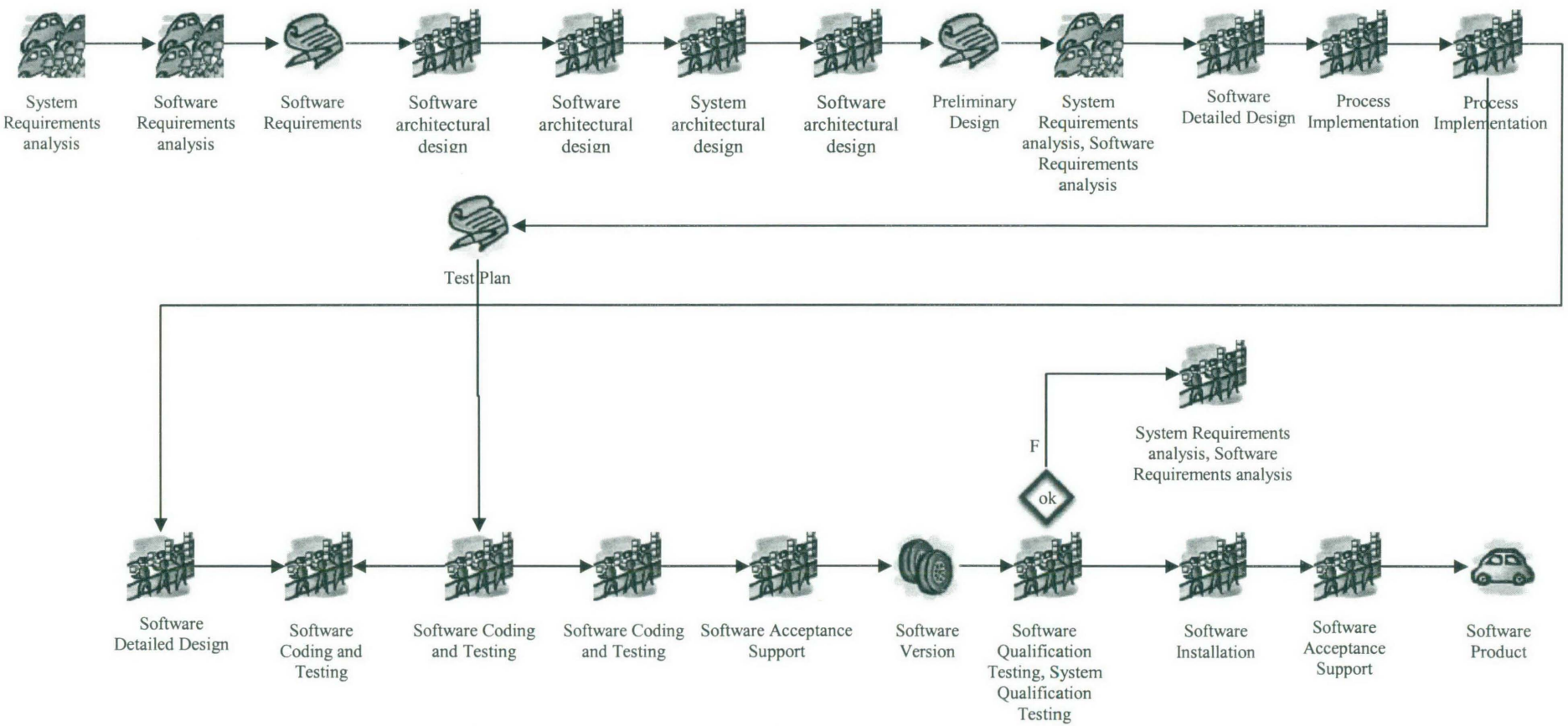
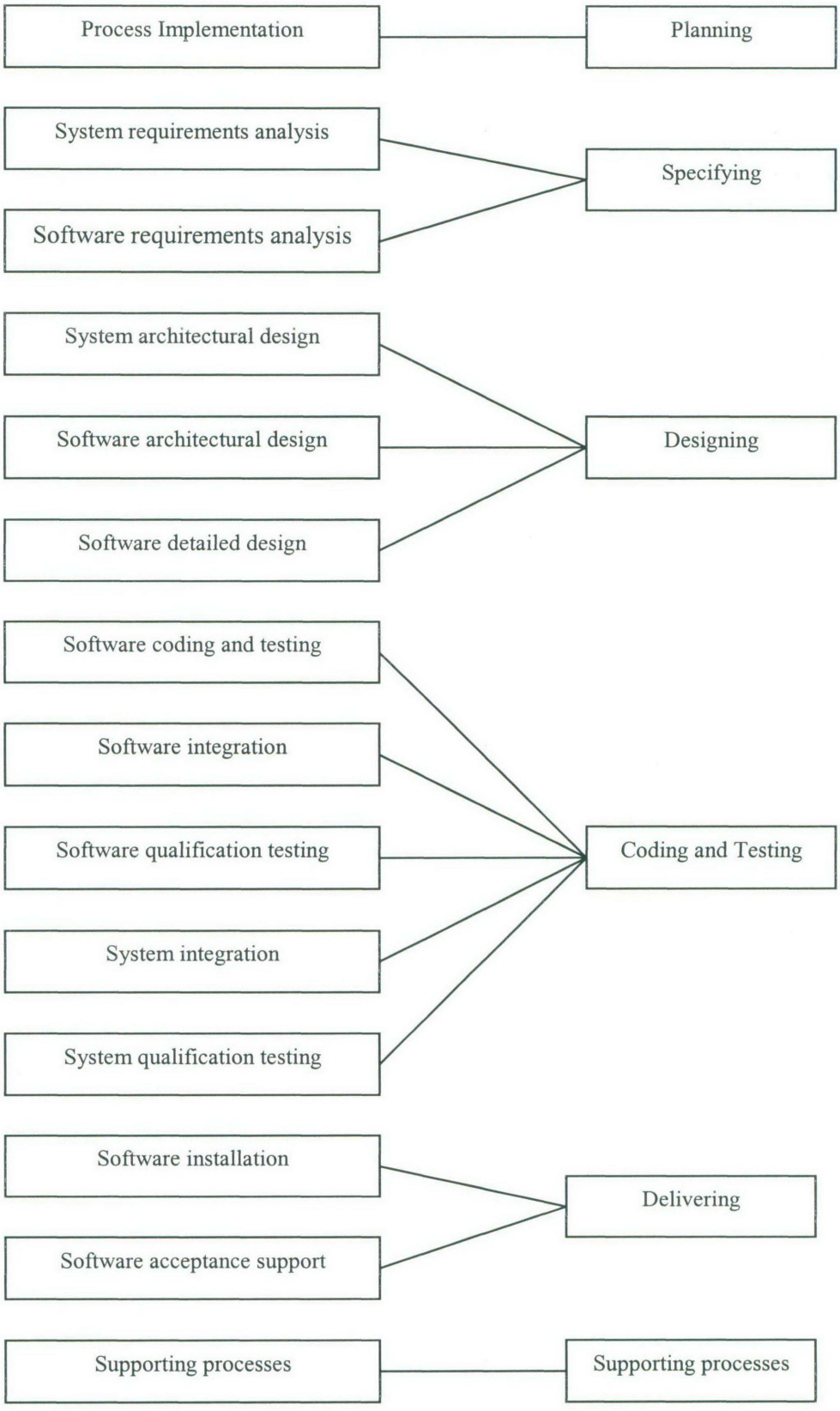


Figure 4-9 The Waterfall model with all process terms replaced by their root terms

Table 4-6 shows mapping from normalized terms to root terms. **Figure 4-10** shows the detailed abstraction. Typically, a normalized term is mapped to its corresponding root term. Sometimes, a process term may relate to more than one root term. For example, integration may represent software integration and/or system integration. In such cases, we list all related root terms (see **Table 4-6**) and they will equally share the strength of connection to interconnecting nodes.

Normalized terms	Root terms
System requirements specification	System Requirements analysis
Software requirements specification	Software Requirements analysis
Document system overview	System architectural design
Design, Specify and Allocate Data Processing Modes	Software architectural design
Define operating procedures	Software architectural design
Preliminary Design Review	Software architectural design
Requirements Analysis	System requirements analysis, Software requirements analysis
Software design documentation	Software detailed design
Test planning	Process implementation
Test planning Documentation	Process implementation
Critical Software Review	Software detailed design
Coding	Software coding
Visual Code Inspection	Software coding
Integration and testing	Software Integration, System Integration
Final Software Acceptance Review	Software Acceptance Support
Software Installation Documentation	Software installation
User training	Software Acceptance Support
Software System Diagnostic test	Software qualification testing, System qualification testing
Functional Enhancement	System requirements analysis, Software requirements analysis

Table 4-6 Abstraction of Waterfall process specific terms



Note: The mapping from root terms to abstract terms are occurred from left to right only

Figure 4-10 Abstraction of the Waterfall process model in detail

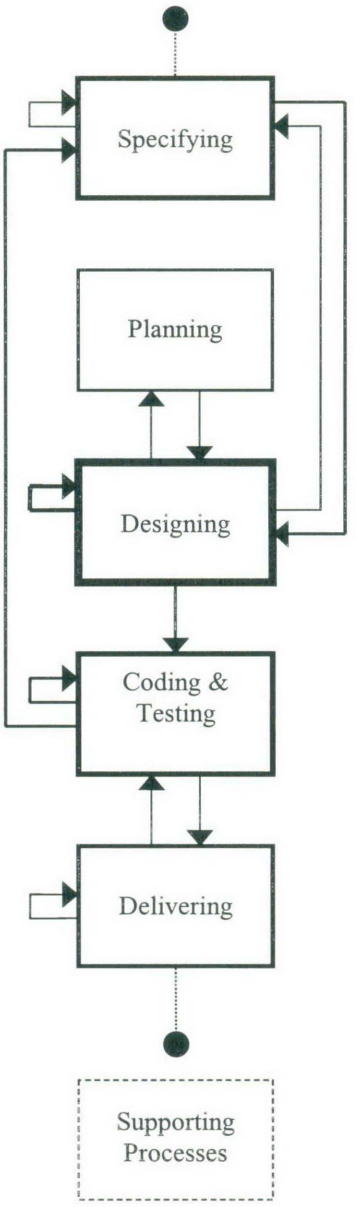


Figure 4-11 Abstraction of the Waterfall model

Table 4-6 gives us some insight information about process description. Some process descriptions do not provide sufficient information to allow accurate classification of process elements. Moreover, some process descriptions do not differentiate between software and system operation, leaving them to be tailored in specific projects. In fact, processes are frequently described in that fashion. Therefore, in order to be able to make useful measurements in the following chapters, we reduce the abstraction diagram to a set of six nodes: *Planning, Specifying, Designing, Coding & Testing, Delivering and Supporting Processes*. These are the groupings of the root-terms identified in Table 4-3. **Figure 4-11** shows the final abstraction diagram reduced

from **Figure 4-10** to simplify the abstraction based on groupings in Table 4-3. The filled circle indicates the starting and ending point of the model. The abstract model reveals that Waterfall process has a significant emphasis on the Designing activity, following by Specifying and Coding & Testing activities. In addition, the thickest link on the Designing node indicates that the most frequent interaction occurrence is actually between various activities that are related to Designing. Moreover, the absence of supporting processes in Waterfall is illustrated as a dashed node. Further investigation of **Figure 4-11** can reveal several other minor observations such as that there is no connection between Software Design and Delivery.

In this study, iteration in software process is illustrated in elaboration. Nevertheless, this is not addressed in abstraction. As the number of iterations is unknown, the minimum number of iteration is used in counting. Therefore, an iterative process is treated as a sequential process. However, for future study purposes, unlike sequential activities which are encapsulated in normal rectangles, these looping activities are encapsulated in rounded rectangles (See Appendix A).

4.5 Summary

Software process models can be easily compared when they are presented in a uniform way. In this chapter, we presented a method called SAM for software process terminology normalization and abstraction that helps process designers to create the baseline for comparison of various software process models. With the use of the Process Dictionary, process specific terms are normalized to standard terms. This eliminated the differences in terminology between software processes. Then a graphical abstract model of the software process is derived showing the strength of key activities and their interactions.

Most SAM steps can be automated and can be part of a process engineering environment (Inchaiwong & Sajeev 2004). Process modeling tools may be used to assist elaboration and abstraction steps. However, at this stage, each step is processed separately since an automated interface for linking the elaborated model and the similarity measurement has not yet been implemented. Additionally, due to the high complexity and lack of standard in the way processes are described, although these steps can be automated, human judgment is still needed in order to verify the

reliability of the results. In the next Chapter we use SAM to analyse and compare five prominent process models.