A Common Software Development Framework For Coordinating Usability Engineering and Software Engineering Activities

Sourabh A. Pawar

Thesis submitted to the Faculty of
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Applications

Approved:

James D. Arthur, Chair

Deborah S. Hix

Osman Balci

May 2004
Blacksburg, Virginia

Keywords: Coordination, Integration, Software Engineering, Usability Engineering, Requirements Generation, Framework, Development Process

A Common Software Development Framework For Coordinating Usability Engineering and Software Engineering Activities

Sourabh A. Pawar

ABSTRACT

Currently, the Usability Engineering (UE) and Software Engineering (SE) processes are practiced as being independent of each other. However, several dependencies and constraints exist between the interface specifications and the functional core, which make coordination between the UE and the SE teams crucial. Failure of coordination between the UE and SE teams leads to software that often lacks necessary functionality and impedes user performance. At the same time, the UE and SE processes cannot be integrated because of the differences in focus, techniques, and terminology. We therefore propose a development framework that incorporates SE and UE efforts to guide current software development.

The framework characterizes the information exchange that must exist between the UE and SE teams during software development to form the basis of the coordinated development framework. The UE Scenario-Based Design (SBD) process provides the basis for identifying UE activities. Similarly, the Requirements Generation Model (RGM), and Structured Analysis and Design are used to identify SE activities. We identify UE and SE activities that can influence each other, and identify the high-level exchange of information that must exist among these activities. We further examine these interactions to gain a more in-depth understanding as to the precise exchange of information that must exist among them.

The identification of interacting activities forms the basis of a coordinated development framework that incorporates and synchronizes the UE and SE processes. An examination of the Incremental and Spiral models as they relate to the SBD is provided, and outlines how our integration framework can be composed. Using the results of and insights gained from our research, we also suggest additional avenues for future work.

ACKNOWLEDGEMENTS

This research would not have been possible without the support of many people, all of whom I would like to thank here. I am forever indebted to my advisor, Dr. James D. Arthur for all the guidance, stimulus, and practical advice provided over the past two years. I am thankful to him for his support and motivation without which completion of the work presented in this thesis would not have been possible. I shall always remember Dr. Arthur for the efforts he has spent in strengthening my understanding about topics related to my research.

I am grateful to Dr. Deborah Hix and Dr. Osman Balci for serving on my thesis committee. Special thanks to Dr. Balci for being an extremely understanding GTA supervisor and giving me enough leeway to help me in managing my research and assistantship duties together. I would also take this opportunity to thank my friend Mr. Pardha Pyla, who is also a Ph.D. student from the Computer Science department at Virginia Tech for all the help and guidance he has provided me throughout my research.

I am also thankful to the Computer Science department – the faculty and staff. Being a graduate student at Virginia Tech has been an incredible experience. I shall always remember these two years I have spent here as one of the best phases of my life.

I am extremely grateful to my parents Anil and Anuja Pawar, who have always provided me the encouragement to acquire the education I wanted. Their love and belief have kept me going through the difficult times. I attribute the degree I am being conferred upon to the efforts of my father, whose able guidance and immense inspiration has positively influenced my life and shaped the person that I am today.

Finally yet importantly, I am thankful to all friends and family back home and in the United States, whose love, blessings and well wishes have shown me the success that I have achieved in the form of this master's degree.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

In this thesis, we present our research, which focuses on coordinating the usability engineering (UE) and software engineering (SE) processes. The rise of interactive systems with substantial functionality mandates the coordination between the usability and software engineering processes. Because each process has significantly different objectives, we have elected to present an approach that emphasizes coordination rather than integration. In this first chapter, we:

- Motivate the need for coordination between the UE and SE processes.
- Identify the problems in defining that coordination.
- Discuss the issues that need to be addressed, and
- Present a solution approach for defining the coordination between the UE and SE processes.

1.1 The Usability Engineering and Software Engineering Processes

The Usability Engineering and Software Engineering life cycle activities help develop “usable” and “functionally satisfactory” software systems. The factors that make a system more or less usable and functionally satisfactory are complex. A usable system should enhance human performance and be easy to learn. At the same time, a usable system should be easily adaptable and should provide a satisfying user experience. A functionally satisfactory system should possess the necessary functionality to satisfy the requirements of all users when deployed in the users’ environments. Additionally, the system should interface well with the other systems already in use.

The ultimate goal of UE is to create systems with a measurably high usability; the practical objective is to provide interaction design specifications to software engineers [Pyla 2004]. The ultimate goal of software engineering is to engineer software systems that possess the necessary functionality to support user requirements.

Figure 1.1 represents a typical software development scenario with parallel and independent UE and SE processes. In such a scenario, the design document containing

UE and SE specifications is given to the developers when the project reaches the implementation stage [Pyla 2004].

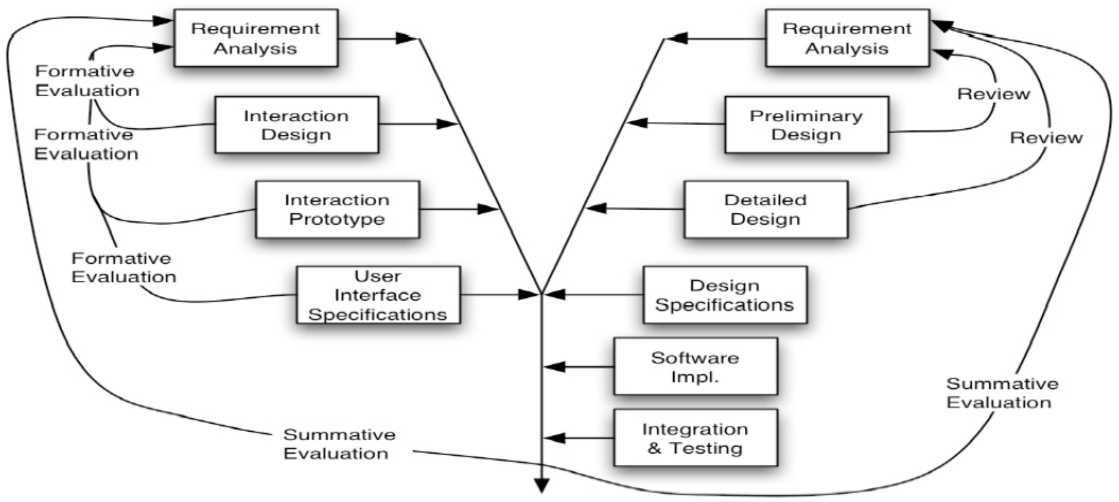

Figure 1.1 The Y-Model [from Pyla 2004]

Several dependencies and constraints exist between the interface specifications and the functional core, which make coordination between the UE and the SE teams crucial for the production of a satisfactory design document. Coordination between the UE and SE teams during the requirements generation and design stages is required to understand and integrate these dependencies and constraints. Failure of coordination between the UE and SE teams leads to software that often lacks necessary functionality. Additionally, the software may also impede user performance. Clearly, the use of usability methods and techniques synchronized with the software engineering process is necessary to make software usable and deployable in the user's environment. To date, however, that coordination is lacking.

Later in this introductory chapter, we provide additional motivation for a process that integrates the UE and SE processes. We discuss the issues that have to be addressed before we can define such a process, and then we present our approach towards a solution.

1.1.1 Usability Engineering

Software systems serve human needs and interests. Therefore, one should expect provisions for effective interaction between the user and the software system

[Constantine 2003]. Today, with an increase in the diversity among end users, the expectation for effective interaction has increased. Designers of interactive systems now design and evaluate systems with respect to usability – the quality of a system with respect to ease of learning, ease of use, and user satisfaction [Rosson 2002]. Rosson and Carroll define usability engineering as a field that "refers to concepts and techniques for planning, achieving, and verifying objectives for system usability." Usability therefore refers to how "usable" software is. A usable software is effective in (a) helping the users interpret the information provided to them by the user interface, (b) helping the users to create and execute an action plan for the tasks to be performed using the software (c) supporting all the functionality that is considered necessary by the different types of users, and (d) providing the users with a sense of satisfaction related to task completion. Usability Engineering is the process that incorporates a set of methods and techniques used by the usability engineers to design a "usable" system. The process helps the usability engineers in iteratively evaluating how usable the software is, and improving the design to make the software more usable.

Three different perspectives have contributed to the modern view of usability and usability engineering as depicted in figure 1.2. These perspectives, *human performance, learning and cognition,* and *collaborative activity,* are complementary to usability and increase its richness.

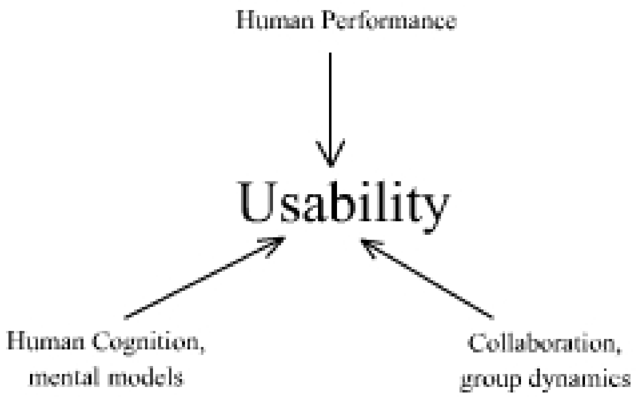

Figure 1.2 Perspectives contributing to Usability Engineering

Studies related to human learning and cognition define and reinforce the basic principles of usability engineering and the guidelines that the usability engineers follow.

Studies related to measurement of human performance are important because they define techniques to measure empirically the usability of software systems. Insights gained from the measurement of software usability play an important role in software development.

With an increase in the variety of users, a software system should most importantly provide a user interface that supports rapid learning and self-study for all its user classes. This requirement necessitates the use of user interface specification methods based on guidelines and theory, and the understanding of user behavior while using the system. User interface prototypes are developed throughout the user interface design process to make a precise test of open issues related to usability of the interface. Continuous redesign of the user interface is required until a sufficient user performance level is achieved. Usability engineering is therefore, an iterative process.

Initially, UE focused on designing an interactive user interface for the effective presentation of information. Now, the usability goals and objectives mandate an inclusion of additional UE activities mirroring those found in SE; e.g., requirements engineering and system prototyping.

1.1.2 Software Engineering

Software Engineering is the application of systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software. Software Engineering addresses the application of engineering methods to software development. With the application of sound software engineering principles, the software developed is reliable, maintainable, and efficient.

The software engineering process identifies the problems for which a software solution is to be generated. It also identifies the characteristics of the software solution that can solve these problems. Software Engineering further identifies the paths to construct a software solution, as well as defines strategies for error detection and downstream maintenance.

Software engineers have introduced a structure to the overall software development process through a better understanding of the various processes involved in software development. These processes have been expressed as software development models. The

Waterfall Model [Royce 1970], the Incremental Model [McDermid 1993], and the Spiral Model [Boehm 1988] are examples of software development models. Software development models decompose the software development process into five major phases: the requirements engineering phase, the software design phase, the implementation phase, the integration phase, and the maintenance phase.

The requirements engineering phase focuses on elicitation of requirements for the software system. These requirements are derived from the high-level requirements defined by the system engineers and from the identified needs of the users. The software design phase translates the software requirements elicited during requirements engineering into architecture for the software system.

Using the elicited requirements, the software engineers during the design phase create software architecture, components, component interfaces, and data elements. During the design of component interfaces, the internal, external, and user interfaces of the components are designed.

The implementation phase involves codification of the components designed during the software design phase. The implementation phase also involves testing the functionality of each component of the software system as it is developed.

The integration phase integrates the implemented components into a software product. It tests the software product for the functionality provided and verifies adherence to the elicited requirements. The maintenance phase focuses on maintaining the software product after delivery. In particular, maintenance activities include fault detection, performance enhancements, and adaptations to accommodate changes in the environment and user needs.

In chapter 2, we discuss the Software Engineering process and the software development models in detail.

1.1.3 Usability in Current Software Development Efforts

One should expect solid connections for collaboration and communication between the UE and SE development processes, given that these processes have the same high-level goals. However, there exists little (if any) coordination between the two processes; that is, they are independently applied during product development.

Usability engineers are frequently brought into the development process during or after the implementation stage to “fix” the usability of an already implemented system. Implementing usability at this stage introduces the risk of significant overhead to make the software usable. Because of budget and time constraints, software engineers often ignore changes proposed by usability engineers that require architectural modifications. Moreover, the few suggested changes that the software developers do implement are often only cosmetic in nature. Even when usability is an explicitly stated concern, usability engineers employ processes and techniques that are not well understood by software engineers. Consequently, these processes and techniques are difficult to translate into a form that readily admits to implementation [Ferré 2003]. Hence, the lack of coordination and effective communication between the usability and software engineers often leads to conflicts and miscommunications. These conflicts and miscommunications lead to software systems that not only lack usability, but also often lack required functionality.

1.1.4 A Shifting of Focus

Over the past few years, efforts to narrow the gap between the UE and SE processes have increased. The problem this engenders is multifaceted, involving differences related to historical evolution, training, professional orientation, and technical focus. Additionally, differences also stem from the use of domain-specific methods, tools, models, and techniques. Nevertheless, usability is an essential part of software quality. Software vendors increasingly perceive it as strategic for their software development businesses. An increasing number of software development organizations are pursuing the goal of integrating usability practices into their software engineering processes [Juristo 2001]. Projects like STATUS focus on introducing a forward engineering approach to usability engineering as part of the software development effort [Ferré 2002]. To bridge the gap between usability and software engineering practices, we must find ways to integrate or at least coordinate them.

1.2 Designing the Common Framework

Because of the differences in the UE and SE methods and techniques, integrating the UE methods and techniques into current SE practices and development processes (or vice-versa) is difficult. This section discusses an approach of designing the *common framework* based on the definition of an interface between the UE and SE processes.

1.2.1 A coordinated development framework

Although they perform similar activities, the usability and software engineers have different objectives. The usability engineers focus on specifying the user interface for the software, while the software engineers focus on the functional specification. This difference in focus leads to a difference in techniques and terminology used. The "integration" of the UE and SE processes therefore, becomes significantly difficult.

Even though the *integration* of the UE and SE processes is difficult, they can be *coordinated* under a development framework that incorporates them as separate processes. The definition of an interface between the UE and SE processes can help define this coordinated development framework. The interface defines the necessary coordination and synchronization points between the processes.

1.2.2 The interface between UE and SE processes

If the system to be developed is a large system, the systems engineering process provides a set of high-level requirements to guide the software development process. On the other hand, if the system to be developed is a small system, these high-level requirements are available from a concept document and the users. In both cases, the high-level requirements are available to both the UE and the SE processes as software requirements. The following diagram shows the UE and the SE processes acquiring software requirements from systems engineering. The diagram also shows the necessary, but often ill-defined or missing interface between the UE and SE processes.

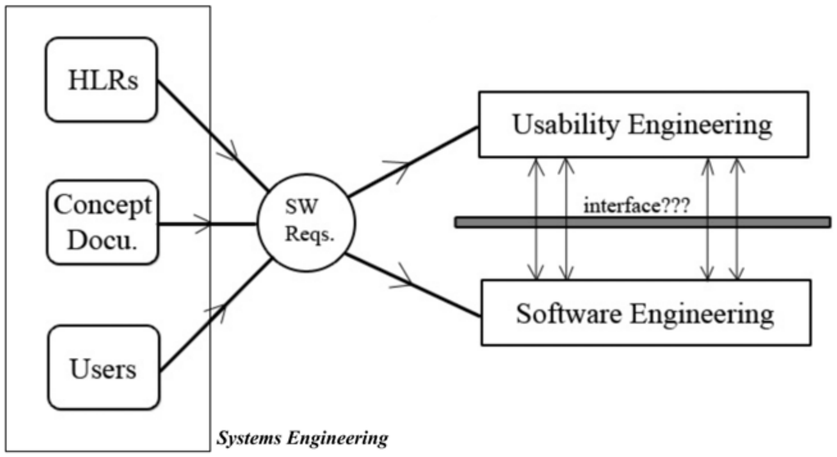

Figure 1.3 The interface between the UE and SE processes

The interface component codifies the need for several levels of interactions that must exist among activities defined by the SE and UE processes. Such interaction can be beneficial, as well as detrimental. An example of a beneficial interaction is when the software engineers develop stakeholder profiles and then share that information with the usability engineers. An example of a detrimental interaction is when a UE activity produces a user interface requirement during the design phase of the SE process. In such cases, the designers must re-visit their design and modify it to support the "late" requirement.

To explore the beneficial aspects of the interactions, we need to study how the candidate SE and UE activities can be coordinated and synchronized within a supporting framework. To mitigate the detrimental aspect, we need to identify the undesirable interactions and design mitigation strategies that tend to minimize such interactions. Understanding the interactions among the UE and SE activities, and investigating how they must be coordinated and synchronized leads to the necessary insights. These insights provide the basis for defining the interface between the two processes.

1.3 The Problem Statement

The problems that emerge due to the lack of coordination among the UE and SE processes during software development can be resolved by defining a development framework that *coordinates* and *synchronizes* the UE process with the SE process. This section discusses the issues to be addressed while defining such a framework and the approach towards the definition of the framework.

1.3.1 Issues to be resolved

The following issues need to be addressed in order to define a development framework that coordinates the UE process with the SE processes during software development:

- *Lack of understanding about impacting activities of the UE and SE processes.*
  Currently, the precise goals and objectives of constituent activities of the UE and SE processes are not well documented. Knowledge about the goals and objectives of those activities is required to identify activities in one process that can influence activities in the other.

- *Lack of adequate knowledge about the information that must be exchanged among UE and SE activities.*
  Activities of the UE and SE processes that influence one another need to synchronize and exchange the appropriate set of information. An understanding of the need for such exchange is required to identify interactions that must exist among these activities. Identification of these interactions provides the basis for defining an interface between the UE and SE processes.

- *Lack of understanding of the issues involved in modeling of a framework based on the information exchange among the UE and SE activities.*
  Multiple issues are involved in the definition of a coordinated development framework based on the exchange of information among the UE and SE activities. For example, when is synchronization needed as opposed to simple information exchange? These issues need to be identified and addressed in order to model effectively the coordinated development framework.

- *Definition of verification and validation approaches to ensure adherence to the development process.*

Adherence to the development process ensures that the necessary exchange of information between the UE and SE processes takes place. Adherence to the development process needs to be encouraged, enforced, and confirmed. Verification and validation approaches need to be defined in order to ensure adherence to the development process.

1.3.2 Our approach towards a solution

Development of a framework based on the definition of information exchange among influencing UE and SE activities is the primary goal of this research. To develop this framework we need to identify the information exchange that must happen among the UE and SE activities and define a coordination strategy among influencing activities.

1. *Definition of the information exchange among UE and SE activities*

   The definition of information exchange among UE and SE activities is the first important step in definition of a framework of interactions among UE and SE activities. To define this information exchange, it is necessary to identify influencing UE and SE activities and the information that must be exchanged among these activities.
2. *Coordinating and synchronizing influencing UE and SE activities*

   After identifying influencing activities of the UE and SE processes, it is necessary to coordinate and synchronize them with corresponding influencing activities from the other process. This is necessary because with concurrency among influencing activities, information exchange and activity awareness among these activities can be promoted.

Our approach towards the solution is as follows:

1. *Identification of the component activities of the UE and SE processes.*

   The component activities of the UE and SE processes need to be identified in order to establish information exchange among related activities from both processes. The identification of component activities of the UE and SE processes can be achieved by surveying relevant literature.

2. *Identification of the goals and objectives of the component activities.*

   A better understanding of the exchange of information that should exist among the UE and SE activities can be obtained by identifying goals and objectives of these activities. Like the identification of component activities of the UE and SE processes, their precise goals and objectives can be also be identified by surveying relevant literature.

3. *Determining influencing activities from both UE and SE processes.*

   The knowledge of precise goals and objectives of the UE and SE processes can be applied to the identification of influencing UE and SE activities. Along with the identification of influencing activities, the information exchange that must exist among them can be identified as well.

4. *Definition of interactions that ideally should exist among the different activities of the UE and the SE processes.*

   The identification of goals and objectives of the UE and SE processes helps us in identifying the information that must be exchanged among influencing UE and SE activities. Knowledge about the exchange of information can be used to define the interactions that should exist among these activities.

5. *Establishing concurrency among influencing UE and SE activities.*

   Coordination and synchronization among influencing UE and SE activities is necessary to facilitate information exchange among them. This coordination and synchronization can be achieved by having influencing activities of the UE and SE processes concurrent with one another.

6. *Identification of issues in the use of standard software engineering models to define a coordinated development process based on the framework of interactions.*

   A coordinated development process based on standard UE and SE models can be defined using the identified interactions. The incremental development model and the spiral model are candidate SE models. The scenario based design (SBD) process can be used to identify the UE activities. We provide a cursory examination of information flow and coordination among these models and then identify additional related issues for future work.

1.4 Blueprint of the Thesis

The remaining chapters of this thesis are set as follows. Chapter 2 discusses the background review conducted to gain a better understanding of the usability engineering and software engineering processes. *Conventional* and the *expanded* views of the UE and SE processes are discussed. The conventional view outlines the major phases of the UE and SE processes. The expanded view then describes the UE and the SE process in detail, as we see it. This expanded view identifies the component activities of each stage in the UE and the SE processes, highlighting the goals, objectives, and techniques of those activities. The Requirements Generation Model is discussed as a representative approach to requirements engineering.

In Chapter 3, we discuss the interactions among the activities of the Usability Engineering and Software Engineering processes. Synchronizations required among the activities of the UE and SE processes are highlighted in this discussion. The discussion also relates UE and SE activity objectives to identification of requisite information exchange that ideally should exist among these activities.

Chapter 4 presents a description of a coordinated framework that illustrates the interaction points and information exchange components between (a) an Incremental SE process and a UE process similar to the SBD and (b) a Spiral SE process and a UE process similar to the SBD. Additional issues that have to be addressed to define a coordinated development framework using standard software engineering models are also discussed in this chapter.

In Chapter 5, the last chapter, we summarize our research and present research topics for future work.

CHAPTER 2. BACKGROUND REVIEW

This chapter presents the background information that has had a major influence on the research presented in this thesis. We present the development methodologies and methods in the fields of Usability Engineering (UE) and Software Engineering (SE) as background information. Our motive behind discussing the UE and SE processes in detail is to achieve more substantial understanding of the SE and UE processes. This understanding will help us in identification of component activities of the processes, and the precise goals and objectives of those activities.

Section 2.1 presents the Software Engineering process. We discuss the advantages of having a structured software development process in this section... We also include a discussion of standard software engineering models.

In sections 2.2 and 2.3, we discuss in detail the requirements engineering and the software design phases of the software engineering life cycle. From the Y-model depicted in Figure 1.1, we observe that the interactions among the UE and SE process activities take place during the requirements and design specification phases of both processes. Identification of component activities of these phases is crucial for the identification of relationships among the activities. Therefore, while discussing the software engineering life cycle, we also discuss in detail the requirements engineering and software design phases.

In section 2.4, we highlight the importance of the Usability Engineering process and introduce the scenario-based development process for Usability Engineering. Finally, in section 2.5, we outline the important observations of other research efforts, directed towards the integration of the UE and SE processes.

2.1 The Software Engineering Process

Software engineering is a combination of methodologies, methods, and techniques that the software engineers employ during software development (e.g. CASE tools, automated testing tools, etc.). SE is performed by creative, knowledgeable people who work within a software development process that is appropriate for the products they build and demands of the marketplace[Pressman 2001].

In this section of the chapter, we define the Software Engineering process and look at its advantages. We also discuss several important Software Engineering models that outline development strategies used by that process.

2.1.1 The Software Engineering process and its advantages

Fritz Bauer [Pressman 2001] describes software engineering as "the establishment and use of sound engineering principles in order to obtain economical software that is reliable and runs efficiently on real machines." The IEEE [IEEE 1993] definition of Software Engineering states that Software Engineering is:

1. Application of systematic, disciplined, quantifiable approach to the development, operation and maintenance of software; that is application of engineering to software, and,
2. The study of approaches as in (1).

Early experience in building software systems showed that existing methodologies were inept. Major projects were, at times, years late and experienced heavy cost overruns [Standish 1995]. These projects were unreliable, difficult to maintain and performed inefficiently [Pressman 2001]. The inadvertent delays and significant budget overruns caused software production costs to rise. New techniques and methods were necessary to control the complexities inherent in large software systems.

The software industry has now made enormous progress in developing reliable software, more often than not, within budgetary and schedule constraints [Sommerville 1996]. A much-needed structure has been imparted to the overall software development process through a better understanding of various activities involved in software development. These activities have been modeled in the form of software development lifecycles (e.g. The Waterfall Model, the Incremental Development Model, the Spiral Model, etc.). Nevertheless, many large software projects are still late and over-budget. Moreover, software that is delivered often does not meet the *real* needs of the customer.

The SE process, like any other engineering discipline, tries to answer questions like [adapted from [Pressman 2001]]:

- What is the problem to be solved?
- What characteristics of the product entity can be used to solve the problem?
- How will the solution be realized and the entity constructed?
- How will errors be uncovered?
- How will the product entity be maintained over a long term when corrections, adaptations, and enhancements are requested by the users?

The work associated with software engineering can be classified into three generic phases. The *definition phase* focuses on understanding the functionality and performance desired, behavior expected, and constraints imposed on the system. The *development phase* focuses on tasks such as those related to definition of data structures, implementation of functions and procedures, and characterization of interfaces. It also deals with transferring software design to code and testing of the generated code. The *support phase,* also referred to as software maintenance, focuses on change in software after deployment. It is associated with changes required for error correction, adaptations required as the software evolves, and changes necessary to support new or changed customer requirements.

A team of software engineers who want to develop software should have a software development strategy that incorporates process, methods, and tools. These strategies of software development are called *software development models*. We will examine some of these models in the next subsection.

2.1.2 Software development models

Software development efforts contain four distinct stages: *status quo, problem definition, technical development,* and *solution integration* [Raccoon 1995] [Pressman 2001]. The *status quo* represents the current state of affairs. Problem definition identifies the specific problem that the software should solve. Technical development solves problems using some technology. Solution integration delivers results. A software development model outlines a development strategy that encapsulates processes, methods, artifacts, and tools to address these four distinct stages in software development.

Software development models typically consisted of five activities. These activities are requirements analysis, software design, implementation, integration and testing, and maintenance [Royce 1970]. Different software process models decompose these “generic” activities in different ways. The timing of the activities varies with the model used, as does the outcome.

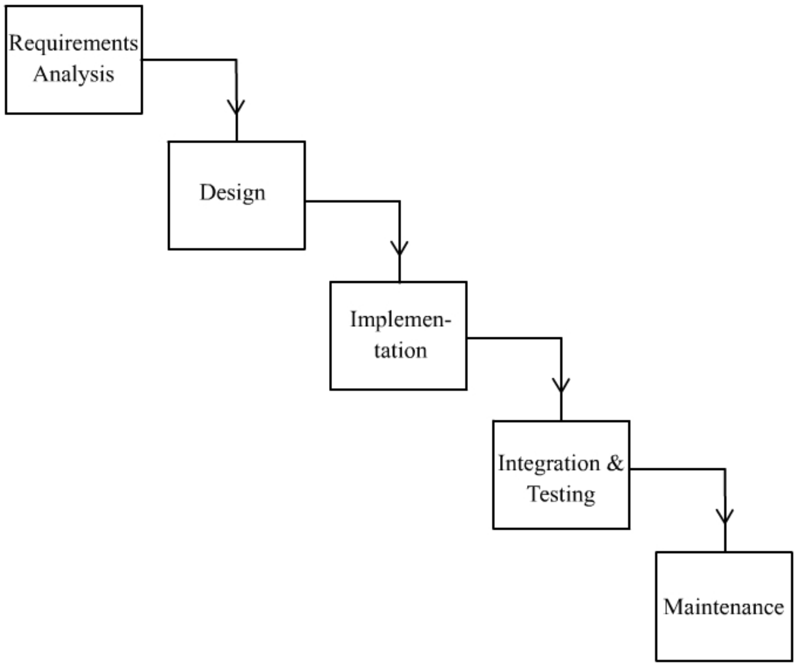

Figure 2.1 The Waterfall Model for Software Development

The first of the development models – The Waterfall Model [Royce 1970] offers means of organizing the development process. This model places the five activities of software development in a linear timeframe, with each phase following the previous one (Figure 2.1). The Waterfall Model places Requirements Analysis process at the start, leading to the development of a baseline document – the Software Requirements Specification (SRS). As advocated by the model, the SRS expresses what the system is to do. The shortcoming of the Waterfall Model is the inflexible partitioning of the project into distinct stages. The model fails to address the requirements creep problems [Carter 2001] that are an inevitable consequence of any development project.

The Evolutionary Development Model developed later is based on the idea of prototyping. This approach suffers from the drawback of consuming time and monetary resources until a correct, compliant product is developed. Reality constraints within the industry rarely afford the luxury of discarding fully functional prototypes and beginning development from the start. Examples of the evolutionary approach to software development include the Spiral Model [Boehm 1988], Rapid Application Development (RAD) [Martin, J. 1991] and the more recent "Extreme" Programming [Beck 1999].

2.2 The Software Requirements Engineering and Software Design Processes

In this section, we study the requirements engineering and the software design phases of the software engineering process. To discuss the requirements engineering process, we have used the Synergistic Requirements Generation Model (SRGM) [Sud 2003]. The SRGM, derived from the Requirements Generation Model (RGM) [Groener 2002], defines the various activities performed as a part of the requirements generation process. Section 2.2.3 [Adapted from [Sud 2003]] details each part of the requirements engineering process and makes explicit the goals of each of the activities performed under the SRGM.

2.2.1 The importance of Requirements Engineering

A closer examination of existing software development models reveals a common thread: they all contain a *requirements analysis* phase. This phase is usually one of the first phases in the model. The requirements analysis phase is composed of a series of activities that relate to the gathering and analysis of requirements from the customer. Although "first" in the sequence of phases, requirements analysis has been last in line for re-examination and refinement [Sidky 2002].

F.P. Brooks [Brooks 1987] mentions that "the hardest single part of building a software system is deciding what to build … no part of the work so cripples the resulting system if done wrong, no other part is as difficult to rectify later." In their empirical study, Bell and Thayer [Bell 1976] observe that inadequate, inconsistent, incomplete, or ambiguous requirements have a critical impact on the quality of the resulting software.

They conclude that “the requirements for a system do not arise naturally; instead, they need to be engineered.”

The late correction of requirements errors could cost up to 200 times as much as correction during requirements engineering [Boehm 1981]. The average cost of fixing errors that go undetected until the integration and testing phase is 36 times the cost of an early fix [Lewis 1977]. Moreover, the largest amount of errors in software is due to erroneous requirements and errors in requirements [Rubey 1975]. In the CHAOS Report [Standish 1995], the Standish Group reveals that five of the top eight reasons why projects fail are related to requirements. Therefore, the most important function that the software engineers perform for the client is the iterative extraction and refinement of the product requirements through the process of requirements engineering.

2.2.2 The Software Requirements Generation process

The somewhat inappropriate name of the *requirement analysis* phase, contributed earlier to the perception of the activities therein. Activities like *problem analysis* and *requirements elicitation* were considered as minor activities [Davis 1993]. Subsequently, only within the last few years have we seen a meaningful refinement of “requirements analysis,” recognizing the major activities underlying requirements generation. The software engineers are constantly reinforcing the importance of a well-defined, effective set of requirements engineering activities. That reinforcement is being achieved with increasing knowledge about relationships between the quality of a product and the quality of requirements from which it is developed.

Software development models place the Requirements Generation process at the beginning. The process is composed of the activities listed below:

- *Requirements Elicitation* – Software engineers elicit software requirements from people or derive them from system requirements. An important precursor to the elicitation process is the *problem synthesis* process. During problem synthesis, the engineers diagnose the underlying issues and elicit customer needs.
- *Requirements Analysis* – (adapted from [Brackett 1990]) Software engineers usually perform requirements analysis before the customer commits to the actual development process. The customer assesses the acceptable level of risk regarding

the completeness, correctness, technical feasibility, and cost required to develop the system.

- *Requirements Specification* – The software engineers document and express the requirements elicited and analyzed in the preceding phases in the form of a formal document. This software document is often referred to as the Software Requirements Specification (SRS).
- *Requirements Verification and Validation* – The software engineers ensure that the requirements elicited and specified in the SRS adhere to the customer needs or the high-level system requirements. They present the requirements elicited to diverse audiences for review and approval, and test adherence of specified requirements to pre-defined quality attributes.
- *Requirements Management* – Leffingwell and Widrig define the Requirements Management process as, "a process that establishes and maintains agreement between the customer and the project team on the changing requirements of the system" [Leffingwell 2000]. The Requirements management methodology permeates throughout the entire process and facilitates easy communication of change among the software engineers.

The next section details each of these phases in the requirements engineering process and highlights the goals and activities of these phases.

2.2.3 The Requirements Generation phase

To detail the Requirements Engineering phase, we use the Synergistic Requirements Generation Model (SRGM) from [Sud 2003]. The SRGM defines the entire process of requirements engineering, the analysis of the problem as described by the customer to the specification, validation, and maintenance of requirements.

Figure 2.2 [Adapted from [Sud 2003]] gives a high-level view the SRGM. The diagram shows the different phases of the requirements engineering process defined by the SRGM. Much of the following description is also adapted from [Sud 2003]. This subsection details the Requirements Generation process and describes the process activities for each phase, listing their objectives.

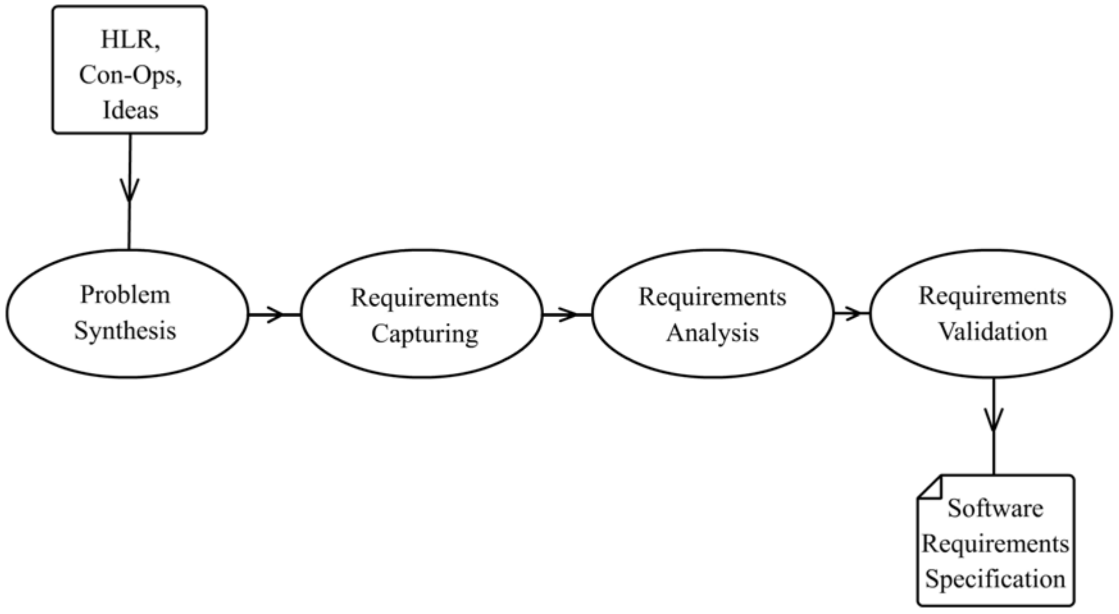

Figure 2.2 An overview of the Synergistic Requirements Generation Model (SRGM)

2.2.3.1 Problem Synthesis

The problem synthesis phase is depicted in Figure 2.2. Problem synthesis begins with the *education* of requirements engineers about the system to be developed. The goal of this education is that the requirements engineers obtain:

- preliminary information about the current and proposed system,
- identify the context in which the proposed system is to be deployed, and
- interact with stakeholders to learn the organization practices relevant to the system

Education of the requirements engineers is followed by the *problem analysis*. Problem analysis consists of three parts:

1. *Problem Identification*, during which the requirements engineer identifies stakeholders and conducts investigations based on the input provided by the customer.
2. *Problem Decomposition*, during which the requirements engineer, in consultation with the stakeholders, gains a deeper understanding of the problem identified earlier.
3. *Context and Constraints Analysis*, during which the requirements engineers determine the operating context and constraints offered by the environment within

which the system will be deployed. These constraints can also include external factors that may constrain the system.

Needs generation follows problem analysis. The output of the problem analysis includes the problem statement, problem elements, and all necessary clarifications (in the form of Context Diagrams and Constraints Documents). The objective of the Needs Generation process is to generate a set of customer needs in consultation with the customer and user. The needs generation process is performed in three parts:

1. *Needs elicitation*, during which the requirements engineers derive customer needs from the problem elements formed by decomposing the problem into its constituent problem elements. The requirements engineers examine every problem element to determine the needs associated with that problem element.
2. *Needs analysis*, during which the needs are analyzed from the perspective of converting them to requirements. The requirements engineers organize the needs generated from the elicitation meeting. After the needs are generated, the requirements engineers relate and analyze those needs with respect to the constraints and context established during Problem Analysis.
3. *Needs evaluation*, during which the requirements engineers conduct an evaluation of needs to determine the "reference" level. They separate the needs to be incorporated from the ones that are to be left out or postponed for a future release. The requirements engineers determine this "reference" level based on the schedule and budgetary restrictions in consultation with all stakeholders. They also evaluate the needs to verify the right information and decide whether iteration is required to derive needs for the particular problem element.

Once the stakeholders are convinced that sufficient needs are generated, we proceed to requirements capturing.

2.2.3.2 Requirements Capturing

Figure 2.2 depicts the requirements capturing phase as the second phase in the SRGM. Requirements capturing uses the needs document created during needs generation and culminates in the generation of software requirements of the proposed system. The requirements engineers may precede the capturing of requirements with an optional process entailing the indoctrination of the customer. The following activities are performed during requirements capturing:

1. *Requirements Elicitation Meeting:* The requirements engineers conduct requirements elicitation meetings to elicit information from stakeholders. The primary objective is to identify and capture requirements as communicated by the stakeholders.
2. *Local Analysis:* The requirements engineers use the local analysis process to analyze requirements *locally*. For a reasonably large system, eliciting requirements for the entire software system in one sitting is almost impossible. Multiple meetings, involving a diverse set of stakeholders, must be conducted to ensure complete coverage. During these meetings, the requirements engineers generate and document the elicited requirements. Local analysis helps the requirements engineers analyze the requirements locally with respect to risk, effort, priority, etc. The local analysis aids the *global analysis* of requirements during the *requirements analysis* phase.
3. *Requirements Evaluation:* Requirements Evaluation verifies the adherence of requirements to requirements quality attributes. The quality attributes include consistency, completeness, correctness, unambiguity, testability, understandability, and traceability. Requirements evaluation helps uncover inconsistencies and redundancies in the requirements. Requirements Evaluation also helps the requirements engineers determine the need of an additional iteration through the requirements capturing phase.

2.2.3.3 Requirements Analysis

The requirements analysis phase is the third phase depicted in Figure 2.2. The analysis of requirements helps the requirements engineers determine if they are building the correct product with respect to customer needs and budget. Requirements analysis incorporates the following activities:

1. *Global Analysis:* Global Analysis activities are similar to those conducted during Local Analysis, but are conducted on the complete set of requirements. The requirements engineers collectively analyze attributes associated with the requirements by the local analysis conducted earlier. This analysis is conducted to assess the overall progress of the project and to assess feasibility from the market, sales, and financial perspectives. Global analysis includes a risk analysis based on the risk associated with the requirement by the local analysis. During global analysis, the requirements engineers perform a cost and schedule estimation in consultation with the software engineers who implement the system. They perform *price/market analysis* considering the target customer and market conditions and *feasibility and profitability analysis* with respect to the product.

2. *Requirements Specification:* The requirements engineers organize and document the requirements captured and analyzed during requirements specification. They produce the *Software Requirements Specification* (SRS). The SRS not only guides further software development, but also acts as a contractual agreement between customers and developers. The SRS should include all requirements for the software system and should follow a standardized format that makes information retrieval easier for designers, developers, and customers.

   The SRS produced after requirements specification is the *unvalidated* SRS. The requirements engineers validate the unvalidated SRS against the original system requirements during the requirements validation phase that follows.

2.2.3.4 Requirements Validation

Figure 2.2 shows the requirements validation phase as the fourth phase in the SRGM. During requirements generation, software requirements are continually evaluated, as and when they are generated. This evaluation ensures that the requirements adhere to quality standards and meet the customer intent. Finally, the requirements engineers validate the requirements contained in the SRS against the needs and problem statement for correctness, quality, and consistency. Requirements validation consists of the following activities:

1. *Requirements Adherence:* Requirements Adherence comprises activities that ensure the adherence of requirements to expected conditions and standards. Requirements adherence includes *requirements traceability*, to ascertain the links among the requirements, needs, and other system elements. Customers and

software engineers use *Formal technical reviews* to seek mutual agreement about the overall SRS between customers and software engineers. The formal technical reviews provide a written report identifying irregularities in the SRS. Irregularities in requirements are expensive to fix later. The requirements engineers should, therefore, ensure the correctness of requirements and removal of all irregularities through rework and correction of requirements.

2. *Configuration Control:* Changes to the requirements may occur during the design, coding, or testing phases of the software development life cycle. After all the requirements are approved, the SRS is base-lined according to the organization's configuration control policy. This maintains consistency of requirements in case of a later change in the requirements. Later changes need to go through a stakeholder approval process and the configuration control before they can be implemented.

The next subsection describes the software engineering design process in detail.

2.3 The Software Engineering Design process

We can divide a typical Software Engineering Design process broadly into two sub-processes, *Structured Analysis* and *Structured Design*. Structured Analysis models the system as a set of functional and behavioral models. Structured Design, on the other hand, uses those models to reach the design of software components that can be integrated to form the system. The following subsections describe the structured analysis and structured design phases.

2.3.1 Structured Analysis

Modeling tasks are necessary after the specification of requirements to reach a complete software design specification. The models developed are called *analysis models* and are the first technical representation of software. *Structured Analysis* is the classical approach to analysis modeling. In Structured Analysis, the analysts create and partition data and also create functional and behavioral models that depict the essence of the software system under development. Pressman states, “Data modeling defines data objects, attributes, and relationships [Pressman 2001]. Functional modeling indicates how data are transformed within the system while behavioral modeling depicts the impacts of events."

An analysis model must achieve three primary objectives:

1. Describe what the customer requires.
2. Establish a basis for creation of software design.
3. Define a set of requirements that can be validated once the software is built.

The *data dictionary* forms the core of the analysis model. It contains a description of all the data objects produced or used by the software. The *entity relationship diagram* (ERD), the *data flow diagram* (DFD), and the *state transition diagram* (STD) are also created during structural analysis. The ERD serves the purpose of depicting relationships among the data objects defined in the data dictionary. Software designers use the ERD to conduct the data modeling activity during data design. DFDs serve the purpose of providing an indication of the data transformation as data moves through the system, also indicating the functions that transform data. Software designers use DFDs during modeling of functions for the software system. The description of each function presented in the DFD is made available to the software engineers through a process specification (PSPEC) document.

Several data processing applications can be modeled using the DFDs with the data model. However, for applications that are "driven" by external events, specification of control flow becomes necessary [Pressman 2001]. System analysts use *control flow diagrams* (CFDs) to depict the flow of control in the software. The CFDs are especially useful for the system architects in designing the component interfaces for the software system.

An indication of the behavior of the system as a consequence of external stimuli is made available to the system analysts using a *state transition diagram* (STD). Through the STD, the system analysts depict the states or modes of behavior of the system and manner in which the transitions from state to state take place. The *control specification* (CSPEC) contains additional information about the control aspects of the software. The use of STDs and the CSPEC artifacts is important used during the design of software components for a system.

The *software design* process follows the process of analysis modeling. Software design uses the artifacts (ERD, DFDs, STDs, CFDs, etc.) produced during the analysis modeling to produce functional specifications of the software system. The software designers can supply these functional specifications to the programmers for implementation. In the next subsection, we describe the software design process and the different stages in the process.

2.3.2 The Software Engineering Design Process

Software Design sits at the technical kernel of software engineering and is applied regardless of the software engineering process model being used [Pressman 2001]. Software Design follows requirements analysis and specification. It is the first of three technical activities performed after requirements specification, the other two being code generation, and integration and testing. Software requirements manifested by data, functional and behavioral models feed the Software Design [Pressman 2001]. Several design methods that transform software requirements to design are available. All of them produce a data design, an architectural design, an interface design, and a component design through corresponding activities within the design phase.

In this section of the chapter, we describe the production of the above design documents within the design stage.

2.3.2.1 Data Design

Data design transforms the information domain model created during analysis into data structures required to implement the software [Pressman 2001]. The entity relationship diagram and the data dictionary provide the basis for data design. A part of data design occurs with design of software architecture, but the detailed design occurs with the design of each software component.

During the data design phase, the software designers apply analysis principles to data with respect to function and behavior. They develop representations of content and data flow in the system and identify data objects. The designers also consider other alternatives to the data organization they have represented and evaluate the impact of the data modeling on the software design to evaluate alternatives. The designers examine the system functionality to identify the appropriate data structures based on the knowledge of the operations that the system would perform on them. Libraries of useful data structures can be of use here. The designers also define abstract data types (classes in OO Programming) to simplify software design. Data and program design is defined in the data dictionary, and low-level data design is deferred until later in the design process. The designers ensure the presence of adequate information hiding and low coupling among functions during the preliminary design itself.

2.3.2.2 Architectural Design

The architectural design process helps software architects define the relationships among major structural elements of software and patterns that can be used to achieve necessary software functionality. During the architectural design, software architects also look at the constraints that affect the way in which the patterns can be applied [Pressman 2001].

The software architects examine and evaluate different types of *architectural schemes* (call and return design, object oriented design, data centered design, layered design, etc.) that can be applied to the software system. The architects weigh the pros and cons of these architectural schemes using a qualitative approach like tradeoff analysis. They also use quantitative methods to evaluate the quality of software design and to assess the overall complexity of the architectural design. These quantitative methods include spectrum analysis, design selection analysis, and contribution analysis.

2.3.2.3 Mapping Requirements to Architecture

Software requirements require a mapping to the architectural styles. A comprehensive mapping that transforms requirements to architectural styles does not exist. Data flow diagrams can be used to derive the software architecture using a process called *Structured Design.* Structured Design provides a convenient transition from data flow to the software architecture. This transition happens through (1) the identification of the type of data flow, (2) indication of flow boundaries, (3) mapping of the DFD into program structure, (4) definition of the control hierarchy, (5) definition of the resultant structure using design measures, and (6) refinement and elaboration of the architectural description [Pressman 2001].

The transition from DFD to program structure necessitates an establishment of the type of information flow in the system. The software designers review the fundamental system model, review and refine the DFDs, and determine whether the DFD has a *transform* or a *transaction* flow. A transform flow exists in a segment of the DFD if the overall data flow is sequential and follows one or only a few data paths. On the other hand, a transaction flow exists if a single data item, called a *transaction*, triggers data flow along several paths.

If the flow is a transform flow, the software designers isolate the transform center by specifying incoming and outgoing flow boundaries. The designers then factor the DFD to levels with increasing detail. If the flow is a transaction flow, the software designers identify the transaction center and examine flow characteristics along each action path. The designers then map the DFD to a program structure compatible with transaction processing. They also factor and refine the transaction structure for each action path.

Refinement of the first architecture iteration using heuristics may be necessary for both transaction and transform flows. After using the above measures to identify transform or transaction flow, the designers indicate the flow boundaries for data within the system to program structure and map the DFD to program structure. The software designers define a control hierarchy in the program and measure the resultant structure using design measures and heuristics. They may further refine the design and elaborate its architectural description.

2.3.2.4 Interface design

An interface implies a flow of information and specific type of behavior [Pressman 2001]. Interface design describes three types of communication interfaces for the software. These interfaces are:

- *Interfaces within the software for inter-component communication.*

Software engineers decompose the system into constituent components that are developed separately and integrated into a complete system. Interfaces among these components have to be designed and formally documented in order to make possible their integration into a complete software system. A major responsibility of the interface design stage is to define these inter-component interfaces.

- *Interfaces for communication of software with systems that interoperate with it.*
  A software system has to interoperate with other software and hardware systems already operational in the user's environment. The software engineers have to identify the systems with whom the software under development has to interoperate. Communication interfaces among the interoperating systems are established during the interface design stage.
- *Interface for communication with human users.*
  The human-computer interface is another major interface that the software system has to provide. The human-computer interface, also called the *user interface,* has to be designed to be easy to learn and use. Moreover, the user should be able to complete his task, and feel satisfied after using the system. Without an adequate user interface design, these goals are impossible to meet. Usability engineers use a formal Usability Engineering process to design the user interfaces to be provided by components of the software. The SE interface design phase defines the necessary user interfaces at a high level based on the specifications suggested by the usability engineers.

2.3.2.5 Component Level Design

Component level design occurs after data, architectural and interface designs have been established. The component-level design transforms the software architecture into a procedural description of software components. The internals of the components are defined during the component design stage. The data structures, algorithms, and the flow of data within the component are designed during the component level design. Information from the process and control specifications and the state transition diagrams help component design [Pressman 2001]. Software designers use design notations such as graphical or tabular notations or program design language (PDL) during component level design.

The user interface for the software system is provided by the components of the software system and is designed during component level design. Therefore, an important responsibility of the component level design is to design, at an implementation level, the user interface provided by the usability engineers. The usability engineers provide the software engineers with their specifications about the design of the user interface to be provided by software components. We should note that not all these specifications might be implementable for the software engineers. Therefore, the software engineers are required to convey the implementation level constraints to the usability engineers. These implementation level constraints should be provided before the finalization of the usability specifications to ensure inclusion of the provided usability specifications into the design of the software components.

The next section discusses the Usability Engineering process. We describe the significance of Usability Engineering and introduce the scenario-based design (SBD) approach to usability engineering. We also introduce other approaches to usability engineering and highlight their more notable features. In the last part of the section, we discuss the SBD approach in detail and highlight the goals of activities performed as a part of each phase of the SBD process.

2.4 Usability Engineering

Usability Engineering (UE) refers to concepts and techniques for planning, achieving, and verifying objectives for system usability. The Usability Engineers must define the usability objectives early in software development. They must then assess these objectives repeatedly during the development process to ensure that the system meets the required usability standards. Earlier, UE focused on the design of the user interface and engineering of effective, interactive presentations. Recently, UE has extended its focus to encompass the Software Engineering activities, particularly, system envisioning and requirements generation [Rosson 2002].

2.4.1 The importance of scenarios in Usability Engineering

Usability Engineering relies on user interaction scenarios, stories about people and their activities [Rosson 2002]. Technological advances in computing supply new opportunities for human activities, changing task structures for people. In response to these opportunities, new needs for technology arise [Rosson 2002], and the cycle continues. Interaction scenarios capture these new opportunities for improvement due to advances in technology.

Scenario-based methods consider descriptions of people using technology essential in discussing and analyzing the reshaping of activities. An advantage of scenario-based design is that usability engineers can create scenario descriptions and feel their impacts before the system is built. Scenarios have a *plot*. They also include sequences of actions and events, actions that actors perform and things that happen to them, changes in the setting, etc. Actions and events may be useful, obstructive, or irrelevant to goal achievement. Representation of a system with a set of user action scenarios makes the system's use explicit, and orients design and analysis towards the system goal [Rosson 2002].

Scenarios are of special importance to UE because:

1. Scenarios can be very useful in managing the tradeoffs of UE.
2. Scenarios help usability engineers respond to current interaction needs and anticipate new needs.
3. Scenarios present a universally accessible language to represent design.
4. Scenarios help the usability engineers during the reflection and analysis of the design.

2.4.2 Scenario-Based Usability Engineering

The scenario-based UE process (SBD) is depicted at a high-level in Figure 2.3. The SBD process has five *iterative* and *interleaved* stages: *requirements analysis, activity design, information design, interaction design,* and *prototyping and evaluation.* During the requirements analysis phase of the SBD process, the usability engineers study the problem through interviews with stakeholders and from field studies of the current situation.

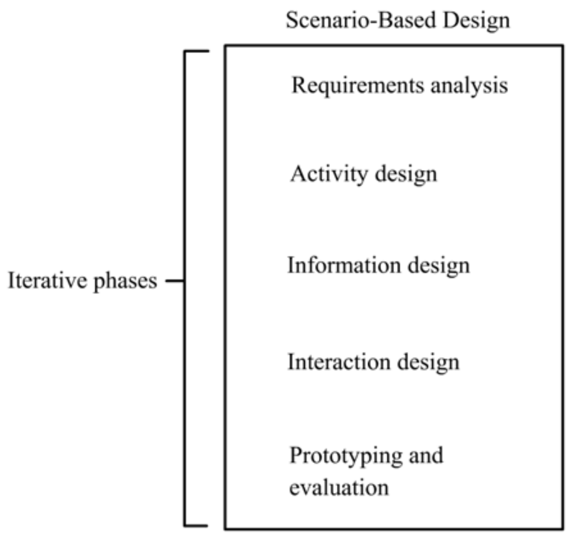

Figure 2.3 High-level representation of the SBD process

Using the information they collect from this study, they formulate problem scenarios that convey important characteristics of the users, including typical and critical tasks, tools used, and their organizational context. During requirements analysis, scenarios promote reflection and discussion and facilitate mutual understanding and communication among different participating groups.

SBD organizes the design stage into three sub-stages. These stages are depicted in Figure 2.3. The first stage, *activity design*, is the envisioning of activity scenarios, narratives of services the users will seek from the system. The activity scenarios provide a correct, early glimpse of the future that the usability engineers are trying to design. The second stage, *information design*, produces information scenarios. The information scenarios provide details about the information that the system will provide to its users. The third stage, *interaction design*, involves the design of interaction scenarios. Each interaction scenario is a fully specified design vision that specifies actions the users take to interact with the system and the responses the system provides for the users' actions.

SBD suggests the evaluation of design ideas in a continuing fashion throughout the design process. The usability engineers often use prototypes to achieve continuous evaluation. The prototypes can have different degrees of completeness and polish depending on the purpose behind their creation. SBD distinguishes between summative evaluation, which is a validation function, and formative evaluation, which is a more continuous process that is used to improve system design.

2.4.3 Other approaches to usability engineering

Several other approaches to usability engineering exist. Mayhew's usability lifecycle shows a linear flow of usability design activities, accompanied by iterative feedback and reworking.

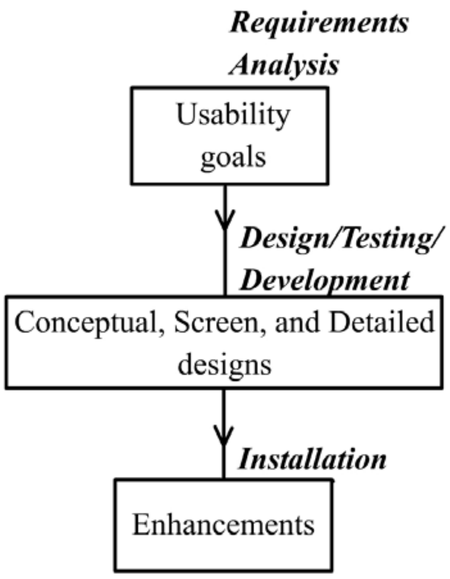
Requirements Analysis
Usability goals
Design/Testing/ Development
Conceptual, Screen, and Detailed designs
Installation
Enhancements

Figure 2.4 High-level representation of Mayhew's usability lifecycle

Mayhew's usability lifecycle [Mayhew 1999] depicted in Figure 2.4 contains five major phases: *requirements analysis, conceptual model design, screen design, detailed user interface design,* and *installation*. All phases of the lifecycle except requirements analysis include assessment activities local to the phase. Although the requirements analysis does not include iterative analysis, an unsatisfactory assessment of the complete design after the design phases can return the process to the analysis of requirements. The design phases themselves include internal, iterative analysis activities. An important difference between the SBD and Mayhew's usability lifecycle is that the SBD uses scenarios as a design representation throughout the design process.

The Hix-Hartson usability design process [Hix 1993] is depicted in Figure 2.5. It is an iterative, evaluation-centered process to design user interaction. The process uses *needs analysis, user analysis, task analysis, functional analysis,* and *design requirements analysis* as early analysis activities. The interaction design follows these early activities. The first stage in the design phase is the *conceptual design* of the system.

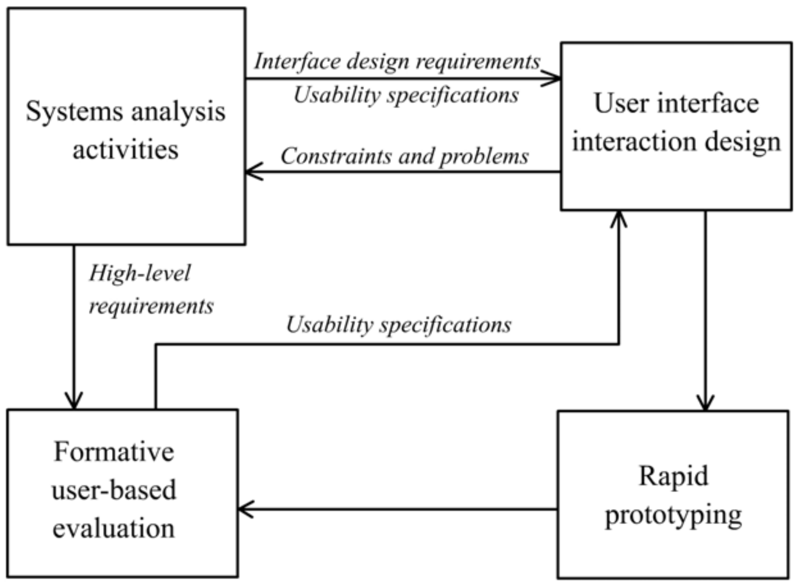
Systems analysis activities
Interface design requirements
Usability specifications
Constraints and problems
User interface interaction design
High-level requirements
Usability specifications
Formative user-based evaluation
Rapid prototyping

Figure 2.5 High-level representation of Hix-Hartson usability design process [Adapted from Hix 1993]

The conceptual design produces a user interaction design that is independent of appearance and includes the operations that would be invoked and carried out on the conceptual objects. The conceptual design is followed by the *initial scenario design* stage. A few initial screen layouts are developed in the first iteration of the initial scenario design stage. An early evaluation follows the initial design and solicits early feedback from the user who visualizes using a system formed from the initial system design. After analyzing the user comments obtained through evaluation, the initial design is improved to obtain a well-developed visual design layout. The iterative process of interaction design and evaluation continues until the required usability specifications are met by the design.

2.5 The Scenario-Based UE process detailed

As mentioned earlier in this chapter, requirements analysis, activity design, information design, interaction design, and prototyping and evaluation are the five iterative and interleaved stages of the SBD based usability process. To identify the component activities of each of these stages and understand their goals, we need to accrue in-depth understanding of these phases.

The following diagram (Figure 2.6) [Adapted from [Rosson 2002]] depicts the scenario-based UE design process (SBD).

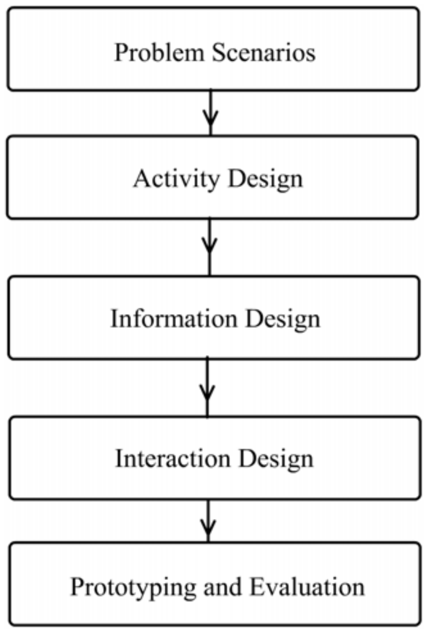

Figure 2.6 The SBD Approach to usability engineering

Although the diagram shows the process activities as a progression, all the process activities happen in an iterative, interleaved fashion. The process uses scenarios to analyze requirements, envision new designs, guide prototyping, and organize evaluation [Rosson 2002]. The following subsections 2.5.1 – 2.5.4 are adapted from [Rosson 2002] and describe the different phases of the SBD process in detail.

2.5.1 The Requirements Analysis phase

The requirements analysis phase is the first phase in the scenario based UE process depicted in Figure 2.6. Root concept design, creation of problem scenarios, and claims analysis represent three important activities from the requirements analysis phase.

Rosson and Carroll state that the root concept "is multifaceted, includes a statement of project vision and rationale and an initial stakeholder analysis, and an acknowledgement of starting assumptions that will constrain or otherwise guide the development." The project vision may be obtained from the management or clients through open-ended discussions. Identification of stakeholders, people who have stakes in the project outcome, helps develop a vision of the project. Starting assumptions have a major impact on the project, as it is necessary to consider them upfront.

The root concept and problem scenario development, therefore, identify the following:

- The purpose of the project, influencing factors, and needs of the users.
- Characteristics of the environment of use and user characteristics.
- Constraints that affect the development of the system and impacts of these constraints.
- Required system performance characteristics and external dependencies that affect design.

Contextual enquiry, prototyping for problem understanding, and *ethnographic observation* are techniques used to identify the above-mentioned influencing factors.

Usability engineers conduct field studies to analyze the current practices at the location where the system being designed is to be deployed. After the field studies, the usability engineers conduct a task and artifact analysis and extract workplace themes. These workplace themes help them reach problem scenarios that tell the story of the current practice. The usability engineers carefully develop the problem scenarios to reveal aspects of the stakeholders and their activities that have implications on the design of the software system. Problem scenario writing is interleaved with claims analysis, where the usability engineers identify features of the situation that may affect the stakeholders. It should be noted that problem scenarios and claims do not specify actual requirements, but suggest requirements implicitly by describing the needs and opportunities in the current system.

The next phase, *activity design*, is geared towards defining the functionality provided by the system. The usability engineers base activity design on requirements analysis and design activities that fulfill system requirements.

2.5.2 The Activity Design phase

Activity Design is the first phase of the design process as depicted in Figure 2.6. During this phase, the problems and opportunities of the current system are transformed into new ways of behaving [Rosson 2002]. The goal of activity design is to specify system functionality. System functionality influences the user’s experience while using the system and is the essence of an interactive system. Unless the system addresses genuine user goals and concerns, it does not give the user a satisfying experience. Activity design helps design systems in a usage context, giving importance to if and how the system supports human goals and activities. In the SBD, activity design is performed distinct from the information and interaction design phases and is performed prior to these phases because:

1. The activity design stage helps design activities that the users will find effective, comprehensible, and satisfying.
2. Analyzing user interface needs and choosing appropriate display and interaction techniques are difficult when the system functionality is not known.
3. Focusing on activities first creates a natural boundary between the system functionality and the UI software, and makes it easier to construct alternate user interfaces.

Activity design tries to address the following issues as a part of the development of an initial concept of system functionality:

- The basic concepts and services provided by the new system, and design of these activities to be effective in satisfying stakeholder needs.
- Issues to be considered while designing a new system and identifying a comprehensive set of requirements.
- Opportunities to improve a current system.
- Information held by the system, kinds of operations permitted on this information and results generated by those operations.
- Problems that exist in the current system, if any. Assumptions that constrain the design of new activities.

The activities designed by the usability engineers during the activity design phase should be effective and comprehensible. To design effective activities, the usability engineers may use participatory design, where they collaborate with the users to reach the design. They may also study how task level information is distributed throughout a situation in many different forms like knowledge and memories of people involved, state of tools and artifacts in use, etc. Scenarios are of a special importance here as they allow the usability engineers to think about aspects of activities that are best supported by the software and apply new ideas to a realistic setting.

It is important to design activities that the users should be able to tell what goals are possible and whether they are making progress towards achieving their goals [Rosson 2002]. The *designer's model* of the system, with the usability engineers' understanding of the information and tasks that will constitute the system, begins to form during activity design. Use of activities designed during the activity design phase should be satisfying for the users. One way to satisfy users is to automate tedious or error-prone tasks. Too much automation, however, works against providing satisfaction of use to the users. Activities designed for systems that may be used by a group or an individual should not only support group activities, but should also be pleasurable for individual use. Usability engineers can employ techniques such as *visual brainstorming, scenarios and post-It notes, essential use-cases, GOMS analysis,* and *operational modeling of systems* to reach a quality activity design. Activity design also contains a validation component that ascertains that the activities designed are computationally feasible and can be implemented. The usability engineers also have to ensure that the stakeholders approve of the functionality provided by the activity design.

After designing the basic system functionality, the usability engineers work towards an effective user interface design. The information design and interaction design phases detailed below guide the usability engineers towards a high-quality user interface design.

2.5.3 The Information Design phase

The information design phase depicted in Figure 2.6 helps usability engineers arrange and represent objects and actions in a system in a way that facilitates perception and understanding for the user. Usability engineers aim to arrange visual information in the user interface in a way that helps the user to perceive, interpret, and make sense of the information and, thus, helps him cross the *Norman's Gulf of Evaluation* [Norman 1988]**.**
Three important goals of information design are to:

- Design and configure the layout of the user interface in a way that it helps the users' perception of information.
- Group together individual display elements that are similar to one another, and aid user perception.

- Design a user interface with controlled complexity while displaying essential entities to give the user an understanding of control.

User interfaces contain many levels of perceptual structure. Basic display elements (pixels, contours, characters) are grouped into higher order structures (icons, paragraphs). The information design activities have to ascertain that this hierarchy of perceptual structures gives an immediate sense of organization to the user. At the same time, the individual control elements from the interface should also be identifiable to the user.

The interface design should help the user interpret the information and make sense out of it. The usability engineers should engineer the visual design of the interface to support user interpretation. The usability engineers also need to make sure that the users understand the meaning of the display elements in context of the software system and that the control elements offer adequate affordance.

While designing the user interface with the above-mentioned considerations, the usability engineers also have to ascertain that the interface looks consistent in shape, size, icon characters, font, and layout.

The next stage in the design of the user interface is to design the interactions between the user and the software system. The *interaction design* phase helps the usability engineers in designing effective interactions.

2.5.4 The Interaction Design phase

The interaction design phase depicted in Figure 2.6 helps the usability engineers in specifying mechanisms for manipulating and accessing task information. It, therefore, helps the users bridge the *Norman's Gulf of Execution* [Norman 1988]. The interaction design phase helps the usability engineers reach a design that:

- Helps the user identify system goals.
- Provides the user with enough visual information to plan action sequences to reach the system goals.
- Aids the user in execution of the action sequences planned.

The usability engineers design the user interface that includes interactions that help the users translate their task goals to system goals. The user interface helps the users identify possible interactions with the system and develop an action plan. The usability engineers simplify complex actions for the user. Some of the techniques they can utilize to simplify tasks are use of simple and natural dialogue, consistency in the user interface design, clearly marked exits, and provision of shortcuts. The usability engineers provide some degree of flexibility in the action plan and actions on the user interface for higher user satisfaction.

The user interface must help the user in executing the action plan. Minimization of the memory load, provision of good error messages, and mechanisms to prevent errors help the user in successfully executing the action plan. Adequate amount of online help and user documentation can also be of use to the users in execution of actions. The usability engineers try to relate physical actions to user actions for conceptual task implementation. An example of this is the support to drag and drop an object into the recycle bin to delete it. The user interface should also provide the user with enough feedback for his actions. The usability engineers also have to take decisions on low-level details about selection techniques and input events, specification of input and output devices, and the definition of interactions. The usability engineers supply adequate feedback to the users through the user interface while designing these details.

The usability engineers have to ensure that the design matches the user interaction requirements specified for the system under development. Continual prototyping and evaluation of the user interface prototypes is an effective way to verify that the user interface design matches the required usability specifications.

2.5.5 Prototyping and the Iterative Design phase

Usability evaluation is an analysis or empirical study of the usability of a prototype or the actual system. SBD assumes that the usability engineers will evaluate the design ideas in a continuing fashion. Evaluation of prototypes is an effective method to achieve this evaluation. Prototypes can take several forms, from a very rough sketch to a high fidelity working model of the system. The degree of completeness and polish required depends upon the stage at which the prototype is built and the purpose it should serve.

Evaluation of the design can be distinguished into two types, formative or summative. Formative evaluation is used as a guide to redesign and is aimed at improving the design prototype. Formative evaluation takes place during the design phases. It attempts to identify defects in the design through evaluation of the prototype and guides the usability engineers in fixing the defects. Summative evaluation is used to measure overall quality of the design and is essentially a validation procedure to validate that the system built matches the original usability specifications.

Figure 2.6 gives a high-level representation of the SBD framework to usability design. It is important to note that the design phase made up of activity, information, and interaction design stages is iterative in nature and iterates between analysis of usability claims and redesign to match the claims. Prototypes are designed and evaluated from the system design using formative evaluation that takes place iteratively during the design process. Summative evaluation, which is a validation activity, is used to ascertain that the right system has been designed. We have stated this earlier and reiterate that the diagrammatic representation of the SBD in a top-down fashion does *not* imply a waterfall process. All the activities in the SBD are iterative and interleaved. We make a special note of this fact at this point to stress that prototyping and evaluation is interleaved with the design and not conducted after the design process.

In the earlier sections of this chapter, we have detailed the SE and UE processes to understand the goals and activities of each phase of the processes. In the final section of the chapter, we present a brief overview of the other efforts directed towards the integration of the SE and UE processes and highlight their achievements.

2.6 Other Efforts

In this section of the chapter, we present an overview of the other researchers' efforts directed towards the integration of Usability Engineering into the Software Engineering process. We also highlight the important observations that have emerged through these research efforts.

2.6.1 Paech and Kohler

Paech and Kohler [Paech 2003] advocate engineering of the user interface during the requirements elicitation phase in order to include UI considerations during requirements elicitation. They claim that requirements engineers need to address UI considerations during requirements elicitation and add that it is incorrect to address HCI considerations after the requirements elicitation is complete.

2.6.2 Milewski

Milewski in his paper [Milewski 2003] focuses on usability and software engineering education and addresses some issues related to integration of the UE and SE processes. Milewski claims that combining the UE and SE teams is not possible because:

- Several application functions are neither directly related to the user, nor would benefit from user involvement. Such functions may exist in the software functionality.
- Having the user advocate semi-separated from the schedule and budget demands of the rest of the project is best.
- Too much work may be required to combine positions and to interface others' roles in the project (systems engineers, developers, designers, marketers, etc.), and this work is typically too much for a single role (combined usability and software engineer) to handle.

Milewski further claims that creating a common integrated process model will not make the situation more manageable because:

- Process models in practice are adapted to fit the specifics of the environment and the needs of the specific project.
- Process models are more descriptive of what actually happens rather than what must happen.

According to Milewski, reaching a common terminology for both the disciplines is unlikely to solve any issues related to integration of the UE and SE processes because:

- Creating new terms for overlapping concepts will complicate the communication between the UE and SE teams.

- Synonymous or not-quite-synonymous terms may also complicate the situation because old terms containing specific meaning for either of the processes continue to be in use while the new terms, neutral in nature, are being created.

2.6.3 Natalia Juristo

Juristo and Lopez explore the use of architectural patterns designed to enhance the usability of software products. They present their research towards application of architectural patterns to enhance software usability in [Juristo 2003].

Juristo and Lopez advocate a *forward engineering* approach to integrate UE into the SE process by suggesting the use of architectural patterns for software to enhance usability of the software product. The approach taken by these researchers differs from the traditional approach of measuring usability after the development of the software product. They consider usability as a quality attribute of a software system and address the problem of integrating usability characteristics into the general software architecture. The general UE quality attributes are too high to integrate into the software architecture. To solve this problem, Juristo and Lopez have decomposed the UE process into levels of abstraction progressively closer to SE. The two intermediate levels are *usability properties* and *usability pattern*s. The usability properties make software usable and usability patterns act as mechanisms to incorporate usability properties into software architecture. The usability patterns do not offer a software solution, but suggest an abstract mechanism to incorporate usability patterns into software architecture. The implementation of usability patterns into software architecture is therefore a problem.

Juristo and Lopez claim that architectural patterns will reflect a possible solution to the problem of implementation of usability patterns. The usability patterns are unique for every usability property. The architectural patterns are therefore the last link of the UE attribute-property-pattern chain connecting software usability with the software architecture. Juristo and Lopez claim that architectural patterns can be obtained using abstraction of patterns from particular designs where usability attributes have been transformed to software architecture. However, the architectural patterns thus generated have to be applied to development to test their feasibility.

2.6.4 Xavier Ferré

In his paper [Ferré 2003], Xavier Ferré states observations made by his research team working on the "STATUS" project. The aim of the STATUS project is to study and determine the connections between software architecture and the usability of the resultant software system. Ferré claims that the use of UE techniques is not straightforward, because they are not integrated into the SE process. Therefore, his research tries to introduce a handy group of increments, which when included in the software engineering process, would ensure higher usability of the resultant software.

Ferré presents a survey of the UE literature to identify activities that were best suited, and agreed upon by researchers, for inclusion in the software engineering process. To identify the best-suited activities, the researchers listed UE activities that enjoyed general acceptance in the HCI field. They also checked for activities that were *less alien to SE*, had *low integration costs,* and had a higher *general applicability*. The "findings" from the UE field were then mapped to the activities in the SE field, adapting their results to the SE concepts and terminology. Grouping of similar activities formed the *increments* or "deltas."

The UE-SE process formulated by Ferré and his team includes requirements analysis, interaction design, help design, and usability evaluation. The paper discusses in detail the phases of the integrated process and the usability increments applicable to the SE process.

The usability deltas provide techniques to cover all usability activities that can lead to improvement in the usability of the resultant software. However, one major factor that affects the eligibility of the SE process for integration of the UE deltas is its sequential nature, which Ferré claims, is an internal characteristic of the SE process. He also suggests that organizations need to evaluate whether the SE process they follow meets the minimum requirements for the incorporation of *all* or *some* of the UE increments.

In this chapter, we discussed the background information required to describe our research. We discussed the SE process, with emphasis on the Requirements Generation, and the Software Design phases of the process. We further discussed the UE process, describing the scenario-based design approach to UE. Finally, we also highlighted some important research that has been done to integrate the UE process into the SE process.

In the next chapter, we present the identification of interactions that ideally should exist among the activities of the UE and SE processes. The identification of interactions forms a major part our research presented in this thesis.

CHAPTER 3. FRAMEWORK OF INTERACTIONS

Both UE and SE processes focus on developing a “usable” and “functionally satisfactory” software product. A usable software product should be easily learn and adaptable. A functionally satisfactory software product possesses the necessary functionality to satisfy user requirements. The activities of the UE and SE processes focus on the same high-level goals, and, therefore, are similar in nature. The objectives of these activities and the techniques they use are, however, *significantly* dissimilar. For the UE and SE process activities to achieve their objectives, a substantial degree of information exchange should exist among activities of the UE and SE processes.

Currently, the UE and SE processes are practiced as being independent, non-interacting. The underlying framework that guides software development should motivate the exchange of information among activities of the UE and SE processes by employing the processes as *coordinated* and *synchronized* practices.

In this chapter, we discuss a *high-level* understanding of the exchange of information that ideally should exist among the different phases of the UE and SE processes. This exchange of information is useful in identifying essential interactions that must exist among the activities of the UE and SE processes. It is essential to note that while envisioning the exchange of information that should ideally exist among the UE and the SE processes, we are constrained by the current UE and SE process models that dictate the activities from these processes. We have used the pre-existing models for UE and SE to identify the activities of the UE and the SE processes and the goals and objectives of these activities.

We conceptualize the exchange of information among activities of the UE and SE processes as *interactions.* To identify the interactions, we search for relationships among objectives of activities and introduce the concept of *activity awareness,* which implies continuous interactions among the activities of the UE and SE processes to exchange design information. We represent this continuous exchange of information as *synchronizations* between the UE and SE activities. We discuss the interactions among the activities of the UE and RE processes in a detail and relate these detailed descriptions to the high-level interactions we present earlier.

3.1 Interactions between the SE and UE processes

In chapter 2, we present the UE and SE processes in detail. We identify the constituent activities of both life cycle processes and discuss the objectives of each activity. The identification of relationships among the objectives of these activities can be useful in identifying the exchange of information that should exist among them.

In this section, we present a high-level understanding of the exchange of information that should exist among the phases of the UE and SE processes.

3.1.1 Visualizing the high-level framework

Figure 3.1 on the next page depicts the interactions that should exist among the activities of the UE and SE processes. The figure shows a high-level identification of these interactions. A description of these interactions follows.

- *Project purpose and goals (Figure 3.1(a))*
- The usability engineers formulate a high-level vision and develop a rationale, called the *root concept*, for the software under development. They identify the purpose behind the development of the software product and the goals to be achieved. The software engineers, on the other hand, develop a conceptual overview of the system to understand the reasons behind the software development endeavor and identify the problems that the software product will be address. The interactions among the UE and SE activities can be represented at a high level as exchanges of information based on the understanding of the project purpose and goals. The SE activity of conceptual overview development can exchange information about the project purpose with the UE activity of project vision and root concept development. The SE problem identification activity can exchange information about project goals with the UE activity of project rationale development and root concept development.
- *Dependencies, constraints, and assumptions (Figure 3.1(b))*

  The root concept developed by the usability engineers includes the constraints that affect the system. The software engineers understand the constraints that affect the system through context and constraints analysis.

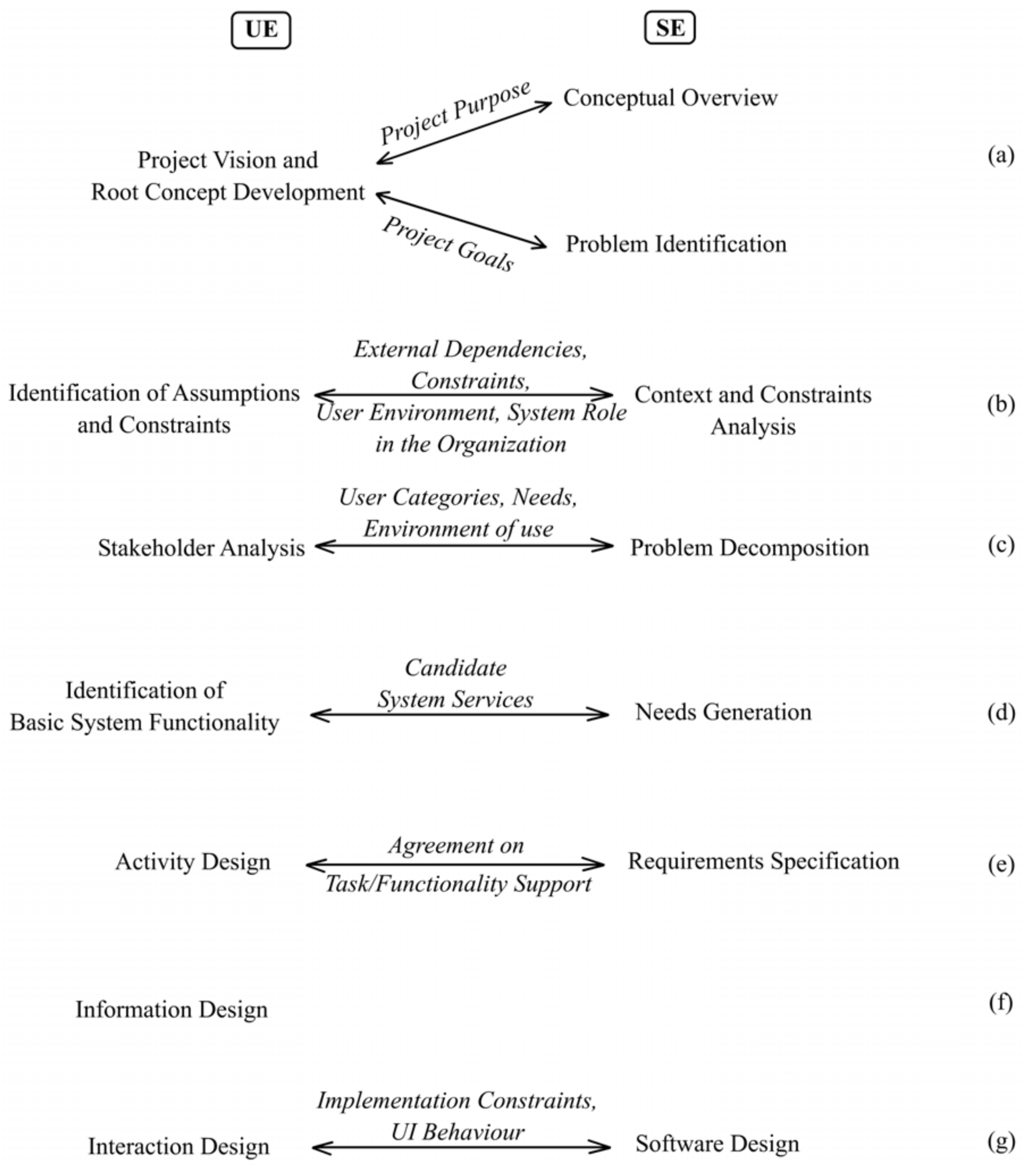
UE
SE
Project Vision and Root Concept Development
Project Purpose
Conceptual Overview
Project Goals
Problem Identification
(a)
Identification of Assumptions and Constraints
External Dependencies, Constraints, User Environment, System Role in the Organization
Context and Constraints Analysis
(b)
Stakeholder Analysis
User Categories, Needs, Environment of use
Problem Decomposition
(c)
Identification of Basic System Functionality
Candidate System Services
Needs Generation
(d)
Activity Design
Agreement on Task/Functionality Support
Requirements Specification
(e)
Information Design
(f)
Interaction Design
Implementation Constraints, UI Behaviour
Software Design
(g)

Figure 3.1 High-level understanding of interactions among the UE and SE process activities

The UE and SE processes should exchange information about the context of the system under development and the constraints imposed on the system.

- *Stakeholder categories and needs (Figure 3.1(c))*

  The software engineers perform problem decomposition to understand in detail the problems identified earlier. The usability engineers, on the other hand, perform field surveys for analysis of stakeholder characteristics and needs. The information collected by the UE stakeholder analysis and the SE problem decomposition activities should be exchanged between the UE and the SE processes. This exchange of information ensures a common understanding between the two teams about the user categories and needs and the context of the system.

- *Candidate services and activities (Figure 3.1(d))*

  The usability engineers design software systems in a usage context. Therefore, after performing stakeholder analysis and designing problem scenarios, they identify candidate activities that the system should support. The candidate activities address the problems depicted in the problem scenarios developed by the usability engineers. At the same time, the software engineers, in consultation with the customers, perform needs generation to generate a set of customer needs to ensure that customer needs are accurately captured. The usability and software engineers should exchange ideas about candidate system services identified during UE identification of basic functionality and the SE needs generation activities to ensure that both usability engineers and the software engineers have an understanding of the candidate system functionality identified by the other team.
- *Tasks and functionality (Figure 3.1(e))*

  The usability engineers, after identifying candidate activities, decide about the activities to be supported in the final system and design the details of those activities. On the other hand, the software engineers elicit system requirements from the users and verify and validate those requirements to reach the functional requirements specification. The interactions among the UE and SE activities during this stage should be based on agreement about the tasks and functionality to be supported by the system under development.
- *Information design (Figure 3.1(f))*

  The usability engineers define the presentation of information on the user interface during the information design phase. The definition of presentation of information, or information design, has little implications on the software engineering activities. Therefore, we include it as an important stage in the UE process, which does not interact with the software engineering activities.

- *Implementation constraints and UI behavior (Figure 3.1(g))*
  During the interface design stage, the usability engineers design the specifics of the interactive objects for the user interface and specify mechanisms in the user interface for manipulating and accessing task information; that is, they design the behavior of the user interface. The software engineers, during the software engineering design, translate the requirements generated into software design and select the architecture for the software system. Based on the architecture selected, they decompose the system into constituent components and define the interfaces among these components. They also define the external interfaces for the system and the interfaces of the system with human users. They design implementation-level details of the user interface, which the software engineers implement. Since not all user interface details designed by the usability engineers may be implementable by the software engineers, the usability engineers and the software engineers need to exchange information about the design and behavior of the user interface and the implementation constraints that may affect it.

3.1.2 Activity Awareness and Synchronization

Interactions among the usability and software engineering activities are required throughout the UE and SE requirements analysis and design phases. Activity awareness keeps both usability and software engineers aware of the particulars of the design being produced by the other team. To ensure activity awareness, the usability and software engineers should maintain continuous interactions throughout the requirements engineering and design phases. If one team feels the necessity of interaction to convey design specifications or constraints, the two teams can be schedule meetings. Figure 3.2 (a) depicts activity awareness among UE and SE activities.

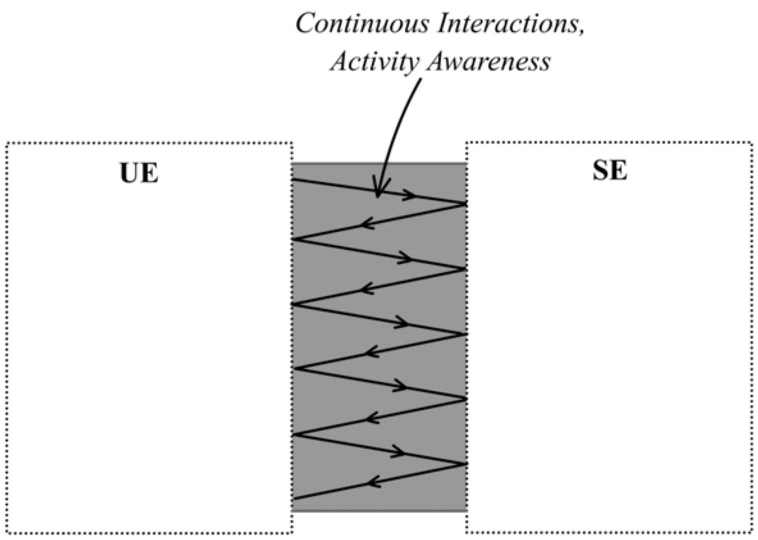

Figure 3.2 (a) High-level understanding of interactions among the UE and SE process activities

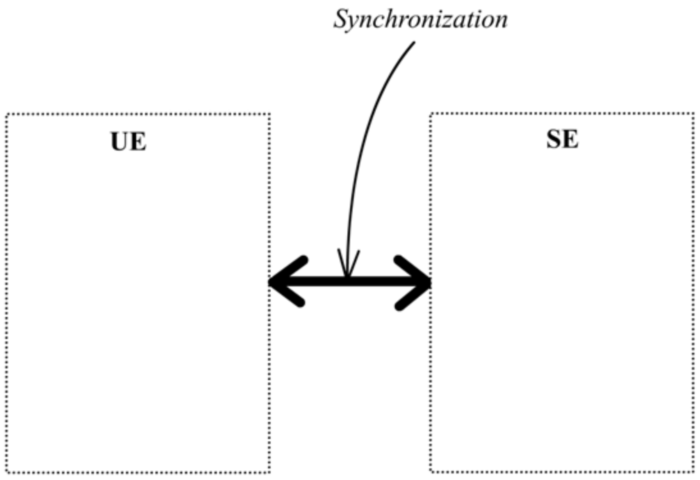

Figure 3.2 (b) Representation of Synchronizations

Figure 3.2 (b) shows *synchronization* among UE and SE activities. By synchronization, which is based on ideas and artifacts, we imply that the activities share an understanding of concurrent activities of the other process through continuous interaction and activity awareness. The ideas or artifacts that form the basis of the synchronization are documented with the synchronization. We employ the synchronization symbol to reflect continuous interaction and awareness.

In the following sections, we discuss the interactions among the activities of the UE and SE activities in detail. Although one can argue for a more detailed set of interactions among the UE and SE activities, we focus on identifying the major points of information flow rather than trying to capture the minutiae.

3.2 Vision and Overview Stage

Before the actual work on the software development project begins, the usability engineers try to gain a *high-level* understanding, called the root concept, of the problems to be addressed by the system to be developed. The software engineers, on the other hand, develop a conceptual overview of the system and formulate an understanding of the problem areas to be addressed by the software. We name this stage in the UE and SE processes as the "vision and overview" stage. This section identifies the interactions that ideally should exist between the UE and SE processes during the vision and overview stage.

3.2.1 High level understanding of the interactions

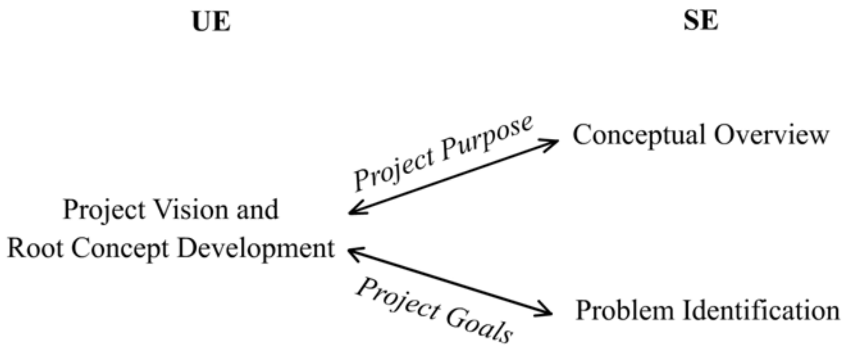

Figure 3.3 (a) High-level interactions among UE and SE activities during vision and overview stage

Figure 3.3 (a) depicts the first part of Figure 3.1 that includes synchronizations based on project purpose and project goals among UE and SE activities. We present details of the interactions among these activities in this sub-section.

3.2.2 The interactions in detail

The interactions among the activities of the UE and SE processes during the vision and overview stage are depicted in Figure 3.3 (b). The figure depicts synchronizations as well as transfer of information among the activities of the UE and SE processes. The remaining portion of this sub-section describes each of these interactions.

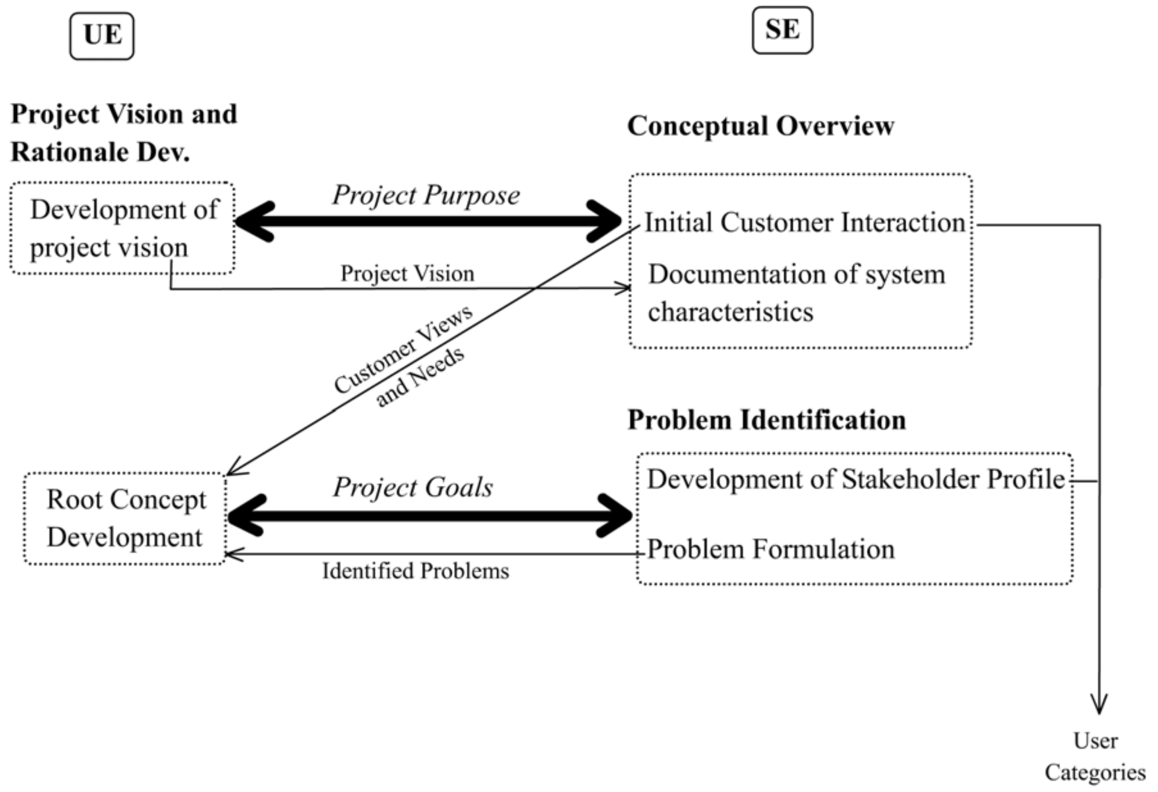

Figure 3.3 (b) Interactions among UE and SE activities during vision and overview stage

3.2.2.1 Transfer of information about the project vision

The usability engineers develop a high-level vision of the project before the high-level details of the project (root concept) are developed. This vision can be useful for the software engineers during initial interactions with customers. Information about the usability engineers’ project vision should be conveyed to the software engineers working on the development of a conceptual overview for the project.

3.2.2.2 Transfer of information about customer views and needs

The software engineers, during the initial customer interaction, collect information about the views of the customers regarding the system and about their needs that the system should satisfy. This information is essential during formulation of project goals and should be conveyed to the usability engineers.

3.2.2.3 Synchronization based on project goals

The root concept formulated by the usability engineers includes the high-level understanding the goals of the project. The software engineers formulate an understanding of the project goals during problem formulation. This understanding about the project goals ideally should be synchronized between the UE and the SE teams to ensure that both teams have a common understanding of the goals of the software project.

3.2.2.4 Transfer of information about identified problems

The usability engineers develop the root concept as an understanding of the project goals. The software engineers should, therefore, convey information about the problems they have identified which might be useful to the usability engineers while they develop the root concept.

3.2.2.5 Transfer of information about user categories

The software engineers collect information about user categories during initial meetings with the stakeholders. This information can potentially be useful to the usability engineers during their analysis of stakeholders and formation of user categories. Therefore, the software engineers should transfer information about user categories to the usability engineers.

3.2.2.6 Synchronization based on project purpose

During the vision and overview stage, the usability engineers develop a root concept for the project, which includes their understanding of the purpose of the project. The software engineers develop the same understanding through their initial interaction with the stakeholders (customers). The UE and the SE teams ideally should synchronize their understanding of the purpose of the project, which ensures that both teams have a common understanding with respect to the purpose of the software project.

After the initial vision and overview development, the usability and software engineers perform an analysis of the problem domain. In addition, they analyze the needs of stakeholders. We describe the activities related to these analyses in the next section.

3.3 Problem Analysis Stage

The usability engineers identify the constraints that affect the system as a part of the root concept by using field studies to perform stakeholder analysis. The software engineers analyze the problems identified using problem decomposition, and context and constraints analysis. We name this stage from both processes the “Problem Analysis” stage. In this section, we discuss the interactions that should exist between the UE and the SE problem analysis.

3.3.1 High-level understanding of the interactions

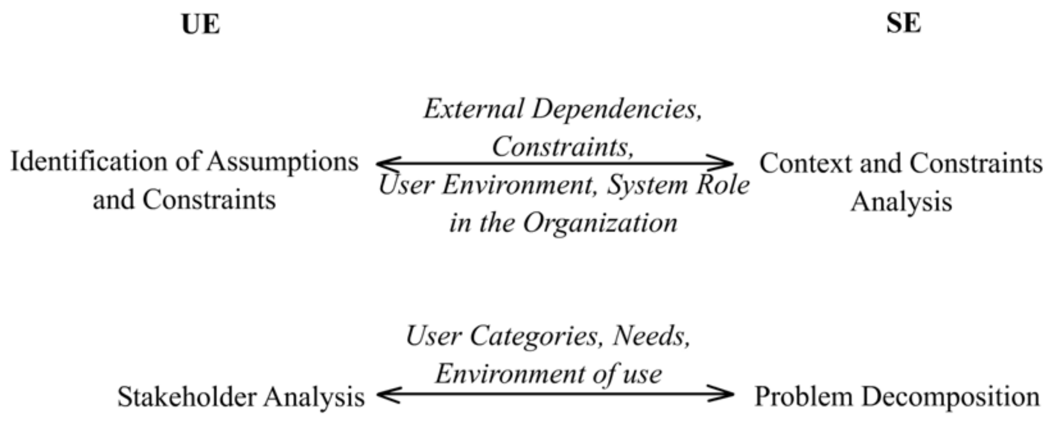
UE
SE
Identification of Assumptions and Constraints
External Dependencies, Constraints,
User Environment, System Role in the Organization
Context and Constraints Analysis
Stakeholder Analysis
User Categories, Needs, Environment of use
Problem Decomposition

Figure 3.4 (a) High-level interactions among UE and SE activities during problem analysis stage

Figure 3.4 (a) depicts synchronizations from Figure 3.1 which correspond to the problem analysis stage. These synchronizations ensure that the UE and SE teams have a common understanding of the system characteristics, user characteristics, and constraints that affect the system. We present the interactions between these UE and SE activities with higher detail in the next sub-section.

3.3.2 The interactions in detail

Figure 3.4 (b) depicts the synchronizations that ideally should exist among the activities of the UE and SE processes during the problem analysis stage. These synchronizations are discussed in detail in the following discussion. The discussion also

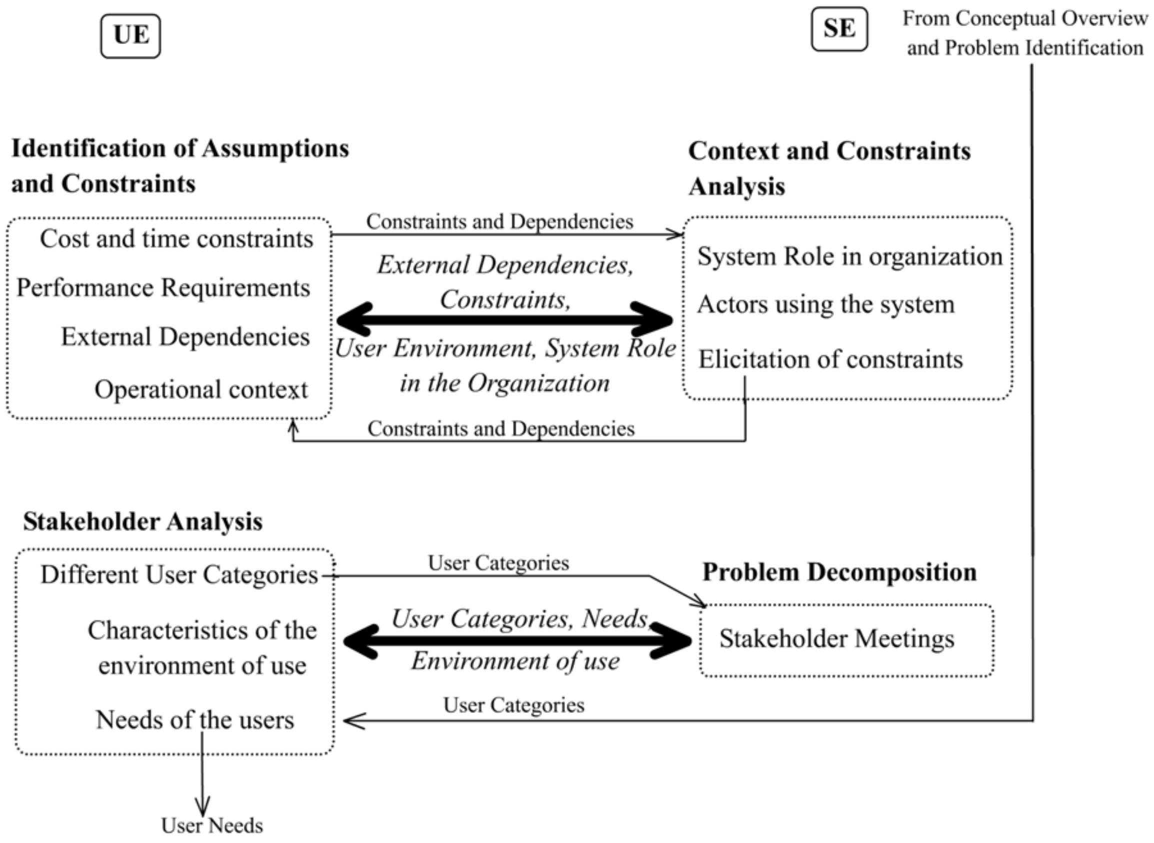
UE
SE
From Conceptual Overview and Problem Identification
Identification of Assumptions and Constraints
Cost and time constraints
Performance Requirements
External Dependencies
Operational context
Context and Constraints Analysis
System Role in organization
Actors using the system
Elicitation of constraints
Constraints and Dependencies
External Dependencies, Constraints,
User Environment, System Role in the Organization
Constraints and Dependencies
Stakeholder Analysis
Different User Categories
Characteristics of the environment of use
Needs of the users
User Categories
Problem Decomposition
Stakeholder Meetings
User Categories, Needs,
Environment of use
User Categories
User Needs

Figure 3.4 (b) Interactions among UE and SE activities during problem analysis stage

includes the transfer of information that should exist among the activities of the UE and SE processes during the problem analysis stage.

3.3.2.1 Synchronization based on context and constraints

The usability engineers include constraints based on cost, time, and performance requirements in their root concept. They also include constraints imposed by the operational context and other external dependencies. The root concept forms the basis of field surveys and stakeholder analysis.

The software engineers, on the other hand, perform an analysis of the context and constraints that are imposed on the system. During this activity, they identify the system role in the organization, the actors using the system, and the constraints imposed on the software system.

The usability and software engineers should synchronize their understanding of the external dependencies and constraints, user environment, and the role that the software system will play when deployed. This synchronization is crucial for the software development effort as it ensures that both the software and usability engineers share a common understanding of the constraints to be respected during software development.

3.3.2.2 Transfer of information about constraints and dependencies that affect the system

Both usability and software engineering teams identify the constraints and dependencies that affect the system. The constraints and dependencies identified by the usability engineers may, however, have a different focus from those identified by the software engineers. Information about constraints and dependencies identified by the other team should be useful to the usability and software engineers during the later stages of their design. Therefore, this information should be exchanged among the activities of the UE and SE processes.

3.3.2.3 Transfer of information about user categories

This knowledge about user categories should be synchronized between the usability and software engineers. The usability engineers identify user categories during stakeholder analysis, while the software engineers do the same during their initial interaction with the customers, earlier in the vision and overview stage. The exchange of information about user categories should help to maintain a common understanding between the two teams with respect to the user categories.

3.3.2.4 Transfer of information about user needs

The UE stakeholder analysis activity identifies the needs of the users that should be fulfilled by a usable system. These identified needs may be useful to the software engineers during their needs elicitation process. These needs are therefore transferred to the needs generation activities in the next stage of the development process.

3.3.2.5 Synchronization based on user categories and needs

The usability engineers perform stakeholder analysis to identify the different categories of users, needs of the users, and characteristics of the environment in which the system will be used. The software engineers, on the other hand, gain a deeper understanding of the problems identified earlier using stakeholder meetings as a means of problem decomposition and reinforce their understanding of the user categories that have been identified during initial stakeholder meetings. They also understand the needs of the users while decomposing the problems identified into smaller problem elements.

The usability and software engineers should synchronize this knowledge about user categories, the needs of users, and the characteristics of the deployment environment. This synchronization helps to maintain a common understanding between the two teams with respect to the above factors that influence the characteristics of the proposed software product.

After the identification of constraints that affect the system under development and the analysis of users, the usability and software engineers identify candidate services that the system should support. The activities of the UE and SE processes related to this identification of system services, and the interactions among them, are described in the next section.

3.4 Activity Identification

After the development of a high-level project vision and the analysis of the problems to be addressed, the usability and software engineers identify the basic functionality that the system should offer. The usability engineers envision the high-level activities that the system should support. The software engineers, on the other hand, identify the *needs* of users that the software system should satisfy. We call this stage the “Activity Identification” stage.

3.4.1 High Level Interactions during Activity Identification

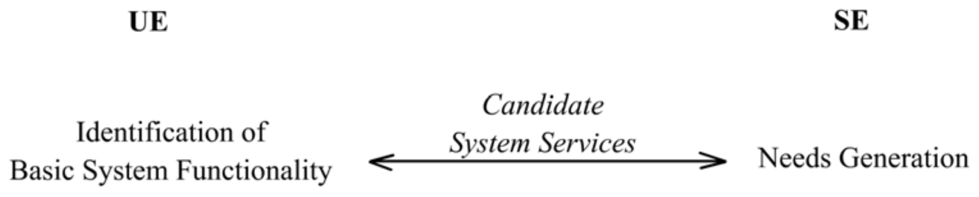
UE
SE
Identification of Basic System Functionality
Candidate System Services
Needs Generation

Figure 3.5 (a) High-level interactions among UE and SE activities during Activity identification stage

Figure 3.5 (a) depicts a part of Figure 3.1, which corresponds to the activity identification stage and depicts synchronization based on candidate system services to be offered in the new system.

3.4.2 The interactions in detail

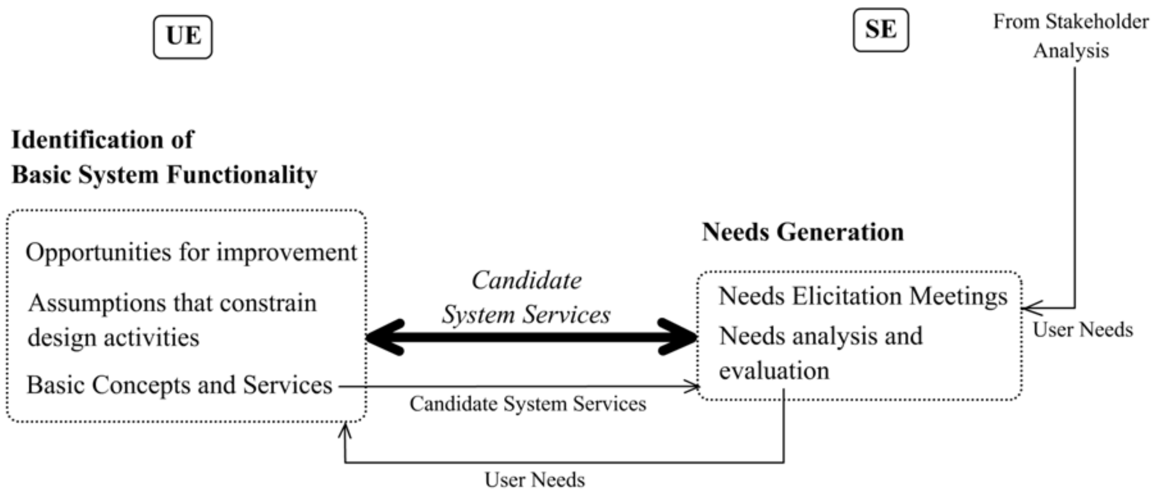
UE
SE
From Stakeholder Analysis
Identification of Basic System Functionality
Opportunities for improvement
Assumptions that constrain design activities
Basic Concepts and Services
Candidate System Services
Needs Generation
Needs Elicitation Meetings
Needs analysis and evaluation
User Needs
Candidate System Services
User Needs

Figure 3.5 (b) Interactions among UE and SE activities during Activity identification stage

Figure 3.5 (b) depicts the interactions among activities of the UE and SE processes during the activity identification stage. The interaction between the two processes during this stage is based on exchange of information about the system services that should be provided by the software system.

3.4.2.1 Transfer of information about candidate system services

During the Activity identification stage, the usability engineers identify the basic functionality to be offered by the software system. This basic system functionality is a high-level understanding of the activities to be supported by the system. The usability engineers analyze current practices to identify new opportunities for improvement and identify the assumptions that constrain design activities. The knowledge about opportunities for improvement and the constraints imposed on the design are used to identify the basic functionality offered by the software system under development. This information about these candidate activities to be supported by the system should be useful to the software engineers during their needs generation and should be conveyed to them.

3.4.2.2 Transfer of information about needs of the users

The software engineers identify the needs of the users through a needs generation process performed in consultation with the stakeholders. User needs generated by the usability engineers during stakeholder analysis are available for reference during the SE needs generation. The software engineers elicit customer needs through decomposition of problems identified during problem analysis into constituent problem elements. The problem elements are then organized and related to the context and constraints identified earlier. Finally, the software engineers perform needs evaluation to delimit the needs to be incorporated into the system from those to be left out or postponed. The identified needs should be conveyed to the usability engineers for use during the identification of candidate system services.

3.4.2.3 Synchronization based on ideas about candidate system services

The usability and software engineers should synchronize their ideas about the high-level understanding of the functionality to be provided by the software system to ensure that they share a common understanding.
After identifying and synchronizing their understanding about candidate system services, the usability and software engineers design the actual activities that will be supported by the system under development. The activities related to this detailed activity design and the interactions among them are discussed in the next section.

3.5 Detailed Activity Design

After identifying activities supported by the software system, the usability engineers design the details of these activities. The software engineers, on the other hand, derive software requirements that contain the functionality of the system and the activities they support. We name this stage in the design the “Detailed Activity Design.”

3.5.1 High Level Interactions during Detailed Activity Design

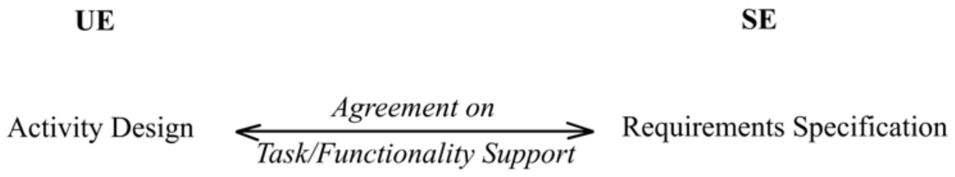

Figure 3.6 (a) High-level interactions among UE and SE activities during Detailed Activity Design

Figure 3.6 (a) depicts the part of Figure 3.1 that corresponds to the design of detailed activities. The interactions among the UE and SE activities during this stage of the design are based on an agreement of the tasks and functionality that the system will support.

3.5.2 The interactions in detail

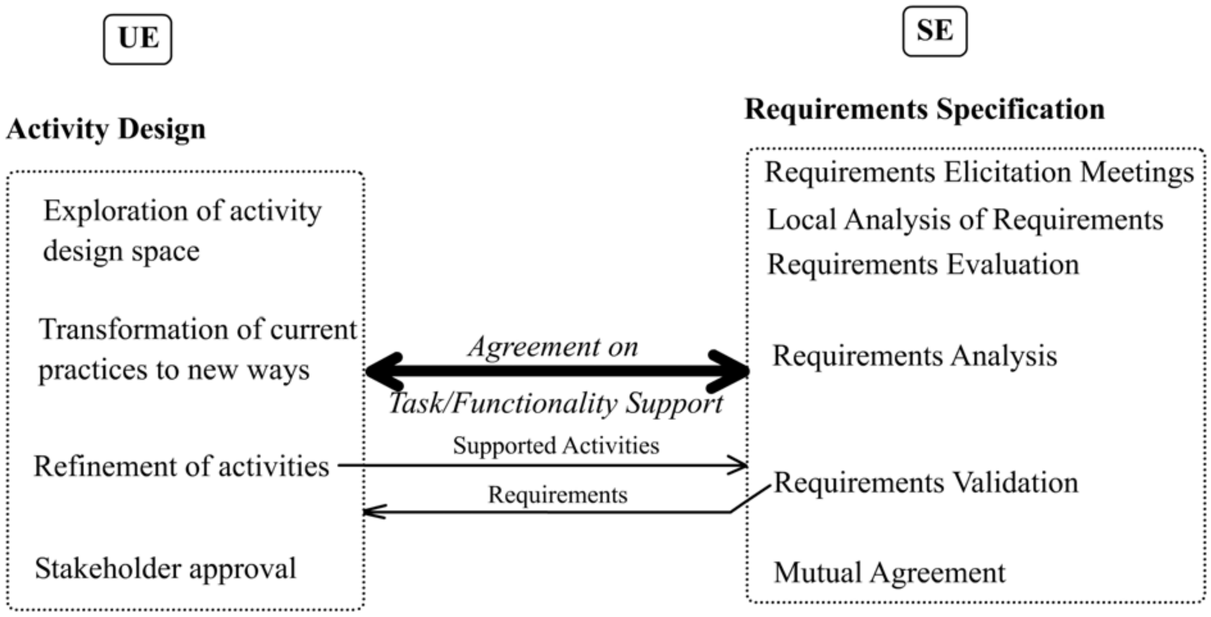
UE
SE
Activity Design
Exploration of activity design space
Transformation of current practices to new ways
Refinement of activities
Stakeholder approval
Requirements Specification
Requirements Elicitation Meetings
Local Analysis of Requirements
Requirements Evaluation
Requirements Analysis
Requirements Validation
Mutual Agreement
Agreement on
Task/Functionality Support
Supported Activities
Requirements

Figure 3.6 (b) Interactions among UE and SE activities during Detailed Activity Design

Figure 3.6 (b) depicts the different activities that the usability and software engineers perform during the Detailed Activity Design stage. It also depicts the synchronization among the usability and software engineers based on agreement of supported activities, and an interaction based on user views and requirements. We describe the information exchange below.

3.5.2.1 Transfer of information about supported activities

The usability engineers design the details of activities and perform a refinement of the design through stakeholder involvement. They explore the activity design space using metaphors and ideas based on technology to identify the details of activities supported. They also transform current practices to new activities by constructing alternate activity design scenarios and by evaluating design features using claims. The refinement of supported activities is performed using the point of view of system objects and the elaboration of activities through participatory design with stakeholders. The usability engineers verify the designed activities for coherence and completeness.

The software development effort requires the usability and software engineers to have a common understanding of supported activities. Therefore, promotion of a common understanding of supported activities between the two teams is necessary: the information about supported activities identified and refined by the usability engineers should be conveyed to the software engineers.

3.5.2.2 Transfer of information about user views and requirements

The software engineers generate software requirements from the user needs identified during needs generation. The software engineers elicit requirements from stakeholders, relative to quality characteristics, risk, effort, and priority. The requirements engineers then specify the requirements in a formal document called the Software Requirements Specification (SRS). Requirements validation ensures that the SRS conforms to standards and expected conditions.

The information about user views and needs collected by the software engineers during requirements elicitation meetings is deemed useful to and should be shared with the usability engineers. This synchronization is based on an agreement about supported activities. During synchronization, activities designed can be traced back to user needs based on the information about user views and needs made available.

3.5.2.3 Synchronization based on agreement on supported activities

The stakeholders need to approve the activities designed by the usability engineers and the requirements generated by the software engineers. The usability and software engineers, however, must synchronize their understanding of the tasks and functionality supported by the system before requesting stakeholder approval. The synchronization ensures that activities designed by the usability and software engineers do not conflict among themselves. Stakeholder approval should be requested only when the activities designed are verified for absence of conflicts.

The usability engineers design the user interface details, and the software engineers design the software after an agreement about supported activities is reached between the two teams. The following sections describe the interactions among the UE activities of the user interface design stages (Information, and Interaction design), and the SE activities performed during the SE design phase.

3.6 Information Design

The information design and the interaction design are two phases of the usability engineering process that deal with design of the interface. This section describes the information design phase.

The information design stage helps user interface designers arrange and represent objects and actions in a system in such a way that facilitates perception and understanding by the user. The designers aim to arrange visual information in the user interface in a way that helps the user to perceive, interpret and make sense of the information and thus help him to cross the *Norman's Gulf of Evaluation***.**

Figure 3.7 on the following page depicts the Information design phase of the UE process, which deals with properties of user interface elements with respect to layout, colors, fonts and font sizes used to display text, and so on. The activities of the information phase do not exhibit a direct relation to the activities on the software engineering side, but they affect the design of the interface. Therefore, we describe the information design phase but do not relate its activities to those from software engineering. The facts identified during the information design, however, are exchanged with the software engineers through the UE interaction design activity.

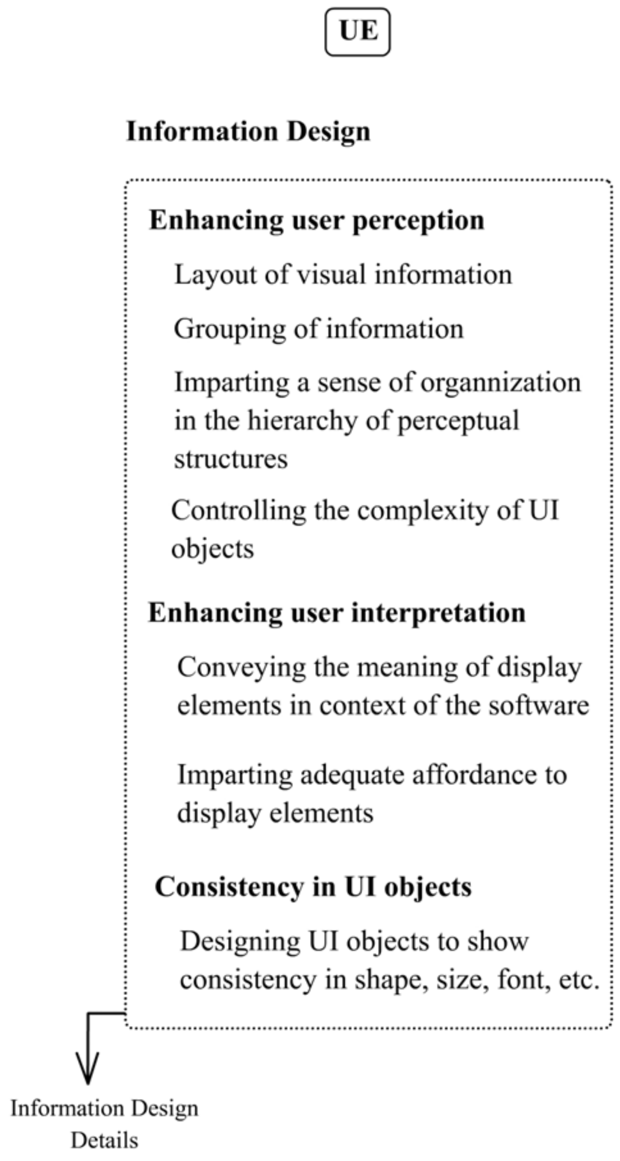
UE
Information Design
Enhancing user perception
Layout of visual information
Grouping of information
Imparting a sense of organnization in the hierarchy of perceptual structures
Controlling the complexity of UI objects
Enhancing user interpretation
Conveying the meaning of display elements in context of the software
Imparting adequate affordance to display elements
Consistency in UI objects
Designing UI objects to show consistency in shape, size, font, etc.
Information Design Details

Figure 3.7 The Information Design phase of Usability Engineering

The information design phase of the software engineering process includes the following major activities:

- *Enhancing user perception*

  The usability engineers design the layout of the visual information in the user interface to help user perception. While doing so, they group similar objects to impart a sense of organization in the interface. They also control the complexity of the user interface due to the presence of user interface objects.

- *Enhancing user interpretation*

The usability engineers design the visual information in the user interface to help the user interpret information. To accomplish this task, they design the user interface objects to convey the right meaning in the context of the software. They also design the interface objects to display characteristics that make their function obvious to the user. These perceptual characteristics guide the user in understanding the way to manipulate those interface objects.

- *Maintaining consistency in UI objects*

  The usability engineers design the user interface objects to be consistent with the rest of the interface and with other objects in the interface. Similar objects are designed to have similar display characteristics. The consistency in the user interface significantly helps user interpretation of information presented in the interface.

As mentioned previously, the activities of the information design stage do not directly influence the activities of the software engineering process. It should be noted though that the information design stage is an important stage in the user interface design and provides user interface design information to the interaction design, which is the next stage in the user interface design process.

3.7 Design of Interface details

The usability engineers design the interactive components in the user interface provided by a software system. These specifications users bridge the *Norman's gulf of Execution*. The software engineers design the software components of the system. They decide on the system architecture, decompose the system into components, and design the components.

Though the design of the user interface is specified by the usability engineers, functionality related to the display of the user interface is designed by the software engineers. This functionality is spread across the different components of the software. We name this stage in the software development as "Design of Interface details."

3.7.1 High Level Interactions

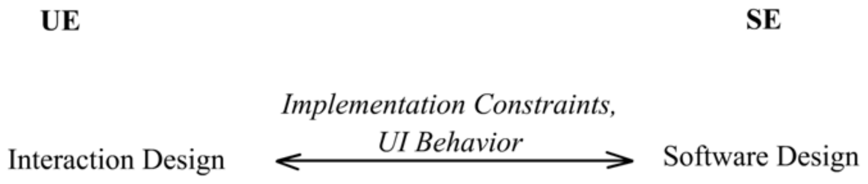

Figure 3.8 (a) High-level interactions among UE and SE activities during Interface Design stage

Figure 3.8 (a) depicts the high-level interactions between the UE and SE processes during the Interface design stage. The UE interaction design and the SE design activities synchronize their designs based on the UI behavior expected and the implementation constraints that affect the design. The interactions among the activities of this stage are described in the remainder of this section.

3.7.2 The interactions in detail

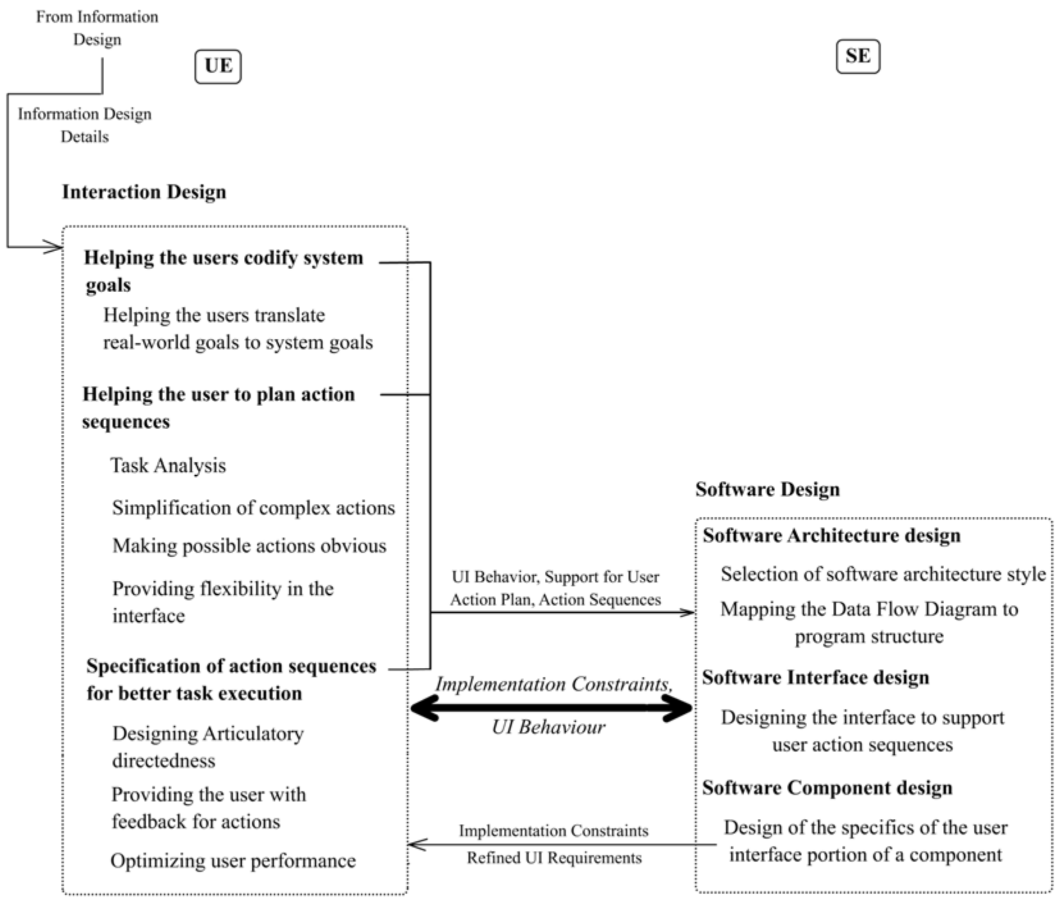
From Information Design
UE
SE
Information Design Details
Interaction Design
Helping the users codify system goals
Helping the users translate real-world goals to system goals
Helping the user to plan action sequences
Task Analysis
Simplification of complex actions
Making possible actions obvious
Providing flexibility in the interface
Specification of action sequences for better task execution
Designing Articulatory directedness
Providing the user with feedback for actions
Optimizing user performance
UI Behavior, Support for User Action Plan, Action Sequences
Implementation Constraints,
UI Behaviour
Implementation Constraints
Refined UI Requirements
Software Design
Software Architecture design
Selection of software architecture style
Mapping the Data Flow Diagram to program structure
Software Interface design
Designing the interface to support user action sequences
Software Component design
Design of the specifics of the user interface portion of a component

Figure 3.8 (b) Interactions among UE and SE activities during the Interaction Design stage

Figure 3.8 (b) depicts the interactions that should exist among the activities of the UE and SE processes during the interface design stage. These interactions are discussed in this sub-section.

3.7.2.1 Transfer of information about interface support for the user action plan, action sequences and UI behavior

During the interaction design stage, the usability engineers design the interface to meet the following goals:

- *Helping the users codify system goals*

  The usability engineers design the interactive objects in the interface to help the users translate real-world goals to system goals. Semantic directedness, which implies matching of real world goals, like clicking of a button, to the system goals, is used while creating interface objects. Semantic directedness helps users codify system goals.
- *Helping the user to plan action sequences*

  The usability engineers plan the user action plan that comprises the steps needed to achieve system goals. The user action plan is defined using a hierarchical analysis of user tasks supported by the system. The usability engineers try to understand the mental models of users and design system information to help users make inferences. The use of mental models of users makes user actions obvious. The usability engineers organize interrelated information in the interface to help simplify the complex actions for the users. The usability engineers also design the interface for flexibility to give the users a feeling of control.
- *Specifying action sequences for better task execution*

  The usability engineers design the system to have articulatory directedness, or mapping of physical movement in the interface to system tasks. They provide the users adequate feedback for their actions. The usability engineers design the interface to optimize user performance by managing the tradeoff between power and ease of use, providing keyboard shortcuts and offering good defaults.

The user interface of the software system must support the user action plan developed by the usability engineers. The usability engineers, therefore, must convey to the software engineers the details about the user action plan, the action sequences that need support, and the behavior of the user interface.

3.7.2.2 Transfer of information about implementation constraints and refined UI requirements

During the software design, the software engineers perform the following activities:

- *Software Architecture design*

  The software engineers select software architecture for use in the development of the system. They create entity relationship diagrams and data flow diagrams to help them understand the relationships between the entities in the software and also to understand the flow of data in the system. The software engineers also use state transition diagrams to represent the states the system encounters during operation. The understanding of the system achieved is used in the decomposition of the software system into components.

- *Software Interface design*

  The software engineers define three major types of interfaces during the definition of interfaces: the interfaces between the components of the software system, the interfaces of the software system with other interoperating software, and the user interface that is offered by the components of the software system. During the user interface design, the software engineers define the interface to follow the user action paths suggested by the usability engineers.

- *Software Component design*

  The software engineers design the details about the internal structure of components during the component design activity in addition to designing the specifics of the user interface provided by the software components.

The software engineers may make several refinements to the user interface as a part of the software engineering design. The interface defined as a part of the software interface can undergo several design refinements during component design. Because the usability engineers must abide by implementation constraints evolved through the above process, the software engineers should convey to the usability engineers information about the evolved implementation constraints and refinements to the user interface.

3.7.2.3 Synchronization based on the UI behavior and implementation constraints

When both UE and SE teams complete their designs, the specifications generated must be synchronized to ensure that the usability engineers have designed the user interface within the implementation constraints. This synchronization ensures that the design of the software system follows the user action plan designed by the usability engineers. It also ensures that the interface design embedded into the software components by the software engineers matches the user interface behavior suggested by the usability engineers.

In this chapter, we discuss the interactions that ideally should exist among the activities of the SE and UE processes. To ensure a common understanding of critical information among activities of the UE and SE processes, we introduce synchronizations.

We recognize that for a development effort to have parallel, coordinated SE and UE efforts, temporal coordination among the interacting activities of these two processes is also crucial. Coordination and concurrency among UE and SE activities are important for the coordinated development process that incorporates the UE and SE processes. In the next chapter, we outline a temporal coordination scheme devised for the UE and SE processes. That scheme uses the synchronization activities and synchronization boundaries as groundwork for attaining temporal coordination.

CHAPTER 4. DEVELOPMENT OF A PROCESS MODEL BASED

In this chapter, we investigate two standard software engineering models of development with the intent of using them as the basis for outlining a coordinated development process that incorporates UE and SE activities. We study the incremental development model and the spiral model for software development in relation to the needs of the coordinated development process. We also discuss issues that need to be resolved in order to reach a coordinated development process. Additionally, we discuss the applicability of the incremental development model in detail and provide an overview of the issues evident in the application of the spiral model to the coordinated development process.

4.1 The varied multiplicity of relationships among the UE iterations and SE increments

The SBD based usability engineering process is iterative in nature. The iterations are based on the observations of the prototyping stage, as depicted in the diagram representing the stages of the SBD process for usability engineering (Figure 2.3). Software design and implementation, on the other hand, can be effectively performed in the form of increments that build upon the previous ones [Cockburn 1995]. Use of increments during development of software does not necessarily imply the use of the Incremental Model, but implies the ideology behind development of evolving systems.

In a coordinated development framework, the iterative UE processes and the incremental SE processes should be concurrent and interactive. The *correspondence* relationships among the UE iterations and the SE increments can be one-to-one, one–to-many, or many-to-one. By this statement, we imply that one, or more than one, UE iteration can correspond (address related concerns) to the development of one increment of the software product. At the same time, a usability iteration may relate to more than one increment from the software engineering process. Therefore, we state that the *multiplicity* of the correspondence relationships among the UE iterations and SE increments can vary. The coordination and synchronization efforts between the UE and SE processes should consider the variance in multiplicity of correspondence.

As discussed in the last chapter, sets of activities of the UE and SE processes need to be concurrent to ensure proper exchange of information among these activities. With a variance in multiplicity, to enforce this concurrency, a one- to- one relationship must be enforced between UE iterations and SE design increments. This imposition makes coordination between the two processes somewhat simpler in nature. However, the solution in the form of this imposition does not conform to the requirement of attaining the maximum possible overlap among the UE and SE activities.

This chapter highlights the various issues, such as the one discussed above, we encounter while developing a *practicable* coordination and synchronization strategy between the UE and SE processes.

4.2 Application of the Incremental Model of software development to the coordinated development process

The Incremental Model for software engineering has a few practical disadvantages when employed in a development framework that incorporates UE and SE processes as separate, but coordinated, efforts. It needs to be modified slightly to make it suitable for a software project that follows the coordinated development framework. This section describes the advantages and shortcomings of the model and presents the modified Incremental Model that meets the needs of the coordinated software development framework.

4.2.1 A detailed investigation of the Incremental Model for software development

Incremental analysis and design and the Incremental Model for software development follow Booch’s philosophy of a *microcycle* [Martin, R.C. 1999] [Booch 1994], which implies “analyze a little, design a little, and code a little.” Figure 4.1 depicts the Incremental Model and shows the development of software in increments.

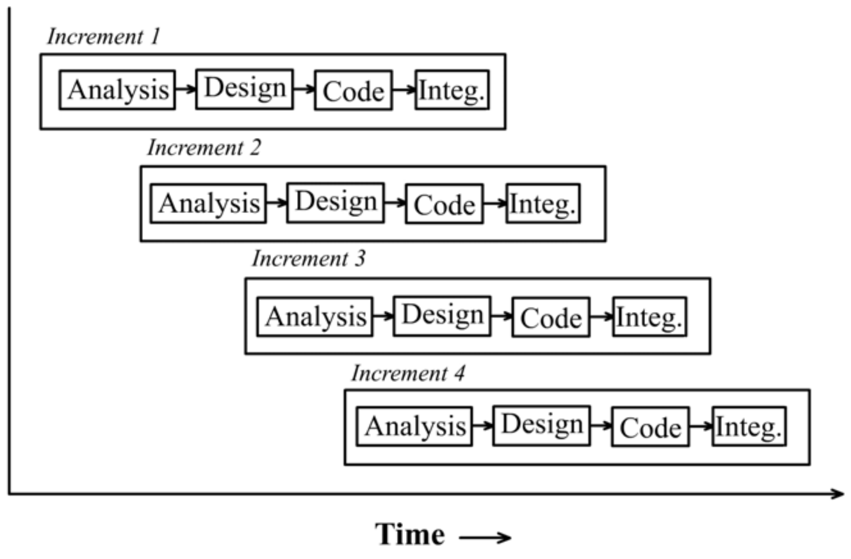

Figure 4.1 Incremental Development Model

Figure 4.1 shows software development in increments. Each increment adds new functionality to the existing software as well as support for features that can be added in future increments. Each increment contains all the principal stages of a software development process. Analysis of requirements, design of software, implementation (coding), and testing are performed with each increment. The Incremental model encourages overlap between consecutive increments to reduce the time delay between consecutive releases of the software product. As depicted in Figure 4.1, the overlap can extend up to the analysis stage of the earlier increment. Commercial software development efforts largely employ the Incremental model for the following reasons:

- *Incremental design lessens project complexity.*

  The software engineers can avoid high complexity of the design introduced by a large upfront analysis and design phase with the use of incremental analysis and design. They split the development effort into increments such that each increment is not a subsystem, but cuts across as much functionality as possible. This division is known as *vertical* slicing [Martin, R.C. 1999]. Slices, or increments, represent features that can be added to the system in increments. The software engineers can eliminate features to meet the schedule, if required.
- *Incremental design emphasizes finding obvious concepts, deferring deep investigation.*

The incremental analysis and design process emphasizes finding obvious concepts in the form of features that the software engineers can build into the software system. The software engineers build these features over the core of the software, which they develop in the initial increments. The software engineers delay in-depth investigation of these features but build the necessary software support for these features in the core of the software. Incremental development allows construction and implementation of software to start early. The early construction gives important feedback to the designers and helps improve subsequent specifications.

- *Incremental design allows adoption of changing requirements.*

  With the use of the Incremental Model, the software engineers can accommodate changing requirements into the system with relative ease. We note that change in requirements always contributes to rework and, therefore, an overhead. The Incremental Model reduces the overhead by allowing the software developers to incorporate changes in requirements at an increment level. The overhead incurred is reduced but not eliminated by the use of the incremental development model. A formal change request procedure that investigates the impacts of the change is, therefore, a necessity even when using the incremental development methodology.

- *Incremental development lessens the complexity of product testing.*

  With the use of incremental development, the software engineers can test the software as they build it. Frequent testing provides continuous, concrete feedback to the developers. The software engineers integrate the additional functionality into the software product as they implement it. Incremental development, therefore, lessens the *big bang* integration problems incurred during product integration that integrates several components into a working system. Testing the functionality of the complete product after integration of additional functionality becomes simpler due to incremental integration and testing.

- *Incremental development allows continuous integration and early, frequent product releases.*

The development teams can release software versions early and often, which works as a positive for the software business of the development organization. Moreover, the system is never a few days from release. The developers can build functionality into the software as prescribed by the business and provide patches to upgrade the older versions to the latest release.

Barry Boehm [Boehm 1988], in his listing of top 10 risk items and remedies, states incremental development as a remedy for the risk introduced by unrealistic schedules and budgets. Boehm also mentions the Incremental Model as a development strategy to follow in an environment with continuous streams of changes to requirements. The developers deliver increments to the customers for use and allow users to participate in development and testing early on. The software engineers evaluate project risks and develop mitigation strategies at each stage of the design in incremental development. This risk identification and mitigation is highly essential for project success.

In the next subsection, we describe the issues involved in employing the incremental development model in the design of the coordinated development process. We also suggest a modification to the Incremental Model to make it more applicable in the coordinated development process. We have derived this modifications from [Martin, R.C. 1999], which addresses the necessary improvements in the incremental development model.

4.2.2 The Modified Incremental development model

The Incremental Model supports change of requirements but the overhead incurred due to the change, though highly reduced, is substantial. Therefore, one of the most important issues to resolve in order to make the model suitable for the coordinated development effort is the lack of support for change in requirements.

In the initial stages of development, a large amount of interactions, exist among the activities of the usability and software engineering processes. Therefore, we expect a large number of changes in requirements to occur during the initial stages of a coordinated development effort. In order to be suitable for a coordinated development effort, the development model has to minimize the overhead due to change in requirements. The development model can minimize this overhead by better accommodating change in requirements. To improve the change handling capabilities of the Incremental Model, we suggest the following modification.

While developing software using the incremental development model, the high-risk increments, which represent some of the most important features of the software, should ideally be the first ones to be addressed [Cockburn 1995]. The most risky increment should be the first to be developed. The software developers should design and develop the high-risk increments as soon as possible. In order to minimize the overhead due to change of requirements in the initial stages of the design of these high-risk increments, we propose the development of the first two or three increments serially. The two advantages of following this approach are:

1. The changes in requirements are easier to implement when the software engineers develop the first few increments sequentially. This is because changes in the requirements require modification of earlier increments but do not initiate a change cascade into future increments that are already under development. However, the sequential arrangement of the first few increments leads to a longer project schedule. It is important to note that this arrangement is a tradeoff reached between the overhead incurred due to rework and overhead incurred due to a slightly larger project schedule. We suggest the sequential arrangement of the initial increments because an overhead due to rework also causes schedule slippage. The software engineers cannot easily estimate this slippage in advance and, therefore, introduce a large risk factor therefore.
2. The development team learns much from the first few increments. The added knowledge gives higher productivity, and the team can save a substantial amount of time in the later parts of the project.

In the next sub-section, we describe the coordination among the UE iterations and the SE increments. We present the relation between a UE iteration and SE increment with respect to the change of requirements.

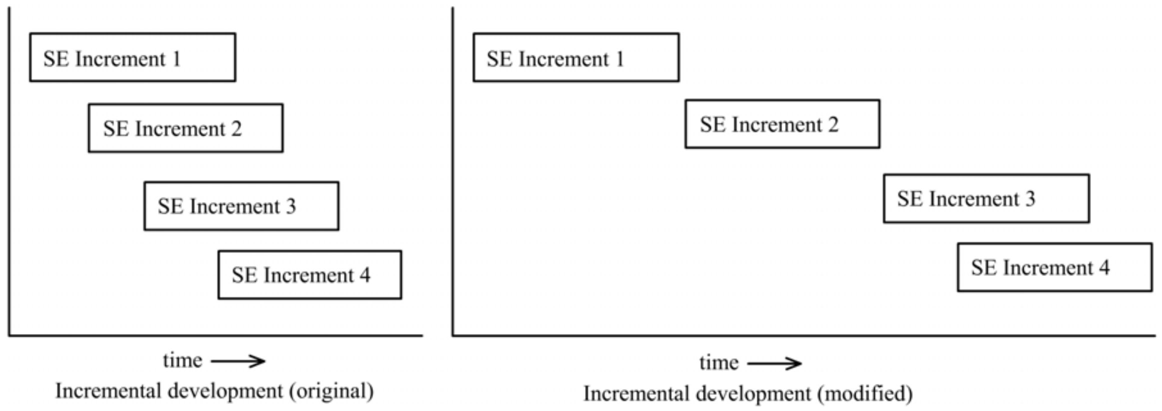

Figure 4.2 The original and modified incremental development model

We also discuss issues related to temporal sequencing that arise due to the variance in multiplicity of the correspondence relationship among UE iterations and SE increments.

4.2.3 Coordinating the UE process iterations with the SE increments

In section 4.1, we introduced the problem caused by the variance in multiplicity among the UE iterations and the SE increments. To resolve the issues introduced by the variance in multiplicity and to ensure synchronization between the UE and SE processes, we take an alternate approach to modeling the interactions among the UE and SE activities. The exchange of information among the activities of the UE and SE processes may introduce *changes* in the requirements or design already produced. We model these changes to promote exchange of information among UE iterations and SE increments with varied multiplicity.

4.2.3.1 Modeling interactions as change in requirements and design

In this subsection, we examine how interacting UE and SE activities can influence one another, and identify the types of changes these influences can introduce in the requirements or design produced by the activities. It is necessary to ensure that the UE iterations and SE increments can accommodate change in requirements with minimal overhead.

The usability engineering process is iterative; therefore, it is simpler for the usability engineers to accommodate change initiated by the software engineering activities. It is relatively difficult for the software engineering increment to accommodate change. We categorize change in requirements into two distinct categories: addition of requirements and change in requirements that already exist:

1. *Addition of requirements*

   The incremental development process supports addition of requirements and, hence the incremental addition of functionality. Accommodation of additional requirements therefore is simple. There may be a case when an additional requirement is imposed on a part of a system that is already implemented. The additional requirements may cause change in some other pre-existing requirements. The issue then moves to the category of changing requirements.

2. *Changes in requirements*

   Existing requirements may need to undergo changes. These changes in requirements may be initiated by several reasons, such as change in the operating environment of the system, change required in the behavior of the system, or additional requirements imposed that in turn require a modification of the existing requirements. The Incremental Model permits change in requirements by accommodating them in future increments. The Incremental Model can also handle a continuous stream of changing requirements with an overhead of rework.

   With incremental development, the software developers need to implement any change required in earlier increments. If the change substantially impacts the system, substantial rework may be required, or the change may be too expensive to implement. A configuration control system, therefore, becomes essential to control change in requirements.

4.2.3.2 Mitigating the variation in multiplicity

As stated earlier, the correspondence relationship among UE and SE increments may be one to one, one to many or many to one. We investigate each of these cases in this subsection with respect to support for change in requirements and design. We highlight the issues that are evident and need to be resolved in order to reach a coordinated development effort using the Incremental Model on the software engineering side.

1. One-to-one relationship

   A one-to-one relationship makes the designing of a coordinated development framework relatively simple. We have defined the interactions that ideally should exist among the SE and UE activities. One complete usability cycle discussed earlier corresponds to a single UE iteration while one complete software engineering cycle corresponds to a single SE increment. A one-to-one relationship is an ideal case, however, may not exist throughout the development process.

2. One-to-many relationships

   In one-to-many relationships, one usability engineering cycle corresponds to several increments from software engineering. This case would also be a rare occurrence because the usability engineering process is faster and highly iterative in nature. Additionally, the usability engineers can produce low fidelity prototypes in initial stages of the design to save time. Therefore, the usability engineering iterations are expected to take less time than software engineering increments, which involve implementation, integration, and testing at every increment.

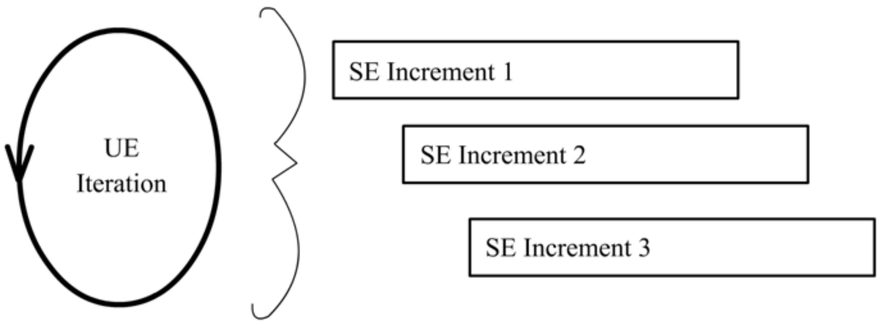

Figure 4.3 One–to-many Relationship

The following issues are evident and need to be resolved to design a coordinated development effort with one to many relations among the UE iterations and the SE increments. We expect that an effort to resolve these issues will encounter several other issues to resolve.

- *Matching of synchronizations among UE iterations and SE increments*

  Both UE and SE processes need to coordinate with each other and synchronize based on the various ideas as discussed in Chapter 3. The synchronizations among

UE and SE activities require sets of related UE and SE activities to be performed concurrently. This requirement ensures that the usability and software engineers share a common understanding about the output of corresponding sets of activities from the UE and SE processes. At the same time, this requirement also disallows a faster UE or SE process from proceeding to the next set of activities. Therefore, even though the synchronizations keep the UE and SE processes synchronized with each other, they may cause a schedule overhead in a one-to-many multiplicity relationship.

- *Transfer of synchronization information to other SE increments*

  The software engineers may work on several increments simultaneously as the SE increments need not be sequential and may be staggered in time. Two major issues arise due to the staggered nature of these increments. The first issue is the selection of the SE increment (among the several simultaneous increments) that will synchronize with the UE iteration. The second issue is the actual communication of synchronization information among the SE increments themselves, after one of these has synchronized with the UE iteration.
- *The transfer of information in the form of interactions that ideally should exist among the UE and SE process activities*

  The transfer of information among UE and SE activities requires the design of artifacts that the teams can create at different points during the coordinated development process. The design of these artifacts present several issues One important issue is designing the artifacts to support the *addition* of information. This is required because several SE increments may execute one activity and produce information that should be included in the artifact. This additional information has to be included in the artifact before the UE process collects the artifact for consideration of the information included in it.

  The usability engineers need to design similar artifacts for the UE process to transfer information to the SE process. The software engineers may work on several increments simultaneously as the SE increments may be staggered in time. The usability engineers cannot supply the artifact designed to all the SE increments. Therefore, the issue here is to recognize the SE increment which should handle the usability artifact.

Finally, a mechanism must be in place that notifies the UE (SE) component that a modification to a SE (UE) artifact has occurred.

3. Many-to-one relationships

   With a many-to-one relationship, numerous usability engineering iterations correspond to a single increment from software engineering. Most coordinated development efforts could fall under this relationship given the nature of the usability engineering iterations and software engineering increments.

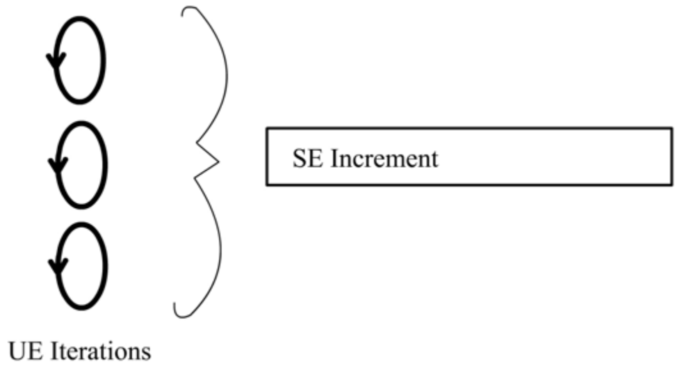

Figure 4.4 Many-to one Relationship

The issues in the many- to one- relationships are similar to those found in the one-to-many relationships.

- *Matching of synchronization activities and transfer of synchronization information*

  This issue is similar to the issue we encountered in one to many relationships. The UE or SE process, whichever is faster, is slowed down by the necessary exchange of information, and the changing requirements introduced by the other process.

- *The transfer of information in the form of interactions that should ideally exist among the UE and SE activities*

  This issue is similar to the issue we encountered in one- to- many relationships. Transfer of information among the UE and SE activities would require the design of specialized artifacts. With a many-to-one relationship, the issue about selection of one of the parallel increments is not substantial because initially the UE iterations are not performed staggered in time.

These are the important issues that need to be resolved in order to reach a coordinated development process that incorporates the UE and SE activities as coordinated, but separate, efforts using the software engineering Incremental Model. In the next section, we investigate the use of the spiral model of development as the software engineering model and discuss the issues that need to be resolved to reach a coordinated development process using that for software engineering model.

4.3 Application of the spiral model of software development to the coordinated development process

In this section, we present an overview of the spiral model for software engineering and study its applicability to a coordinated software development effort that incorporates UE and SE processes. We provide an overview of the issues that can be encountered when modeling the coordinated development process using the spiral model. We base this overview on issues that may be encountered while modeling with the Incremental Model.

4.3.1 An investigation of the Spiral Model to establish a relationship with the SE model used to design the UE-SE interactions.

Figure 4.5 [adapted from [Boehm 1988]] depicts the spiral model. The spiral model takes a *risk-driven* approach to the software process, rather than a document or code driven approach [Boehm 1988]. The radial dimension (Figure 4.5) shows the cumulative cost incurred in accomplishing the steps to date, and the angular dimension represents the progress made in completing each cycle of the spiral. Each cycle begins with objective analysis, alternative means identification, and constraints identification. The next step is to evaluate the alternatives relative to the objectives and constraints.

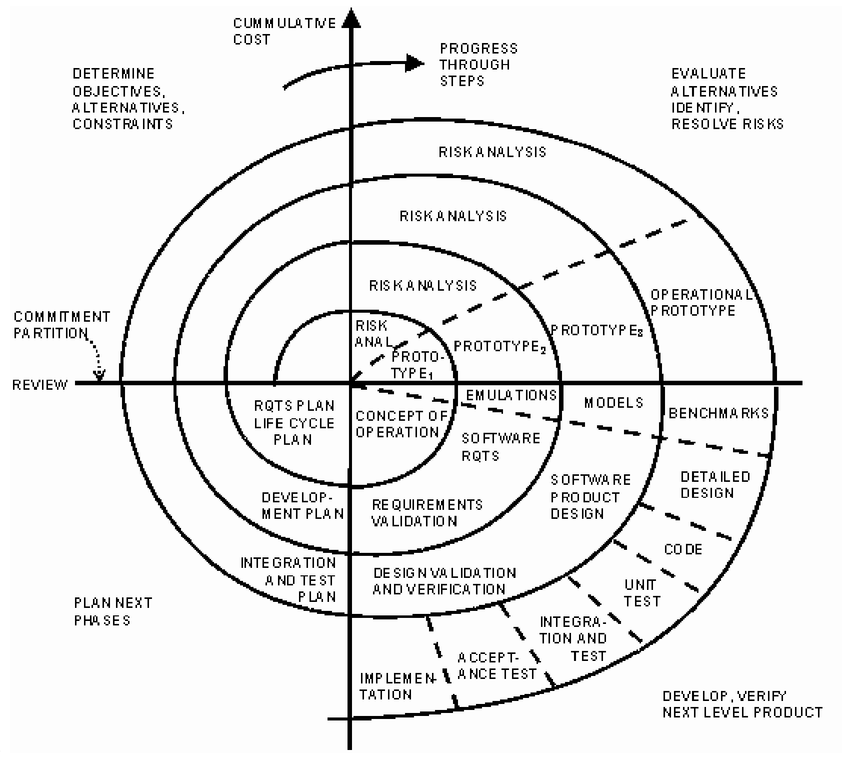

Figure 4.5 Boehm's Spiral Model [Adapted from Boehm 1988]

This process can identify areas of uncertainty and risk. If risks are identified, the next step should be a cost effective resolution of the risks. Prototyping, simulations, modeling, and benchmarking are effective strategies. Evolutionary prototyping could be practiced until the performance or user interface risks are mitigated. In case of program development or internal interface control risks, the next steps follow the basic waterfall approach, modified to incorporate incremental development. Plans for the next cycle are developed before the end of the current cycle. Each cycle reaches completion with a review that involves primary stakeholders and covers all products developed during the previous cycle. The review includes plans for the next cycle and ensures that the stakeholders are mutually committed to the plan. Plans for successive phases might partition the product into increments for successive development of components. We can visualize a series of parallel spirals, one for each component, adding a third dimension to the Figure 4.5.

To use the spiral model for modeling the SE activities in a coordinated development process, we need to relate the spiral model activities and the activities that are a part of the incremental SE process defined earlier. A simple overview of the spiral model does not establish this relationship. The spiral model would not support the interactions and synchronizations with the concurrent UE process unless it conforms to the pattern of SE activities used to model the interactions.

We did not explicitly investigate this relationship while investigating the Incremental Model because it is largely based on the standard Waterfall Model. The internal construction of each increment shows waterfall modeling with requirements analysis, software design, implementation, and integration and testing. Therefore, the Incremental Model conforms to the software engineering process used to model the interactions between the UE and SE processes.

In order to establish the relationship between the spiral model and the SE model used to define the interactions, we unwind the spiral to understand the sequence of activities it suggests. Figure 4.6 shows the unwound spiral.

From the unwound spiral, we observe that:

1. The spiral model has an objective analysis, alternative means identification, and constraints identification stage (O/A/C) at the beginning of every phase during software development.
2. The spiral model supports risk analysis through prototyping as a part of each stage in the software development process.
3. The spiral model includes simulation, modeling, and benchmarks at each stage during software development.

The actual set of activities at each stage is the set of activities after the O/A/C, risk analysis, prototyping, and simulations. We use this actual set of activities to establish a relationship between the spiral model and the software engineering process used to define the interactions between the UE and SE processes.

The following activities from the spiral model correspond to the activities from the SE process used earlier and help establish a relationship between the two:

1. *Con-Ops, Requirements, and Life cycle plans.*

   This stage in the spiral model includes obtaining a conceptual overview from the concept of operations, establishing a requirements plan, and planning the life cycle of the project. The activities in this stage strongly correspond to the conceptual overview, problem identification and decomposition, context and

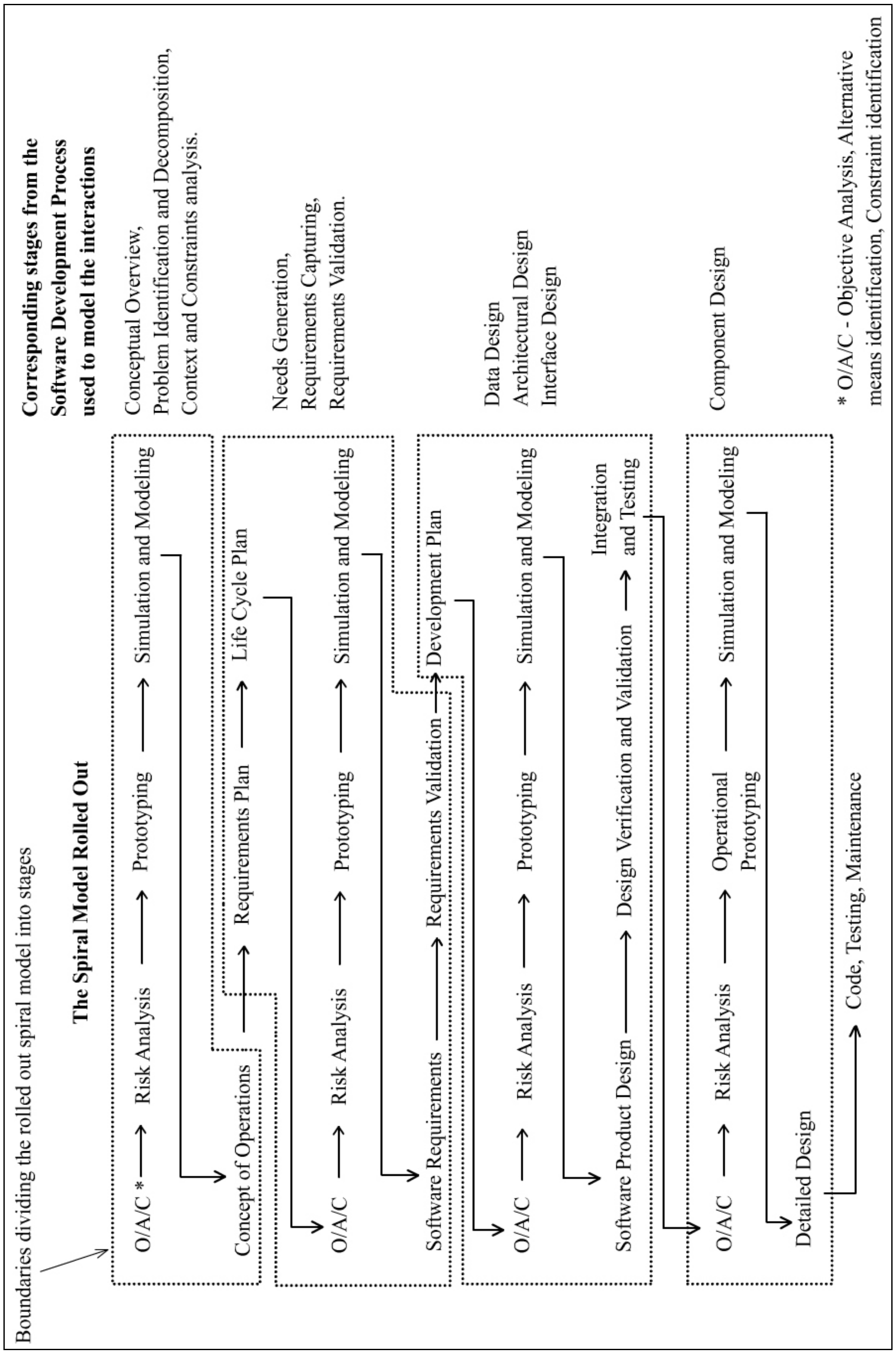
Boundaries dividing the rolled out spiral model into stages
The Spiral Model Rolled Out
Corresponding stages from the Software Development Process used to model the interactions
O/A/C *
Risk Analysis
Prototyping
Simulation and Modeling
Conceptual Overview, Problem Identification and Decomposition, Context and Constraints analysis.
Concept of Operations
Requirements Plan
Life Cycle Plan
O/A/C
Risk Analysis
Prototyping
Simulation and Modeling
Needs Generation, Requirements Capturing, Requirements Validation.
Software Requirements
Requirements Validation
Development Plan
O/A/C
Risk Analysis
Prototyping
Simulation and Modeling
Data Design
Architectural Design
Interface Design
Software Product Design
Design Verification and Validation
Integration and Testing
O/A/C
Risk Analysis
Operational Prototyping
Simulation and Modeling
Component Design
Detailed Design
Code, Testing, Maintenance
* O/A/C - Objective Analysis, Alternative means identification, Constraint identification

Figure 4.6 The Spiral Model rolled out

constraints analysis, and needs generation phases. Therefore, we conclude that a correspondence relationship exists between the first stage of the spiral model and the initial stages of the SE development process used to model the interactions.

2. *Software requirements elicitation, requirements validation, and development of an action plan.*

   The activities of software requirements elicitation and requirements validation correspond to the needs generation, requirements capturing and validation from the SE development process. Additionally, the development of an action plan, or a project plan, which is performed as a part of the spiral model activities, is also performed as a part of the SE development process. Therefore, a correspondence relationship exists between the second stage of the spiral model and the requirements specification stages of the SE development process.

3. *Software product design, design validation and verification, integration and testing plans, and detailed design of components.*

   The product preliminary design, overall design validation and verification, and integration and testing plans are a part of the data, architecture, and interface design practiced in the spiral model. The detailed design stage involves the design of the details about the components of the software being developed using spiral development. The design validation and verification validates the preliminary design to ensure good design practices. Plans for integration and testing of the product are developed once the design validation is complete. Once the integration and testing plans are complete, the developers move to the detailed design of components. A correspondence relationship is evident among the software design, integration planning stages of the spiral model, and the data, architecture, and interface design stages of the SE development process. The detailed design stage corresponds to the design of component details from the SE development process used to model the interactions.

4.3.2 Coordinating UE process iterations with the SE spiral development

In this subsection, we provide an overview of the issues that could be encountered while modeling a development effort that incorporates usability and software engineering processes as separate but coordinated efforts using the spiral model for software engineering.

4.3.2.1 The accommodation of change in requirements in the UE iteration and the SE increment

The plans for further stages from the spiral model may partition the product into increments for successive development of components. In such a case, we can visualize the development process as a set of parallel spirals, one for each component. The accommodation of additional requirements should, therefore, not pose a large threat to the project development from the schedule and rework points of view. The change in requirements would generally lead to a large amount of rework, but the spiral model incorporates risk analysis. Therefore, the spiral model should help the developers anticipate risks related to volatile requirements. Knowledge about the existence of a risk itself is useful for the developers to be cautious and plan mitigation strategies. The spiral model can therefore be effective in accommodating volatile requirements.

4.3.2.2 Issues in designing a coordinated development framework using the spiral model

In section 4.2, we have described the variation in multiplicities of the correspondence relationships among the SE and UE process activities if the Incremental Model is used to model the software engineering process. We expect a similar variation in multiplicities of the correspondence relationships when the spiral model is used to model the software engineering process. If plans for further stages of development use the spiral model to partition the product into increments for successive development, we can visualize the development process as a set of parallel spiral increments. These parallel increments can have a variance in multiplicity with the usability iterations. The development team would then encounter issues similar to those encountered while using the Incremental Model.

In this chapter, we present an investigation of the Incremental Model of development, with the intent of tailoring it to the needs of a coordinated development process. We describe the advantages in application of the incremental development model to coordinated development and suggest modifications to mitigate its drawbacks. We describe the spiral model and give an overview of the issues involved in its application to the coordinated development process.

In the next and final, chapter of this thesis report, we summarize our research which we have documented and derive some conclusions from the research. We also describe the avenues for possible future work on this subject.

CHAPTER 5. SUMMARY, CONTRIBUTIONS, EVALUATION, AND FUTURE WORK

In the preceding chapters, we have presented the exchange of information that ideally should exist among the UE and SE process activities, while designing a common development framework that incorporates the UE and SE processes. In chapter 2, we presented the relevant background for identification of required exchange of information between these processes. We then presented the exchange of information that should exist among the activities of the two processes. We also identified issues that must be addressed while developing a coordinated development framework that incorporates these processes. We now summarize this body of work and present future opportunities for research related to the development of the coordinated development framework.

5.1 Summary

Research on the development of a coordinated framework that incorporates UE and SE processes is motivated by the required exchange of information among activities of these processes during software development. The exchange of information ensures common design understanding between the UE and SE teams. The intent behind the identification of interactions between the UE and SE teams is to design a framework that incorporates the UE and SE processes. The differences in focus, methods, and terminology used by the UE and SE teams make integration of the two difficult. However, the UE and SE processes have to be performed in coordination during software development. In our research, we have identified the interactions necessary between the UE and SE teams, and have highlighted issues in the design of a coordinated development framework. The research documented in this thesis can be summarized as follows:

1. *Definition of an interface between the UE and SE processes*

   The interactions between the two teams define an interface between the two processes. The identification of interactions between the two teams, and the definition of an interface between the two processes is not enough to implement the interactions in practice. The identification of interactions was achieved as follows:

- While identifying the interactions that should ideally exist among the activities of the UE and SE processes, we studied the UE and SE processes in detail, understanding the focus and objectives of each activity of both processes. The Scenario Based Design (SBD) process was used to understand the activities of the UE process. The Synergistic requirements generation model (SRGM) was used to understand the SE requirements engineering activities, while the structured analysis and design approach was used to understand the activities of the SE design process.
- An initial understanding of the activities of the UE and SE processes led us to a high-level identification of information exchange that should exist between the UE and SE processes. An in-depth analysis of the focus and objectives of these activities helped us identify activities that influence one another and identify the information exchange necessary among these activities.

2. *Issues to be addressed while defining a coordinated development framework*

   A coordinated development framework is required to guide the implementation. The development framework must incorporate the UE and SE processes, and, at the same time, have enough flexibility to be tailored to the requirements of specific development efforts. In our work, we identify the issues that must be resolved in order to design the framework. This identification of issues is summarized as follows:

   - As mentioned earlier, just the identification of interactions among activities of the UE and SE processes is not enough for coordinated development because of the nature of the UE and SE processes. The UE process is highly iterative, while the SE process takes a “waterfall” form. Moreover, the software engineers largely practice incremental development of software, with overlapping development of increments. To accommodate these development practices, a coordinated development framework that guides software development is necessary. This framework should not only guide software development, but also have enough flexibility to be tailored to the needs of specific software development efforts.

- Due to the nature of the UE and SE processes, an iteration of the UE process can require information exchange with one or more than one SE increment. Similarly, one SE increment can correspond to several UE increments. This variation in multiplicity is the major issue that should be addressed during the design of the coordinated development effort. The coordinated development effort should also use a standard software development model that guides the software engineering. Therefore, we investigate the applicability of the incremental and the spiral models of software development for this purpose.

The following section discusses the conclusions we derive from this research effort.

5.2 Contributions

The coordinated development framework is required to facilitate interactions between the UE and SE teams. The definition of an interface between the UE and SE processes is the crux of this framework. The interface provides a much-needed structure to the communication between the UE and the SE teams during software development. Identification of interactions that should ideally exist between these two teams is crucial to the definition of this interface.

Identification of exchange of information that should ideally exist among activities of the UE and SE process is a major contribution of this research. The exchange of information among major activities has been identified and visually represented at a high level. This exchange has also been detailed at a lower level of decomposition of major activities. The high-level representation serves the purpose of implementations that follow the same generic UE and SE activities, but have tailored processes. Implementations that do not follow a formal usability process, or follow the SBD for usability engineering, and, at the same time, use incremental software engineering also, can use the detailed exchange of information to tailor their own process. A coordinated development framework can be designed from the identified information exchange and can be modeled using the incremental or spiral software engineering models.

5.3 Evaluation of the framework of information exchange

In this subsection, we present a subjective evaluation of our framework of information exchange based on the initial goals and objectives that are a part of the solution approach included in the introductory chapter. We also present guidelines that can be used to evaluate the effectiveness of our framework using an objective evaluation.

5.3.1 Subjective evaluation of the framework

We base the subjective evaluation of the framework of information exchange among UE and SE activities on the goals and objectives mentioned in Chapter 1. Our goals were to define a framework that guides the exchange of information among UE and SE activities, and coordinates influencing activities of both processes with one another. We realized these goals through a set of objectives. In this subsection, we evaluate our success in meeting these objectives.

The first set of objectives was to identify component activities of the UE and SE processes and the goals and objectives of these activities. Based on the identified goals and objectives, activities from both processes that influence one another were identified. The SRGM [Sud 2003] and Structured Analysis and Design [Pressman 2001] were used respectively to identify the activities of the Requirements Engineering and Software Design phases of the SE process. On the other hand, the SBD process [Rosson 2002] was used to understand the analysis and design activities of the UE process. While identifying component activities of both processes, the purpose and role of these activities as a part of the larger process was also identified. The focus was on coordinating the activities of the Requirements and Design phase of the SE process. After identifying the goals and objectives of both activities, activities that influence one another and the information that must be exchanged among them were identified. One can argue for a more comprehensive set of information exchanges. We have focused on the major points of information exchange rather that going into the minutiae.

We have identified groups of related activities of the UE and SE processes and established concurrency between groups of activities that contain influencing activities. The notion of activity awareness and continuous interactions between concurrent groups of activities is represented in the form of synchronizations. Synchronizations maintain groups of influencing activities concurrent to one another. They do this by enforcing a requirement that both SE and UE processes can proceed to the next group of activities only after the completion of the current groups of activities. Synchronizations therefore make coordination and synchronization among UE and SE activities possible.

After identifying the influences among UE and SE activities and establishing concurrency between groups of activities, we get our basic framework of information exchange among the UE and SE activities. This framework is then applied to the modeling of a coordinated development process that incorporates both UE and SE processes as separate but coordinated practices. The Incremental and Spiral models of software engineering are used as candidate SE models. An issues perspective is taken and issues that must be addressed while developing this model are highlighted. The Incremental and Spiral models are examined to ensure their applicability to the development of a coordinated process model. Modifications are suggested to the Incremental model to enhance its capability in handling change in requirements, which is a major issue in the coordinated process model. Moreover, issues evident while relating iterations of the UE process and increments of the SE process are highlighted.

Concluding this subjective evaluation, we claim to have achieved our initial goals of defining a basic framework of information exchange among the UE and SE activities and coordinating influencing activities of both processes with one another.

5.3.2 Objective evaluation of the framework: guidelines to measure effectiveness

The following guidelines can be used during an objective evaluation of the framework. Here, we assume that in an experimental setup two teams containing both usability and software engineers, one using our framework, and one using an ad-hoc approach develop software using the same high level requirements for the same customer. The following factors can be used to evaluate the effectiveness of the framework of interactions.

- *Usability of the product developed.*

The final product developed by both teams can be evaluated using potential users for measuring the usability of the product. A higher usability rating for the product developed using our framework of information exchange shall imply that our framework ensures higher usability of the product developed.

- *Support for all functionality that the users perceive necessary*

  The final product developed by both teams can be evaluated for support of necessary functionality. The product developed using our framework of information exchange should ideally support all the functionality identified as necessary.

- *Changes to requirements earlier during development*

  If the development processes of both teams are logged to keep track of changes in requirements and rework necessary to address these changes, the team using our framework should ideally experience changes in requirements earlier in the development process. We claim so because in a development process using our framework, a majority of conflicts between the ideas of the UE and SE teams shall be resolved early in the project. This shall happen because a development process using our framework shall require frequent interactions between usability and software engineers during the initial stages of the project.

- *Lesser rework*

  The development team using our framework should ideally experience lesser rework than the team who does not use the framework. We claim so because our framework supports frequent exchange of information between the usability and software engineers that leads to reduced conflicts and lesser rework therefore.

5.4 Future work

This section identifies the avenues for future work towards the development of a coordinated framework that incorporates UE and SE activities.

5.4.1 Resolution of unaddressed issues in the application of the incremental model to the coordinated development framework.

We have investigated the application of the incremental and spiral models to model the software process under the development framework that incorporates the software and usability engineering processes as coordinated but separate efforts. Two major issues need to be resolved in the application of these models to the development framework. While applying the SE incremental model to the SE process, the first issue refers to the accommodation of added and changing requirements on the part of the software engineering process modeled using the incremental model. The second concern refers to the variance in multiplicity among the UE iterations and the SE increments, and the issues related to temporal coordination that arise because of this variance. We expect several other issues to become evident during the resolution of the issues already evident.

5.4.2 Resolution of unaddressed issues in the application of the spiral model to the coordinated development framework.

Issues faced in an effort to define the coordinated development process using the spiral model for software engineering are largely similar to the issues in applying the SE incremental model to the coordinated development process. The issues related to addition and change in requirements, though partly addressed by the risk management mechanism embedded in the spiral model, are a major concern. The issues introduced by the variation in multiplicity among the UE iterations and spiral development in the form of increments is a major issue as well. An effort that attempts to develop a coordinated development process based on the spiral model needs to address these issues. As in the case of the incremental model, we expect several other issues to become evident during the resolution of the issues already evident.

5.4.3 Design of artifacts and communication protocol for change notifications.

Changes in requirements and design produced by the usability or the software engineers can take place during the software development process. These changes may influence the requirements or design of the other coordinated process. Therefore, communication of these changes between the usability and software engineers is necessary. The communication of change information may be achieved asynchronously, using change artifacts. At the same time, some changes may give rise to conflicting requirements or design specifications. These conflicts must be resolved through active communication between the usability and software engineers in the form of negotiation meetings. Artifacts for asynchronous notification of changes in requirements, and a communication protocol for one team to notify the other about a necessary negotiation meeting have to be designed.

5.4.4 Design of verification strategies.

Another major concern is the design of verification strategies. The communication of information among process activities is the crux of the interactions among the SE and UE process activities. This communication of information is possible through the modeling of artifact templates that contain all the required information. Software developers should ideally tailor these artifacts to the requirements of particular development efforts. Verification strategies are required to ensure that the artifacts developed adhere to the defined templates, if any, and are adequate to transfer all the required information of one process activity to another activity of the other concurrent process. We recognize design of verification strategies as an important issue and an avenue for future research on the topic.

5.4.5 Design of validation strategies.

The definition of validation strategies is also a major concern in the development of a coordinated development process. Developers should ideally validate the interface and functional design with the customer to ensure that the design matches the customer requirements. We can design the validation strategies to be distinct for the SE process and the UE process. The SBD based UE process employs regular prototyping throughout the UE process to ensure good design. The software engineering models advocate and employ similar verification strategies to ensure customer buy-out. In the case of a coordinated development model, the developers could combine the validation strategies for the SE and the UE processes. The advantages of a combined design validation would be that the customer would not need to interact with two different teams: one for verification of the interface and the other for the verification of functionality. This would reduce the complications in customer interaction. Moreover, in cases where the customer cannot necessarily devote enough time for validation activities all throughout the development process, having combined validation strategies would reduce the load on the customer.

5.4.6 Objective Evaluation of the framework.

Section 5.2 includes basic guidelines for the objective evaluation of the framework of information exchange among the UE and SE activities. The objective evaluation has to be performed and results evaluated statistically in order to evaluate the effectiveness of the model in ensuring the production of usable software.

REFERENCES

[Beck 1999] Beck, K., *Extreme Programming Explained: Embrace Change*, Addison-Wesley Publishing Company, 1999.

[Bell 1976] Bell, T. E., Thayer, T. A., *"Software requirements: Are they really a problem?"* Proceedings of the 2nd international conference on Software engineering, pp.61-68.October 13-15, 1976, 1976.

[Boehm 1981] Boehm, B., *Software Engineering Economics*, Prentice-Hall, Engelwood Cliffs, NJ, 1981.

[Boehm 1988] Boehm, B., *"A Spiral Model for Software Development and Enhancement"*, IEEE Computer, vol. 21, no. 5, pp.61-72.1988.

[Booch 1994] Booch, G., *Object-Oriented Analysis and Design with Applications, 2nd ed.*, Addison-Wesley, 1994.

[Brackett 1990] Brackett, J. W., *Software Requirements*, SEI Curriculum Module SEI-CM-19-1.2, 1990.

[Brooks 1987] Brooks, F. P., Jr., *"No silver bullet: essence and accidents of software engineering"*, v.20 no.4, pp. 10-19, April 1987., 1987.

[Carter 2001] Carter, R. A., Anton, A. I., Dagnino, A., Williams, L.; *"Evolving beyond requirements creep: a risk-based evolutionary prototyping model"*, Proceedings of the Fifth IEEE International Symposium on Requirements Engineering, pp.94-101.2001.

[Cockburn 1995] Cockburn, A., *Unraveling Incremental Development*, 1995, http://alistair.cockburn.us/crystal/articles/uid/unravelingincrementaldevelopment.html

[Constantine 2003] Constantine, L., Biddle, R., Noble, J.; *"Usage-Centered Design and Software Engineering: Models for Integration"*, Proceedings of the ICSE 2003, pp.106-113.2003.

[Davis 1993] Davis, A. M., *Software Requirements: Objects, Functions, & States*, Prentice-Hall, Upper Saddle River, New Jersey, 1993.

[Ferré 2003] Ferré, X., *"Integration of Usability Techniques into the Software Development Process"*, Proceedings of the International Conference on Software Engineering, pp.28-35.2003.

[Ferré 2002] Ferré, X., *The STATUS project*, 2002, http://www.ls.fi.upm.es/status/

[Groener 2002] Groener, M. K., *Capturing requirements meeting customer intent: a structured methodological approach*, Virginia Tech ETD Collection, 2002, http://scholar.lib.vt.edu/theses/available/etd-05232002-234024/

[Hix 1993] Hix, D., Hartson, H. R., *Developing User Interfaces*, John Wiley, New York, 1993.

[IEEE 1993] IEEE, *IEEE Standards Collection: Software Engineering*, IEEE Standard 610.12-1990, IEEE, 1993.

[Juristo 2003] Juristo, N., Lopez, M., Moreno, A. M., Sánchez, M. I.; *"Improving software usability through architectural patterns"*, Proceedings of the International Conference on Software Engineering, pp.12-19.2003.

[Juristo 2001] Juristo, N., Windl, H., Constantine, L.; *"Special Issue on Usability Engineering in Software Development"*, IEEE Software, Vol. 18, no. 1, 2001.

[Leffingwell 2000] Leffingwell, D., Widrig, D., *Managing Software Requirements: A Unified Approach*, Addison Wesley Publishing Co, 2000.

[Lewis 1977] Lewis, R. O., *The Cost of an Error: A Retrospective Look at Safeguard Software*, Science Applications International Corporation, Huntsville, AL, 1977.

[Martin, J. 1991] Martin, J., *Rapid Application Development*, Prentice-Hall, 1991.

[Martin, R.C. 1999] Martin, R. C., *Iterative and Incremental Development (IID), Part 2,* Journal of C++ Report, 1999.

[Mayhew 1999] Mayhew, D. J., *The Usability Engineering Lifecycle: A Practitioner's Handbook for User Interface Design*, Academic Press/Morgan Kaufmann, 1999.

[McDermid 1993] Mcdermid, J., Rook, P., *"Software Development Process Models"*, Software Engineer's Reference Book, CRC Press, pp.15/26 - 15/28.1993.

[Milewski 2003] Milewski, A., *"Software Engineering Overlaps with Human-Computer Interaction: A Natural Evolution"*, Proceedings of the International Conference on Software Engineering, pp.69-71.2003.

[Norman 1988] Norman, D. A., *The psychology of everyday things*, Basic Books, New York, 1988.

[Paech 2003] Paech, B., Kohler, K., *"Usability Engineering integrated with Requirements Engineering"*, Proceedings of the International Conference on Software Engineering, pp.36-40.2003.

[Pressman 2001] Pressman, R. S., *Software Engineering: a Practitioner's Approach, 5th Edition,* McGraw-Hill, New York, NY, 2001.

[Pyla 2004] Pyla, P. S., Pérez-Quiñones, M. A., Arthur, J. D., Hartson, H. Rex; *"Towards a Model-Based Framework for Integrating Usability and Software Engineering Life Cycles"*, ACM: Computing Research Repository (CoRR), Technical report, cs.HC/0402036, 2004.

[Raccoon 1995] Raccoon, L. B. S., *"The Chaos Model and the Chaos Life Cycle"*, ACM Software Engineering Notes, vol. 20, no. 1 January, 1995, pp.55-66.1995.

[Rosson 2002] Rosson, M. B., Carroll, J. M., *Usability Engineering: Scenario-based development of human-computer interaction*, Morgan Kauffman Publishers, 2002.

[Royce 1970] Royce, W. W., *"Managing the Development of Large Software Systems: Concepts and Techniques"*, Proc. WESCON, 1970.

[Rubey 1975] Rubey, R. J., Dana, J. A., Biché, P. W.; *"Quantitative Aspects of Software Validation"*, IEEE Transactions on Software Engineering, Vol. 1, no. 1, pp.150-155.1975.

[Sidky 2002] Sidky, A. S., Sud, R. R., Bhatia, S., Arthur, J. D.; *"Problem Identification and Decomposition within the Requirements Generation Process"*, 6th World Multiconference on Systems, Cybernetics, and Informatics (SCI 2002), Vol. VIII, pp.333-338.July 2002, 2002.

[Sommerville 1996] Sommerville, I., *Software Engineering, 5th ed.* . Addison Wesley Publishing Co, Reading, MA, 1996.

[Standish 1995] Standish, *Chaos Report, The Standish Group*, 1995, http://www.standishgroup.com/sample_research/chaos_1994_1.php

[Sud 2003] Sud, R., *A Synergistic Approach to Software Requirements Generation: The Synergistic Requirements Generation Model (SRGM) and, An Interactive Tool for Modeling SRGM (itSRGM)*, Virginia Tech ETD Collection, 2003, http://scholar.lib.vt.edu/theses/available/etd-05182003-111744/

VITA

Sourabh Pawar was born on August 25, 1980 in Phaltan, India. Brought up in Bombay, he graduated from the University of Bombay, in June 2002, with a Bachelors' degree in Computer Science and Engineering. Subsequently, he has been pursuing a Masters' degree in Computer Science at Virginia Tech, Blacksburg, Virginia.

During his Bachelors' degree, he was actively involved in research in computer science through his affiliations with the Department of Computer Science at the Indian Institute of Technology, Bombay, and as a member of the IEEE. He has also been the founder member of the IEEE Students branch in his undergraduate institution, registered with the IEEE Bombay section. Sourabh has also worked as the technical lead for an inter-collegiate event, one of the finest in Bombay, held in his undergraduate institution in the spring of 2001.

During his Masters' degree, he has served as a Graduate Teaching Assistant, and has been conducting research in the field of Software Engineering.

On completion of his degree, Sourabh has plans of gaining industrial experience and conducting research in the field of Enterprise Resource Planning. He can be reached at sourabh@vt.edu.

Sourabh A. Pawar