13 RIPPLE: AN EVENT DRIVEN DESIGN REPRESENTATION FRAMEWORK FOR INTEGRATING USABILITY AND SOFTWARE ENGINEERING LIFE CYCLES

Pardha S. Pyla,
Manuel A. Pérez-Quiñones, James D. Arthur, and H. Rex Hartson

Department of Computer Science, Virginia Polytechnic Institute and State University,
660 McBryde Hall, Blacksburg, VA 24061, USA
{ppyla, perez, arthur, hartson}@cs.vt.edu

Abstract

Ripple is a database-centered, event-triggered, shared design representation framework that provides a development infrastructure within which the usability engineering and software engineering life cycles co-exist in cooperative and complementary roles. Ripple identifies connections and dependencies within each life cycle and between the two life cycles and provides a framework to represent artefacts generated at each stage of the two development life cycles. Our approach to integrating these two development life cycles does not merge them into a single life cycle; rather it coordinates each life cycle's activities, timing, scope, and goals using a shared design representation and management for the two life cycles. Ripple incorporates tech-

A. Seffah (eds.), Human-Centered Software Engineering – Integrating Usability in the Development Process, 245–265.

niques to accommodate communication about design insights and change. In response to design changes by either the interface or software side, Ripple sends possibly cascading messages (ripples) to inform developers on both sides, asking them to satisfy associated constraints (dependencies, relationships) affecting related other parts of the overall design. We describe the motivation, barriers, rationale, arguments, and implementation plan for the need, specification, and potential contributions of such an integrated design representation framework. We provide a high level description of this design representation framework and conclude with the usefulness and potential shortcomings of this approach.

13.1 INTRODUCTION

13.1.1 Parts and Processes of Interactive Software Systems

Interactive software systems have both functional and user interface parts. Although the separation of code into two clearly identifiable modules is not always possible, the two parts exist conceptually and each must be designed on its own terms.

The user-interface part, which often accounts for half or more of the total lines of code (Myers and Rosson, 1992), begins as an interaction design, which is ultimately implemented in user interface software. Interaction design requires specialized usability engineering (UE) knowledge, training, and experience in topics such as human psychology, cognition, visual perception, specialized design guidelines, task analysis, etc. The ultimate goal of UE is to create systems with measurably high usability, i.e., systems that are easy to learn, easy to use, and satisfying to their users. A practical objective is also to provide interaction design specifications that can be used to build the interactive component of a system by software engineers. In this chapter we define the UE role as that of the developer who has responsibility for building such specifications. (We use the term developer to refer to someone who has the skills to participate in all stages of a software development life cycle and not just a software coding or implementation expert).

The functional part of a software system, sometimes called the functional core, is manifest as the non-user-interface software. The design and development of this functional core requires specialized software engineering (SE) knowledge, training, and experience in topics such as algorithms, data structures, software architectures, calling structures, database management, etc. The goal of SE is to create efficient and reliable software systems containing the specified functionality, as well as integrating and implementing the interactive portion of the system. We define the SE role as that of the developer who has the responsibility for this goal.

To achieve the UE and SE goals for an interactive system, i.e., to create an efficient and reliable system with required functionality and high usability, effective development processes are required for both the UE (Figure 13.1) and SE life cycles (Figure 13.2). The UE development life cycle is an iteration of activities for requirement analysis (e.g., needs, task, work flow, user class analysis), interaction design (e.g., usage scenarios, screen designs), prototype development, and evaluation thereby producing a user interface interaction design specification. The SE development life cycle consists primarily of concept definition and requirements engineering, design

(preliminary and detailed design), design review, implementation, and integration & testing, I&T).

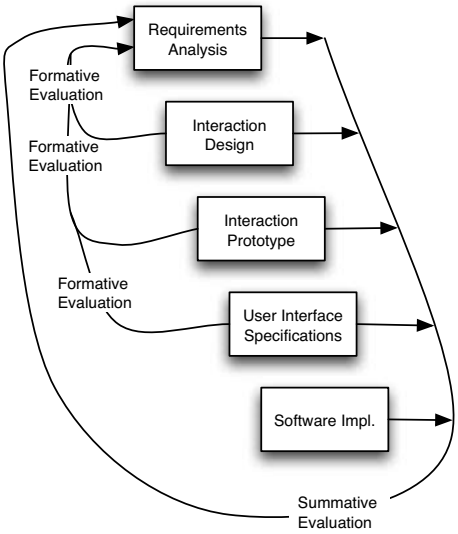

Figure 13.1 Usability engineering life cycle

13.1.2 The Problem: Coordinating the Development of the Two Life Cycles

Given the facts that each of these development life cycles is now reasonably mature and well established, both have the same high level goal of producing software that the user wants and needs, and that the two must function together to create a single system, one might expect well-defined connections for collaboration and communication between the two development processes. However, the two disciplines are still considered as separate entities and are applied independently with little coordination during product development. For example, it is not uncommon to find usability engineers being brought into the development process after the SE implementation stage. They are asked to test and/or 'fix' the usability of an already-implemented system, and then, of course, many changes proposed by the usability engineers that require significant modifications must be ignored due to budget and time constraints. Those few changes that actually do get included require a significant investment in terms of time and effort because they must be retrofitted.

The lack of coordination between the usability and software engineers often leads to conflicts, gaps, design and requirements mismatches, miscommunication, "spaghetti" code due to unanticipated changes, brittle software, and other serious problems during development. The result is a system falling short in both functionality and usability,

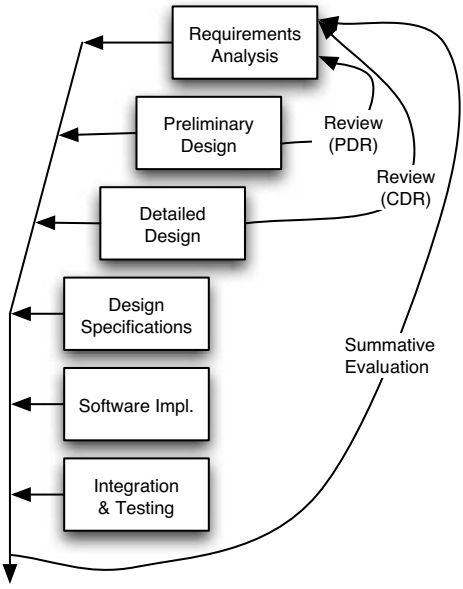

Figure 13.2 Software engineering life cycle

and in some cases a completely failed project. In particular, for the projects containing a significant interaction component, there is a need for:

- communication between the UE and SE roles, each of which uses different development activities, techniques, and vocabularies;
- coordination of independent development activities (usability and software engineers coordinating while mostly working separately on role-specific activities);
- identification and understanding of constraints and dependencies between the SE and UE processes;
- synchronization of dependent development activities (timely readiness and timeliness of making use of respective work products); and the
- provision on each side for anticipating and reacting to change on the other side.

Unfortunately, the significance of UE and the importance of the bulleted items above are not described or prescribed in most of the software development standards that exist today. For example, the 31-page IEEE-830 standard (IEEE, 1998) on recommended practices for software requirements specification (SRS) contains only about

10 lines relating to user interfaces (Section 5.2.1.2), and states that user interface specifications should be a part of the SRS. This part of the standard takes an ad hoc stab at a few user interface issues (e.g. required screen formats, page and window layouts, screen content, availability of programmable function keys, etc.) which seem arbitrarily chosen from the enormous possibilities not mentioned. More importantly, it says nothing about the UE life cycle process for creating the interaction design, which is a main part of the user interface software specification. It is misguided (and worse, misguiding) to expect the user interface specifications to be available that early in the requirements process without having followed a proper UE design life cycle. We believe that this document should have a reference to another standard for user interface software requirements.

Another source of confusion with the IEEE-830 standard is that the items mentioned in this document such as required screen formats, page and window layouts, and screen content are design specifications for usability engineers (the standard includes nothing about how to design them for usability). For the UE role, 'requirements' are mostly stated in terms of usability attributes such as learnability, subjective satisfaction, ease of use, etc. Even these usability specifications are subject to calibration in later stages of the UE development process.

However, we do not disagree with the intent behind the idea that user interface requirement specifications for user interface software are properly a part of the SRS. But in reality it is not possible to generate requirements specifications for user interface software without going though an iterative process of interaction design and evaluation, but standards such as the above described IEEE-830 (on SRS) and IEEE/EIA-12207.1 (on software life cycle processes-life cycle data—Software Productivity Consortium, 1997) do not acknowledge the kind of life cycle process that is needed to develop a high usability interaction design. Neither do they acknowledge the myriad relations and dependencies between the activities and work products of the SE life cycle with that of UE and vice versa.

13.1.3 Objective

The objective of our work has been to produce a design representation infrastructure that:

- integrates the two life cycles under one common framework;
- retains the two development processes as separately identifiable processes, each with its own life cycle structure, development activities, and techniques; and
- is built upon a database-centered, event-triggered and constraint-based framework, that provides a common 'overall design representation and management' approach, shared by the SE and UE roles and activities.

The common 'design representation and management' is the key to the coordination of interface and functional core development activities, and to the communication among the UE and SE roles. The constraint-based event triggers are important in recognizing an event with an associated dependency or constraint and sending a message to remind

the developers to enforce a constraint. The common 'design representation' identifies and addresses the effects of change and also incorporates techniques to record design reminders. This allows the two life cycle roles to

- design for change by keeping the design flexible,
- analyze the implications of change in either of the processes,
- take necessary corrective action to address change,
- mitigate the changes that could be imposed on each life cycle, and to
- record design insights and reminders for future development activities.

13.2 BACKGROUND

13.2.1 Operating hypothesis

A strong hypothesis for our work is to maintain UE and SE as separate, but coordinated, life cycle development processes. It is not our goal to merge either development process into the other, but to establish a development infrastructure in which both can coexist and function in parallel. UE and SE processes each require special knowledge and skills. Given the differences in activities and focus, it is not realistic or desirable to expect the two roles to 'work together'. A combined life cycle process is unlikely to give balanced attention to both parts. Trying to combine, for example, the UE life cycle into the SE life cycle, as done in (Ferre, 2003), creates a risk (and a high likelihood) of deciding conflicts in favor of software development needs and constraints, and against those of usability. The two roles must however communicate, coordinate, and synchronize as they work on essentially two different parts of a larger design, parts that must come together for implementation of a single system.

13.2.2 Similarities Between Life Cycles

At a high level, UE and SE share the same objectives:

- seeking to understand the client's, customer's, and users' wants and needs;
- translating these needs into system requirements;
- designing a system to satisfy these requirements; and
- testing to help ensure their realization in the final product.

13.2.3 Differences Between Life Cycles

The objectives of the SE and UE are achieved by the two developer roles using different development processes and techniques. At a high level, the two life cycles differ in the requirements and design phases but converge into one at the implementation stage (Figure 13.3). This is a natural expectation because ultimately software developers implement the user interface specifications. At each stage, the two life cycles have

many differences in their activities, techniques, timelines, iterativeness, scope, roles, procedures, and focus. Several of the salient differences are identified next.

Different Levels of Iteration and Evaluation. Developers of interaction designs often iterate early and frequently with design scenarios, screen sketches, paper prototypes, and low-fidelity, roughly-coded software prototypes before much, if any, software is committed to the user interface. Often this frequent and early iteration is done on a small scale and scope, and primarily as a means to evaluate a part of an interaction design in the context of a small number of user tasks. Usability engineers evaluate interaction designs in a number of ways, including early design walk-throughs, focus groups, usability inspections, and lab-based usability testing. The primary goal is to find usability problems or flaws in the interaction design.

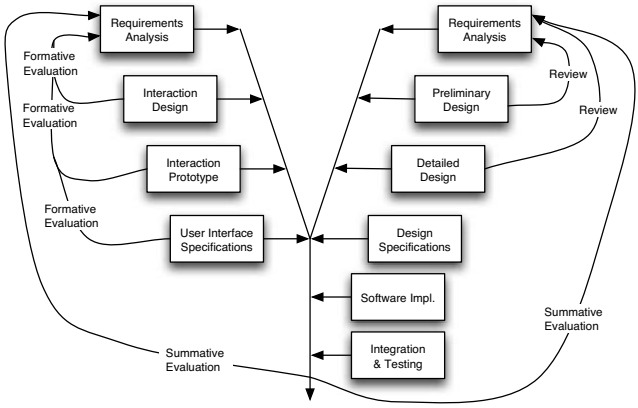

Figure 13.3 Current practices: Process without communication/coordination

Software engineers identify the problem, decompose and represent the problem in the form of requirements (requirements analysis block in Figure 13.2), transform the requirements into design specifications (preliminary and detailed design blocks in Figure 13.2), and then implement those design specifications. In the early days of software engineering, these activities were often performed using the sequential waterfall model (Royce, 1970). Later, these basic activities were incorporated into more iterative processes such as the spiral model (Boehm, 1988) (which has a risk analysis and an evaluation activity at the end of each stage). Even though the more recent SE development life cycles are evolving towards the UE style by anticipating and accommodating changes at each iteration, they still stress iteration on a larger scale and scope. Moreover, testing and validation, which ensures integration accuracy and conformance to system specifications, are performed more towards the end of the

development process and can include software for both the functional core and the user interface.

Differences in Terminology. Even though certain terms in both life cycles sound similar they often mean different things. For example:

- In UE, 'testing' is a part of design, and is diagnostic in nature and is used to find and fix problems in the interaction design (identified as formative evaluation in Figure 13.1). In SE 'testing' is an independent stage where the objective is to check the implementation of the system and to validate its conformance to specifications. Analysis and verification of the design specifications performed in SE is often called 'review' (identified in Figure 13.2). When the specifications pass the review stage, they become a binding document between the client and the development team.

- A (use case) scenario in SE (in object oriented design paradigm) is used to "identify a thread of usage for the system to be constructed (and) provide a description of how the system will be used" (Pressman, 2005b). Whereas in UE, a design usage scenario is "a narrative or story that describes the activities of one or more persons, including information about goals, expectations, actions, and reactions (of persons)" (Rosson and Carroll, 2002a).

- The SE group refers to the term 'develop' to mean creating software code, whereas the usability engineers use 'develop' to mean iterate, refine, and improve usability to create an interaction design.

Overall, the software engineers concentrate on the system whereas the usability engineers concentrate on users. Such fundamental difference in focus is one more reason why it is difficult to merge these two life cycles.

Differences in Requirements Representation. Most requirement specifications documented by software engineers use plain English language and are generally very detailed. These specifications are written specifically to drive the SE development process. On the other hand, usability engineers specify interactive component issues such as feedback, screen layout, colors, etc. using artefacts like prototypes, design scenarios, and screen sketches. These artefacts are not detailed enough to derive software design, instead they require additional refinement and reformulation before implementation. Therefore, they cannot be used to directly drive the software development process.

13.3 CURRENT PRACTICES

In spite of the extensive research and maturity levels achieved in the UE and SE life cycle areas, there has been a marked deficiency of understanding between the corresponding developer roles. In general, the two teams do not understand the other's goals and needs and do not have an appreciation for the other's area of expertise (see Chapter 15 by Battle for more on a practical view of the relationship between the two

sides of this issue). One apparent reason for this situation is the way computer science courses are typically offered in colleges: SE courses often omit any references to user interface development techniques (Douglas et al., 2002), and UE courses do not discuss the SE implications of usability patterns (Pyla et al., 2004).

Some software life cycles in practice today are documentation intensive and static in nature. The ponderous weight of voluminous static documentation does not allow effective mechanisms to predict or counter the effects of change, especially changes that occur very rapidly in early stages of a life cycle. It can be argued that configuration management processes (Joeris, 1997) that exist in SE are an exception to this. Configuration management tools provide mechanisms and procedures to track changes in the work artefacts generated in a software development life cycle. However, these tools and techniques were mostly developed for SE life cycles; whereas, our work brings some of these principles to the UE side and between the two sides and also incorporates change prediction.

On the other side of the spectrum, many project managers use intensively hands-on-project-management principles wherein a project leader walks around managing and communicating with the various developers in a direct "hands-on" manner taking individual responsibility to make sure all the details are addressed. This approach is based on the potential effectiveness of an informal and low-documentation approach to software development and the fact that a skilled human manager can keep track of what needs to be done better than an automated system. However, this approach does not scale up well as projects get more complex because one person cannot keep track of all the little details and insights about a very large project as it progresses. While intensively hands-on project management can work for some SE life cycles, they are not as suitable for a rapidly evolving and changing life cycle like that of UE, and are even less likely to be effective in communicating all the details of rapid changes between the SE and UE teams.

The general principles and tools of project management (Reifer, 2002) are useful and are well studied in the SE literature. We are aware of their existence and acknowledge their usefulness. However, these tools and principles are mostly about high level issues such as schedules and timelines. Our contribution is more about improving the communication, collaboration, and synchronization of the SE and UE life cycles and thereby increasing the awareness of the specific needs, details and insights for the overall design process. In the process we are hoping to bring some of the advantages of SE life cycle project management to the UE side.

13.3.1 Lack of Coordination of Development Activities

When translated into development activities, this lack of understanding between the two developer roles, combined with an urgency to get their own work done, often leads to working without collaboration (as shown in Figure 13.3), when they could be more efficient and effective communicating and coordinating with one another. For example, both SE and UE roles include field visits to learn about client, customer, and user needs, but they often do this without coordination. Software engineers elicit functional requirements (Pressman, 2005b), and determine the physical properties and operational environments of the system (Lewis, 1992), etc. Usability engineers visit clients

and users to determine, often through "ethnographic studies" (Blomberg, 1995), how users work and what computer-based support they need for that work. They seek task information, inputs for usage scenarios, and user class definitions. Why not coordinate this early systems analysis effort? Much value can be derived from cooperative system analysis and requirements gathering. Such joint activities help in team building, communication, and in each life cycle role recognizing the value, and problems, of the other, in addition to early agreement on goals and requirements. Instead, each development group reports its results in documentation not usually seen by people in the other life cycle. Each uses those results to drive only their part of the system design and finally merge at the implementation stage (Figure 13.3), where it is much too late to discover the differences, inconsistencies, and incompatibilities between the two parts of the overall design. Moreover, this lack of coordinated activities presents a disjointed appearance of the development team to the client. It is likely to cause confusion on the clients: "why are we being asked similar questions by two different groups from the same development team?"

Another significant shortcoming of the practice shown in Figure 13.3 is the fact that the independently generated user interface specifications on the UE side and functional design specifications on the SE side are submitted to the development team at implementation stage. However, because these specifications were developed without coordination and communication, when they are now considered together in detail, developers typically discover that the two design parts do not fit with one another because of large differences and incompatibilities.

13.3.2 Lack of Synchronization of Development Schedules

In current practices, the life cycle roles must synchronize the work products eventually for the implementation and testing phases. However, waiting until one absolutely must synchronize obviously creates problems. Therefore, it is better to have many synchronization points, earlier and throughout the development life cycle. These timely synchronization points would allow earlier, more frequent, and less costly 'calibration' to keep both design parts on track for a more harmonious final synchronization with fewer harmful surprises.

However, as shown in Figure 13.3, the more each team works without communication and collaboration, the less likely they will be able to schedule their development activities to arrive simultaneously at common checkpoints.

13.3.3 Lack of Communication Among Different Life Cycle Roles

Although the two life cycle roles can successfully do much of their development independently and in parallel, a successful project demands that the two roles communicate so that each knows generally what the other is doing and how that might affect its own activities and work products. Each group needs to know how the other group's design is progressing, what development activity they are currently performing, what features are being focused on, what insights and concerns they have for the project, and so on. Especially during the early requirements and design activities, each group needs to be 'light on its feet' and able to respond to events and activities occurring in the counter-

part life cycle. However, current practices (Figure 13.3) do not permit that necessary communication to take place because the two life cycles operate independently; that is, there is no structured development framework to facilitate communication between these two life cycles.

One might argue that the communication process need not be more formal than it is right now and that the usability and software engineering practitioners should be on the same analysis team. Indeed, in their day-to-day life, the two developers are technically on the same analysis team. But our real world experience has shown that this is not enough to foster the necessary communication (especially about features and changes) because each role still focuses almost completely on their own problems and their own designs. For example, the SE role in general is not concerned about UE role's interaction design and vice versa. So the communication focus is not on being formal, but on being complete. Based on our real world experience, day-to-day communication processes have proven to be inadequate and often result in nasty surprises that are revealed only at the end when serious communication finally does occur. This is often too late in the overall process.

13.3.4 Lack of Constraint Mapping and Dependency Checks

Because each part of an interactive system must operate with the other, many system requirements have both SE and UE components. If SE component or feature is first to be captured, it should trigger (or be mapped to) a reminder that a UE counterpart is needed, and vice versa. When the two roles gather requirements separately and without communication, it is easy to capture requirements that are conflicting, incompatible or one-sided. Even if there is some ad-hoc form of communication between the two groups, it is inevitable that some parts of the requirements or design will be forgotten or will "fall through the cracks."

As an example, software engineers perform a detailed functional analysis from the requirements of the system to be built. Usability engineers perform a hierarchical task analysis, with usage scenarios to guide design for each task, based on their requirements. Documentation of these requirements and designs is maintained separately and not necessarily shared. However, each view of the requirements and design has elements that reflect counterpart elements in the other view. For example, each task in the task analysis can imply the need for corresponding functions in the SE specifications. Similarly, each function in the software design can reflect the need for access to this functionality through one or more user tasks in the user interface. When tasks are missing in the user interface or functions are missing in the software, the respective sets of documentation are inconsistent - a detriment to success of the project.

Constraints, dependencies, and relationships exist not only among activities and work products that cross over between the two life cycles but also within each of the life cycles. For example, on the UE side, a key task identified in task analysis should be considered and matched later for a design scenario and a benchmark task. To our knowledge, there are no life cycle frameworks that help in addressing such internal and external constraints, dependencies, and relationships among life cycle activities.

In general, design choices made in one life cycle constrain the design options in the other. In our consulting experience we often encountered situations where the

user interfaces to software systems were designed from a functional point of view and the code was factored to minimize duplication on the backend core. The resulting systems had user interfaces that did not have proper interaction cues to help the user in a smooth task transition. Instead, a task oriented approach would have supported users with screen transitions specific to each task; even though this would have resulted in a possibly "less efficient" composition for the backend. Another case in our consulting experience was about integrating a group of individually designed web-based systems through a single portal. Each of these systems was designed for separate tasks and functionalities. These systems were integrated on the basis of functionality and not on the way the tasks would flow in the new system. The users of this new system had to go through awkward screen transitions when their tasks referenced functions from the different existing systems.

The intricacies and dependencies between user interface requirements and functional core have begun to appear in the literature. For example, in (Bass and John, 2001b), user interface requirements and styles, such as support for undo, are mapped to particular software architectures required for the implementation of such features (see Chapter 6 by Adams, Bass, and John).

Because of the constraints on one another, independent application of the two life cycles (Figure 13.3) is likely to fail. Hence, an integrated design representation framework that facilitates communication and coordination between these two life cycles is essential.

13.3.5 Lack of Provision for Change

In the development of interactive systems, each phase and each iteration has a potential for change. In fact, at least the early part of the UE process is intended to change the design iteratively. This change can manifest itself during the requirements phase (growing and evolving understanding of the emerging system by developers and users), design stage (evaluation identifies that the interaction metaphor was not easily understood by users), etc. Such changes often affect both life cycles because of the various dependencies that exist between and within the two processes. Therefore, change can conceptually be visualized as a design perturbation that has a *ripple* effect on all stages in which previous work has been done. For example, during the usability evaluation, the usability engineer may recognize the need for a new task to be supported by the system. This new task requires updating the previously generated hierarchical task analysis document to reflect the new addition (along with the rationale). This change to the HTA generates the need to change the functional decomposition (by adding new functions to the functional core to support this task on the user interface) on the SE side. These new functions, in turn, mandate a change to the design, schedules, and in some cases even the architecture of the entire system. Thus, one of the most important requirements for system development is to identify the possible implications and effects of each kind of change and to account for them in the design accordingly. Another important requirement is to try to mitigate the impact of change by communicating about changes as early as possible, and by directing that communication directly to the development activities most affected. The more the two developer roles work without a common structure (Figure 13.3) the greater the possi-

bility that inevitable changes in each part will introduce incompatibilities, revealed as "surprises" when they finally do communicate.

13.3.6 Lack of Provision for Accommodating Design and Development Insights

Some dependencies between life cycle parts represent a kind of 'feed-forward', giving insight to later life cycle activities. For example, during the early design stages in the UE life cycle, the usage scenarios provide insights as to how the layout and design of the user interface might look like. In other words, for development phases that are connected to one another (in this case, the initial screen design is dependent on or connected to the usage scenarios), there is a possibility that the designers can forecast or derive insights from a particular design activity. Therefore, as and when the developer encounters such premonitions or potential effects on later stages (on the screen design in this example), there is a need to document them when the process is still in the initial stages (usage scenario phase). This way, when the developer reaches the initial screen design stage, the previously documented insights are readily available to aid the screen design activity. To our knowledge, none of the current approaches to the development of systems with interactive components provide this capability.

13.4 RIPPLE: A DESIGN REPRESENTATION FRAMEWORK

Ripple, a work-in-progress research effort, is a design representation framework that draws concepts from graph theory (relations), analogies from physics (perturbations and ripples), and of course, content from SE and UE. Ripple provides mechanisms for the two development roles to communicate, collaborate, and synchronize with one another, while allowing each life cycle role to function independently. Ripple provides each developer role with activity awareness, information about changes and insights from the developer's own life cycle and from the other development life cycle. It uses a common design representation, which includes an aggregation of the work artefacts from each development life cycle, and the semantics of various constraints, dependencies and relationships between and within the two life cycles. Ripple addresses changes and design perturbations using messages that can be passed along (ripples) among developer roles. Ripple can be implemented within a database-centered tool using database triggers to recognize events associated with constraints and dependencies and to respond by sending various types of messages to the appropriate developers.

In this section we provide a high level description of Ripple, our design representation framework. Ripple embraces:

- the *definition* of the stages and associated activities and work products from each life cycle in the integrated development effort;
- the *definition* of dependencies, constraints, relationships;
- the triggers and messages for *enforcement* of constraints and dependencies between and within the two development life cycles; and

- the *implementation* of a constraint-based, database-centered tool that works within this framework to support the concepts in the above bullets.

13.4.1 Constraint-based Database-centered Framework

Ripple is a constraint-based framework that supports the complementary existence of the SE and UE development roles. A constraint is a "relation that must be maintained" (Borning and Duisberg, 1986). Such relations are generally enforced by "delegating to the constraints solver the task to satisfy them automatically" (Kwaiter et al., 1998). In other words, a constraint-based system is one that automatically updates a predefined set of relations and dependencies between different entities when a change occurs in one or more of such entities. Constraint-based systems were traditionally used to specify declaratively the relative layout of interface objects according to prespecified rules (Szekely and Myers, 1988). Some of the other important applications for constraint-based systems include:

- specification of relations (constraints) among the user interface objects that should be maintained upon resizing a given UI window (Mugridge et al., 1996; Chok and Marriott, 1995),
- visual representation of simulation algorithms (Ege, 1988),
- automatic updating of (to make consistent) multiple views representing the same data when the objects in one of the views is changed (Borning and Duisberg, 1986), and
- triggering of events based on changes made to objects in a dataset (Bharat and Hudson, 1995).

It is this last application of constraint-based systems that we focus on. Conceptually, this framework represents the various products of the shared design process in a single database with each of the SE and UE roles having two separate views to this single dataset. When any life cycle role changes or updates the database through their corresponding view, the system automatically triggers update messages or design reminders to all related or connected phases of the integrated development process. Such reminders or updates are propagated in our framework using *messages*.

13.4.2 Constraints and Dependencies Among Related Activities

When a new insight is gained into the system being development, or when something changes in either of the two life cycles, or when the developer roles needs to communicate with one another, the system triggers a message of a particular type to the related and connected phases of the design representation framework. Also, it should be noted that, constraints and relationships exist among activities and work products within each of the life cycles as well as those that cross over between the two life cycles. An example of such a relationship on the UE side is when a key task is identified in task analysis, that task should be flagged for consideration for a design scenario and a benchmark task later in the life cycle.

13.4.3 Messages and Triggering Agents

Messages are the communication and synchronization agents in Ripple. They convey the ripple effects of change, design insights, notes, and observations made during a particular development activity on future design stages. The five types of messages are discussed below:

For Your Information Message. This type of message informs the software engineers and usability engineers about the completion of a particular activity or phase in the life cycle and shows the link where the relevant products of this development stage are located. The developers in the other life cycle or developers at a different stage of the project (within the same life cycle) can use this link to view the product (artefacts). This message is generally used when the type of communication is purely informational and no corresponding action is necessarily required. For example, when the usability engineers complete the initial screen layouts or the derivation of the conceptual metaphor for the interaction design, they can send this type of message to the software engineers to peruse. Another example for this type of a message is when the usability engineer informs the software engineer about the completion of the screen design so that the user interface can be implemented by the functional core developers pending the summative evaluation.

Synchronize Activity Message. This type of message informs about the need for a joint activity by both the SE and UE roles. In other words, this message addresses the synchronization need for activities that require a combined presence of the two developer roles. For example, when the usability engineers plan an evaluation session, they can send this type of message to the software engineers to request them to be present (to help argue the case for required changes in the user interface when the SE role sees the users having problems). Similarly, early systems analysis and ethnographic study activities that require joint presence can be arranged using this kind of message (to help identify the broader constraints of the project and get the overall context).

Consistency Check Message. This type of message is used to enforce the consistency of data objects in the database. This message informs the developers of the need to perform a consistency check on the two development roles' products. For example, when the software and the usability engineers complete the hierarchical task analysis and the functional decomposition, respectively, there is a need for a consistency check to see that every task in the HTA has a function or set of functions in the SE specifications, and vice-versa. In the object oriented development paradigm, this type of message can be initiated after the use case specifications phase in the SE life cycle or the usage scenario descriptions in the UE life cycle. Since these two stages of development concentrate on two aspects of the same issue: interaction between the system and user, there is a need to ensure that they are consistent. Another important example for the need for consistency is after the usability specifications phase in the UE life cycle and functional requirements in the SE life cycle. A consistency check message is required here to initiate an analysis that ensures that these specifications are

supportable by the functional core (and to discuss alternatives if not supportable or negotiate for middle ground). This type of a message is used to enforce such mandatory consistency checks.

Change Request Message. This type of message is used to inform the two developer roles of changes made in one part of the design and the potential effects of that change in that and other parts of the design. This is perhaps the most useful message in the development of interactive systems because of the potential for constant and frequent changes in the products during the development life cycles. As an example, this message can be used when a new task is identified by the UE role, and that new addition should be communicated to other development activities within the UE role and to the SE role. Upon the receipt of the message by the SE role, efforts can be made to incorporate the necessary functions in the functional specifications to support the corresponding task. These updates in the functional specifications, in turn, can trigger changes in various dependent stages' products in the integrated life cycle.

Response to a Change Request Message. A response to a change request message is sent by developers to acknowledge a change request message. Because the control of decisions to make changes or not ultimately resides with the developer roles, one possible response to change could be 'change request considered fully, but declined' with an explanation or note, for the record, saying why the request was declined.

Change-in-response-to-change Message. The need for this kind of message is to avoid endless loops of messages due to cycles in the graph of relations. Suppose a relation 'R' exists from function A to B ($A \xrightarrow{R} B$). For example, if A is task analysis in UE and B is functional decomposition in SE, then R is a relation meaning that changes in task analysis (A) require related changes to be considered in functional decomposition (B). The relation R is expressed as a message that is sent whenever a change occurs in A, informing the developer role in charge of B to consider changing B accordingly. These dependency relations are often symmetric (i.e. changes in functional decomposition also require consideration of changes if task analysis), so that a development process could have both $A \xrightarrow{R} B$ and $B \xrightarrow{R} A$ among its dependencies. This could lead to endless loops; a change in A triggers a change in B, which in turn triggers a change in A, and so on. To break these cycles we introduce a new message type called the 'change-in-response-to-change' message. A change made in B due to a change request message from A would return a change-in-response-to-change message that would not require further changes in A.

Design Reminders. Design reminders are a type of message used to record design reminders for future development stages. This type of message could be used as a reminder to handle something later (such as a feature that has been temporarily stubbed in the current activity, say, in the prototype stage), where there is no time presently to consider it. For example, while developing a calendar management system the developers may "hard wire" the alarm feature to go off 10 minutes before each

appointment, but want a reminder to fix the design later by allowing the user to set the lead time for the alarm.

Framework generated messages are formalized in terms of the life cycle activities of both the development processes and the communication/dependency relationships identified. The database implementation of our framework will automatically generate the consistency and change messages.

In addition to messages automatically generated by Ripple due to pre-defined constraints, messages can be sent by developers for design reminders and 'for your information' purposes. For example, the developers can specify when they would like to send a "for your information" message to the other groups. Developers can send a "for your information" message to the other developer role to let them know work is being done on a certain part of the design, even though the current state of work is not ready for sharing yet. On the other hand, if they come across new insights or new additions to the project, they can send a change request message.

13.4.4 The Ripple Framework

Consider the following schematic (Figure 13.4) in which the two development processes are shown, simplified as three stages in the life cycles: 1, 2, and 3 for UE stages and A, B, and C for SE stages. The messages from each phase are labeled using the <development stage ID>$_{<messagecounter>}$. The different types of communication or dependency relations are marked using different line widths and styles.

In the example shown in Figure 13.4, the UE cycle triggers three messages in stage one: $M1_1$, $M1_2$, and $M1_3$. Similarly, SE cycle triggers MA_1 and $MA_2$and so on.

A developer using Ripple to work on a particular life cycle stage, will have a list of waiting messages from other phases in the SE and UE cycles. These waiting lists are shown on the far right and left sides of the figure as 'message queues' at each phase. These messages can be reminders from

previous stages or constraints or change effects from other stages. Depending on the type of message, the developer responds accordingly.

When developers make changes to existing documents in the design repository, those changes, in turn, trigger ripples of new messages. The history of ripple messages can support traceability of changes in the overall framework, and includes a rationale for the change and details of who initiated the change and when.

Ripple uses a 'score card' approach to list the status of each phase of the development life cycle, showing which stages are bottlenecks and which stages need the most attention.

13.5 CONTRIBUTIONS

13.5.1 Activity Awareness and Life Cycle Independence

Using Ripple (Figure 13.5), each developer role has significant insights into their own and the other's life cycle status, activities, the iteration of activities, the timeline, techniques employed or yet to be employed, the artefacts generated or yet to be generated, and the mappings between the two life cycles if present. The view of each role shows

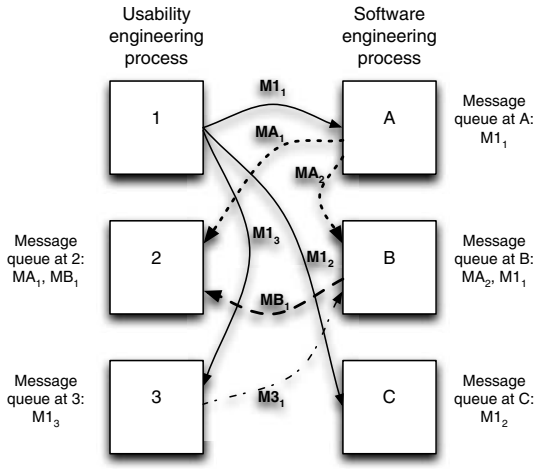

Figure 13.4 Message passing and accumulation in the integrated process framework

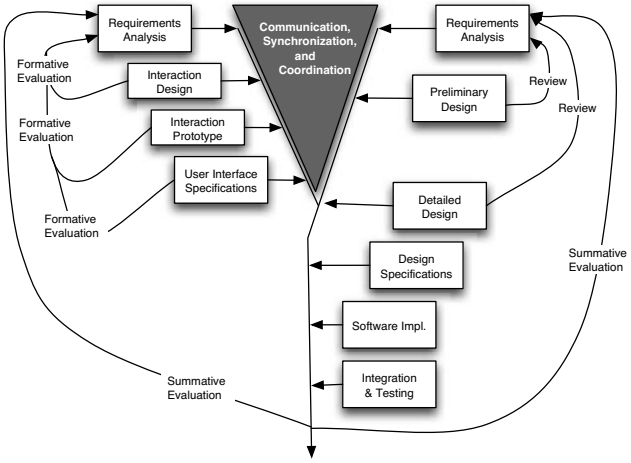

Figure 13.5 Ripple: Framework with communication/coordination

only those activities that are relevant to that role. Each role views the shared design representation through its own filters (Figure 13.6). For example, the software engineers see only the software implications that result from the previously mentioned iterativeness in UE, but not the techniques used or the procedure followed. Similarly, if software engineers need iteration to try out different algorithms for functionality, it would not affect the usability life cycle. Therefore, the process of iteration is shielded from the other role, only functionality changes are viewable through the UE filter. Each role can also contribute to its own part of the life cycle; Ripple allows each role to see a single set of design results, but through its own filter. Ripple emphasizes the placement of these connections and communication more on product design and less on development activities. This type of 'filter' acts as a layer of insulation, between the two processes, i.e. Ripple helps isolate the parts of the development processes for one role that are not a concern for the other role. This insulation needs to be concrete enough to serve the purposes, but not over specified so as to restrict the software design that will implement the user interface functionality. This prevents debates and needless concerns emanating from the use of specialized techniques. Because Ripple does not merge, but coordinates, the two development processes, life cycle roles from one process need not know the language, terminology, and techniques of the other, and therefore can function pseudo-independently.

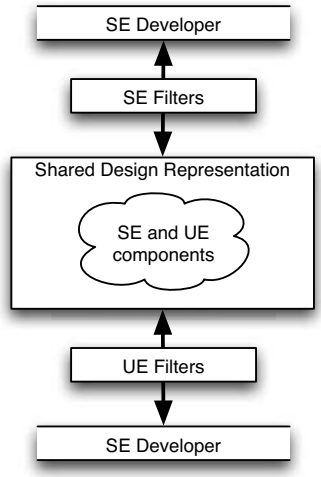

Figure 13.6 Shared design representation

13.5.2 User Interface and Functional Core Communication Layer

Ripple advocates the need for the two life cycle roles to specify a common communication layer between the user interface and the functional core parts. This layer is

similar to the specification of the communication between the model and the other two parts (view and controller) in the 'model view controller' (MVC) architecture (Krasner and Pope, 1988). This communication layer describes the semantics and the constraints of each life cycle's parts. For example, the usability engineer can specify that an undo operation should be supported at a particular part of the user interface, and that in the event of an undo operation being invoked by the user, a predetermined set of actions must be performed by the functional core. This type of communication layer specification will be recorded by our design representation framework, and allows the software engineers to proceed with the design by choosing a software architecture that supports the undo operation (Bass and John, 2001b). How the undo operation is shown on the user interface does not affect the SE activities. This type of early specification of a common communication layer by the two life cycles minimizes the possibility of change on the two life cycle activities. However, this common communication layer specification can be difficult to specify and might change with every iteration. These changes should be made and take into account the implications that such a change will have on the already completed activities, and/or the ones planned for the future.

13.5.3 Coordination of Life Cycle Activities

Ripple coordinates schedules and specifies the various activities that have commonalities within the two life cycle processes. For such activities, Ripple indicates where and when those activities should be performed, who the involved stakeholders are, and communicates this information to the two groups. For example, if the schedule says it is time for usability engineers to visit the clients/users for ethnographic analysis, Ripple automatically alerts the software engineers and prompts them to consider joining the usability team and to coordinate the SE's user related activities such as requirements analysis, etc.

13.5.4 Communication Between Development Roles

Another important contribution of Ripple is the facilitation of communication between the two roles. Communication between the two roles takes place at different levels during the development life cycle. The three main levels in any development effort are: requirements analysis, architecture analysis, and design analysis. Each of these stages results in a set of different artefacts based on the life cycle. Ripple has the functionality to communicate (using messages) these requirements between the two life cycles. For example, at the end of UE task analysis the usability group enters the task specifications into the design representation framework and the SE group can view these specifications to guide their functional decomposition activities. At the end of such an activity, the SE group enters their functional specifications into Ripple for the usability people to cross check. This communication also helps in minimizing the effects of change and the costs to fix these changes. By communicating the documents at the end of each stage, the potential for identifying errors or incompatibilities early in the process increases compared to waiting till the usability specifications stage. This early detection of mismatches is important because the cost to fix an error in the requirements that is detected in the requirements stage itself is typically four times

less than fixing it in the integration phase and 100 times less than fixing it in the maintenance stage (Boehm, 1981).

13.5.5 Constraints, Dependencies and Provision for Change

Ripple incorporates automatic mapping features, which will map the SE and UE part of the overall design based on their dependencies on each other. Recall the example of the many-to-many mapping between the tasks on the user interface side, the functions on the functional side, and how Ripple will automatically alert the software group about the missing function(s) and vice versa. So, when the software engineer tries to view the latest task addition, a description that clearly specifies what the task does and what the function should do to make that task possible, is provided. This way the developers can check the dependencies at regular time intervals to see that all the tasks have functions and vice versa. It also helps ensure that there are no 'dangling' tasks or functions that turn up as surprises when the two roles finally do get together.

13.6 POTENTIAL DOWNSIDES OF RIPPLE

Ripple has the following downsides due to the various overheads and additional tasks that arise because of the coordination of the two life cycles:

- Increase in the overall software development life cycle;
- Additional effort required by the roles in each life cycle for document creation and entry into the design representation framework;
- Additional effort required for coordination of various activities and schedules;
- Need for stricter verification process than conventional processes to enforce the various synchronization checkpoints during the development effort; and
- Resource overhead to carry out all the above mentioned drawbacks.

13.7 CURRENT STATUS

Ripple is a work-in-progress. We have currently identified many different dependencies and constraints within the UE life cycle. We will do a similar mapping on the SE life cycle and then on an integrated framework. We are currently working toward representing the products of a development effort in a database system. We have yet to implement the triggers and constraints. We also intend to test the framework using a project in simulated real life settings. We plan to do this by offering the SE and UE courses in an academic semester and having half the teams use the current practices and the other half use our framework.

Acknowledgements

The authors would like to thank the reviewers and editors for their insightful comments and feedback. This feedback helped us address some of the issues we overlooked in our early versions of this chapter.