

## The Role of Iteration in the Early Phases of a Traceability-Oriented Software Development Process

Borislava I. Simidchieva  
Stefan C. Christov  
Department of Computer Science  
State University of New York, College at Brockport  
350 New Campus Drive  
Brockport, NY 14420. USA

Faculty Advisors: Sandeep Mitra, Ph.D. and Thomas A. Bullinger<sup>1</sup>

### Abstract

According to the recent Standish Group's CHAOS reports, the "software crisis" persists. The Waterfall Model correctly identified the phases of software development (requirements capture, problem analysis, design, implementation, and testing), but did not accommodate *iteration* between phases. Many later methodologies attempt to facilitate iteration (e.g., the Spiral Model, RUP, etc.), including Agile methodologies like eXtreme Programming (XP), which claim to "embrace" iteration by rework primarily at the code level. However, techniques for embracing iteration during earlier phases, focusing on artifacts created only during these phases, remain relatively untouched. Few methodologies explicitly demonstrate how iteration may actually be achieved in *specific types of applications*. This paper reviews existing iteration strategies, noting their focus on the design/implementation level – principally, "code refactoring". It then explores the Software Engineering Effectiveness Model, (SEEM<sup>TM</sup>) and its unique focus on ensuring accurate and obvious *traceability*. SEEM<sup>TM</sup> advocates using certain principles to create a *precise, unambiguous* analysis model for applications that have a GUI front-end and a database back-end. It then uses *standard design patterns* to map from the technology-independent analysis model to the implementation technology-specific design/implementation models. The paper demonstrates this process using a detailed case study of a "Journal Maintenance System". It then focuses on techniques for evaluating the *quality* of the analysis and design models. A high-quality analysis model should enable the programmer to systematically map it to later phase models. If this mapping is not easily achievable, the paper shows how a developer can reexamine the analysis model, attempting to improve it. Thus, the developer effectively iterates at the analysis/design phases, improving quality and correctness early.

**Keywords:** Software Development, Formal Methodologies, Iteration, Traceability, Software Engineering

### 1. Introduction

Since the inception of software engineering, many methodologies have been proposed to standardize and facilitate the development of software. In 1970, Dr. Winston W. Royce proposed the "waterfall model"<sup>2</sup>, in which he identified the main phases of software development as requirements capture, problem analysis, design, implementation, and testing. Although Royce himself believed software development is an iterative process, his "waterfall model" largely established itself as a milestone approach through which a developer could progress to a later phase only after completing the earlier phase. This made iteration, or going back to revisit artifacts at previous stages based on new discoveries, very cumbersome, thus unnecessarily limiting the developers and resulting in software that is hard to extend or maintain.

Software engineering is an innately iterative process, and many later methodologies have been developed in an attempt to rectify the "waterfall model" deficiency by accommodating and facilitating iteration. While these methodologies preserve all the key phases that Royce identified, they also strive to provide the developer with

techniques to achieve effective iteration--by allowing faster development and tangible results, and also easily accommodating changes due to customer feedback or a newly discovered flaw in the design. By going through the phases quickly the developer can achieve a working prototype, which enables him or her to solicit feedback from the customer and assure that the initial requirements are being met. Going through this process repeatedly results in a final deliverable product that satisfies the customer's requirements.

Some of the methodologies that concentrate on iteration are the Rational Unified Process (RUP)<sup>3</sup>, eXtreme Programming (XP)<sup>4</sup>, and the Spiral Model<sup>5</sup>. RUP is a prescriptive approach, which gained significant popularity largely due to its introduction of the Unified Modeling Language (UML) and its adoption by IBM. RUP also provides abstract guidelines on how iteration is achieved. XP, on the other hand, has been widely discussed as an Agile methodology; unlike RUP and most other prescriptive approaches, XP underemphasizes the importance of explicit analysis and design artifacts. XP concentrates on quickly developing a working system and then incrementally augmenting it while performing extensive testing. Thus, it claims to *embrace* iteration, but it primarily concentrates on working at the code level.

Though several of the above methodologies suggest that they facilitate iteration, they do not provide explicit guidelines on how to achieve this iteration. Moreover, most of them concentrate on iteration at the code level, or rework/refactoring of code, ignoring iteration at higher levels of abstraction. Fairley and Willshire<sup>6</sup> talk about the different scenarios in which rework occurs; they identify three kinds of rework: evolutionary (change in the requirements), retrospective (improving the quality of the product), and corrective (aims to fix discovered flaws). They conclude that most rework is retrospective or corrective in nature, and it is thus *avoidable* had the design been improved in an earlier stage. Although they discuss rework in detail, no guidelines or example case studies are provided on how to achieve high-level iteration and rework in developing an actual application.

In recent years, XP and the family of Agile methodologies have received much attention because of their promise of fast development of deliverable code and adaptability. Agile methodologies dispense with many of the traditional development phases and concentrate on the design and code levels. This may be beneficial in the case of smaller projects and programmers with outstanding abilities, but a large project, which requires many developers, would suffer from such an approach. Code is the only deliverable required when using Agile methodologies. After an initial basic prototype is developed and feedback is solicited from the customer, the system is built very quickly by gradually augmenting the initial code. This development approach produces a complete prototype very fast, but very often requires extensive debugging and correctional rework. While the augmentation is taking place, the system is also tested for cohesion and coupling using standard software engineering metrics and improved if necessary. Measuring code quality using such metrics and techniques for improving the code are all very subjective and require a high level of experience and skills.

Proponents of the Agile approach expect these methodologies to be executed perfectly because the programmers have outstanding abilities and can easily analyze, correct, and improve design by looking at thousands of lines of code without needing high-level, abstract artifacts: this is the process of *code refactoring*. Excessive reliance on the quality and experience of the programmers to effectively execute this process can be dangerous. Since Agile methodologies are designed to be very flexible and adaptable, the use of analysis and design artifacts, while not outlawed, is not required. As a result, these artifacts often get omitted. There are many resources available that indicate how to perform iteration in the context of Agile methodologies<sup>7</sup>. However, because of the emphasis on code, most of these guidelines facilitate *code* refactoring, not iteration at higher levels of abstraction.

## 2. Methodology

The Software Engineering Effectiveness Model™ (SEEM™)<sup>1</sup> is a prescriptive approach, which requires the creation of concrete artifacts at different stages, but also seeks to provide the flexibility and ease of iteration that Agile methodologies boast. SEEM can be customized to suit the size and complexity of the problem to be solved. The SEEM methodology relies on some fundamental principles:

### 2.1. Problem domain model

During the requirements capture and analysis phase, a high-quality problem model must be created. This model is fully technology-independent, and the emphasis is on the features of the *problem*, not the technology used for the *solution*. This ensures that all aspects of the problem are addressed and the resulting solution can be implemented using any suitable technology. This model must be precise and unambiguous to the greatest extent possible. Consequently SEEM emphasizes concrete artifacts, using notation such as the Unified Modeling Language (UML).

### 2.2. Traceability and standard mapping techniques

The artifacts from early phases are mapped to more detailed artifacts in later phases by using standard techniques like software design patterns and object-oriented coding idioms. By applying these standardized mapping techniques to the problem domain (analysis) model, a gradual, *traceable*<sup>8</sup> transition can be achieved to an implementable form of this model, as the design, detailed design and coding phases progress. These standardized mapping techniques must be appropriately identified by the programmer based on her understanding of the technology of the implementation environment.

### 2.3. Progressive learning

If the abovementioned mapping techniques cannot be easily applied, or if their application results in artifacts of subpar quality at a later phase (e.g., the design is of poor quality after no apparent problems at the analysis stage), then the original problem model was not drawn up correctly, or drawn up such that it is not of the best quality. When such a situation is identified, the developer should revisit the earlier stage with the newly acquired knowledge and improve the initial model. This iteration enables the creation of a high-quality, cohesive design at all stages.

This iteration is a type of “refactoring”. But unlike Agile methodologies, SEEM does not confine refactoring to the code level; iteration is facilitated and encouraged at all phases of development. SEEM enables the developer to check the quality of early stage models. Thus, a model with good cohesion and desired levels of low coupling can be achieved even at the analysis and design phases of the development effort. Appropriate techniques to achieve this will be described in this paper.

## 3. Case Study

To demonstrate the effectiveness of this methodology, a case study of a “Journaling” system was developed, following the standard three-layer architecture design: a graphic user interface front-end, business logic, and a database back-end (GUI-on-database, or GOD system). This system allows users to manage “Journals” that record activities and times by personnel working on software development projects. A Journal in the system consists of a collection of “Sessions”, with each session associated with a date, time range and personnel names. Each Session has a collection of “Entries” associated with it. The system has several main behaviors: create a new journal, open an existing journal, add a session to a journal, add an entry to a session, view or edit existing entries, and calculate time spent on the project. Because of length limitations, this paper only focuses on the “Add Entry” functionality.

## 4. Data

### 4.1. The analysis model

At the analysis stage, the main behaviors of the system, which result in discrete, observable goals for the user, are identified and *user stories* are written using step-by-step descriptions in carefully structured English. These artifacts are developed in a completely technology-independent manner. Descriptions follow the following basic sentence structure: “Who does What to Whom”. Later in the lifecycle, the *who* and *whom* will translate into objects, and the *what* will likely become a method. At this point, however, no such technology is introduced.

To facilitate the construction of the user stories, SEEM advocates considering how the transaction would be carried out in a computer-less world: by Clerks, Secretaries, Tellers, etc. who would interact with the customer and carry out the required actions or update the appropriate repositories of “index cards” in catalogs where data is stored. This person who facilitates the interaction will later become an agent or entity at the system boundary. The first version of the user story for “Add New Entry” is shown in Figure 1.

Despite the structure imposed on user stories, natural language is inherently ambiguous. To rectify this, a UML sequence diagram is constructed, which exactly corresponds to the workflow depicted in the user story and further clarifies it. Figure 2 shows the sequence diagram corresponding to the Add New Entry user story.

To complete the analysis model, some more artifacts are necessary, including the Participant-Responsibility-Collaboration (PRC)<sup>9</sup> cards, which are created from the sequence diagrams. For each participant (entity) identified, these cards list the responsibilities (easily identified as incoming requests in the sequence diagrams) and the collaborators for each responsibility (the entities, to which there are outgoing messages as a result of an incoming request). The PRC cards allow the developer to easily evaluate the distribution of responsibilities among entities.

### 4.2. Mapping the analysis model to design and implementation models

SEEM uses standard design patterns to facilitate the generation of design artifacts that are fully traceable to the analysis artifacts. Since the Journaling system follows the three-layer architecture, and is a GOD system, SEEM suggests using the Model-View-Controller<sup>10</sup> (MVC) design pattern for the front-end. For such systems, the

developer begins design by identifying the entity at the *system boundary*. This entity will interact with the user and therefore will need to provide a GUI.

<b>User Story Name:</b>	Create a New Entry.
<b>Description:</b>	
Create a new entry within an existing Session within an existing Journal.	
<b>Stakeholders:</b>	
Journal Keeper, Manager	
<b>Preconditions:</b>	
<ol style="list-style-type: none"> <li>1. There must be a session to which the entry can be added.</li> <li>2. The Journal they want to add the entry to must already be open.</li> </ol>	
<b>Workflow:</b>	
<ol style="list-style-type: none"> <li>1. The Journal Keeper indicates that he wants to work on an existing Journal</li> <li>2. The Secretary asks the user for the name of the Journal that needs to be open</li> <li>3. The Journal Keeper supplies the Secretary with the name.</li> <li>4. The Journal Keeper tells the Secretary they want to create a new entry in the Journal they are working on.</li> <li>5. The Secretary goes to the Journal cabinet and picks up the notebook whose label corresponds to the supplied name.</li> <li>6. The Secretary asks the Journal Keeper for search criteria to find the session in which the user would like to create the new entry.</li> <li>7. The Journal keeper provides session start date, start time and optional end date.</li> <li>8. The Secretary looks through the Journal and gives to the Journal Keeper a list with all the sessions that match the search criteria.</li> <li>9. The Journal Keeper chooses a session in which he/she wants to add and entry.</li> <li>10. The Secretary asks the Journal Keeper for the type of entry he/she would like to add.</li> <li>11. The Journal Keeper tells the Secretary the type of entry they want to add and what they did.</li> <li>12. The Secretary writes down the one letter notation and then a description of what has been done as the last entry in the last session of the Journal.</li> </ol>	
<b>Results:</b>	
<ol style="list-style-type: none"> <li>1. A new entry is added to the most recent session of the Journal.</li> </ol>	
<b>Alternates:</b>	
<ol style="list-style-type: none"> <li>1. There is no existing session in the Journal.</li> <li>2. The Journal Keeper provides a non-existent entry type.</li> </ol>	
<b>Entities Involved:</b>	
Journal Keeper, Secretary, Journal, Session, Entry	

Figure 1. “Add New Entry” user story.

The developer needs to identify which is the first front-end entity in the automated part of the workflow. This is accomplished by inspecting the analysis sequence diagrams and identifying the entity that receives incoming messages from the human user. In Figure 2 this is the Secretary.

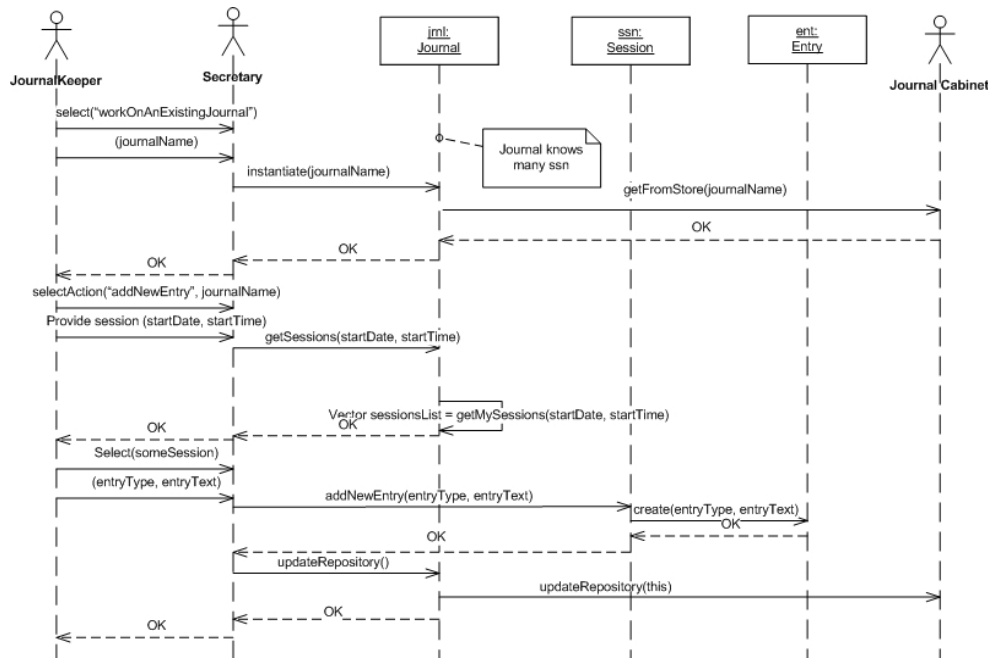


Figure 2. “Add New Entry” analysis sequence diagram.

As a result, the developer notes that the Secretary will be automated, and the MVC design pattern is applied to introduce a GUI between the outside user and the Secretary *object*. The automated Secretary object must ensure that it provides the exact functionality required of it by Figure 2, and other similar diagrams.

Since the Secretary object at the design level provides the functionality required of it from the analysis, and, *in addition*, provides the GUI, it may be concluded that the analysis level Secretary entity “morphed” into multiple entities at the design level. MVC is used to achieve the “morphing”: the “original” Secretary becomes the model entity of MVC, and has several different views associated with it. Each view displays different information to the user and also enables the invocation of different interactions between the system and the external user. The Secretary entity must incorporate additional functionality (beyond what was required by the analysis model) to handle the MVC architecture: it must know *how* and *when* to create and display each of its views. Figure 3 is a design level sequence diagram, which is a direct mapping from the analysis sequence diagram, accomplished with the help of the standard MVC design pattern.

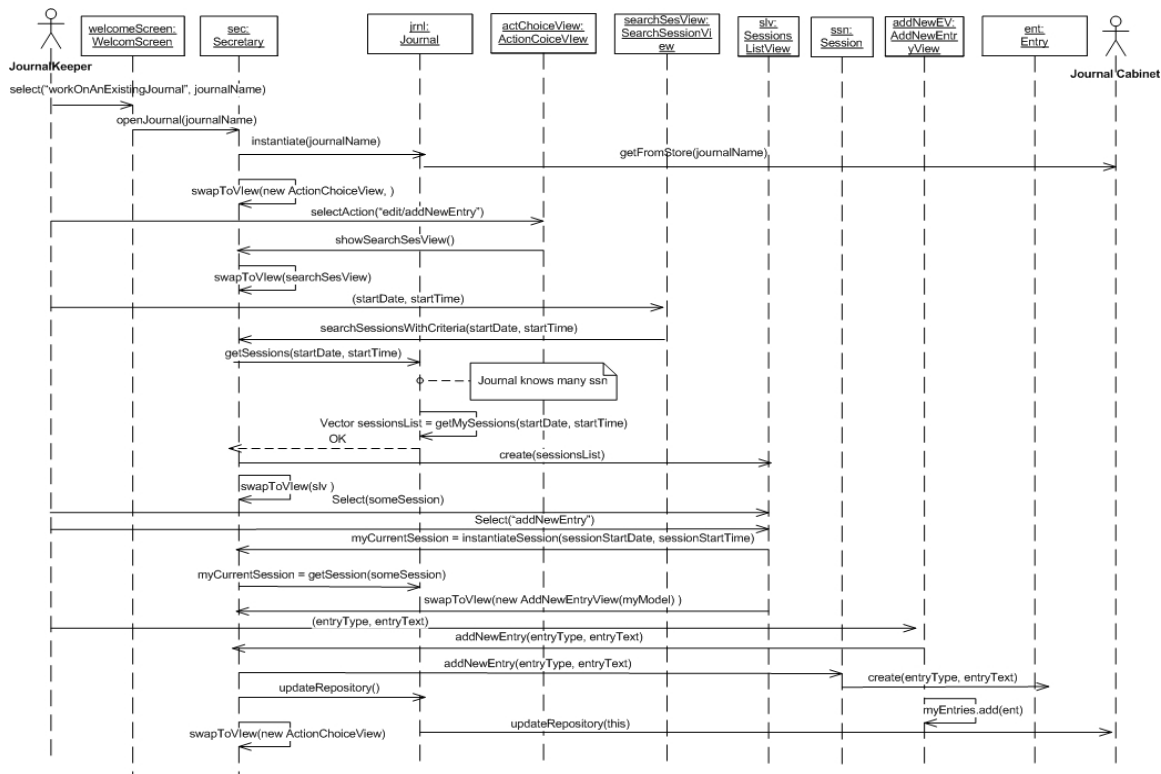


Figure 3. “Add New Entry” design sequence diagram.

At this point, with additional entities introduced, it is advisable to check for coupling and cohesion. A good way of checking these metrics is by considering the balance of responsibilities among the design entities. As can be seen from Figure 3, the Secretary entity is overloaded while the rest of the entities have considerably fewer responsibilities compared to it. In our case study, the PRC card for the Secretary contained significantly more responsibilities than any of the other entities’ PRC cards.

While formal quantitative cohesion measurement techniques were not used yet, an informal analysis indicated that as a result of assuming all these different responsibilities, the Secretary becomes an “overloaded”, monolithic entity. This design is potentially hazardous because when additional functionality is added to the system, the Secretary will grow in an uncontrolled way, acquiring more responsibilities that do not naturally pertain to it, and increasing the level of coupling even more.

SEEM considers making such discoveries to be an intrinsic part of the software development process. It is hard for developers to account for low-level details at the high levels of abstraction when they are trying to capture and model the main behaviors of the system precisely. In order to rectify the discovered deficiency in the model, the developer needs to stop at this point and reconsider the early artifacts that led to the problematic design. With the acquired insight about the system, the developer can now revisit the early analysis artifacts and try to fix the

problem there, instead of continuing with imperfect design and risk encountering problems after the program has been implemented.

While developers may be tempted to fix the issue at the design level and never revisit the analysis artifacts, this must be avoided in the interest of maintaining traceability. The problem resulted from the fact that the creation of the analysis model is not an exact science and largely depends on the experience and knowledge of the developer. Any flaws may not be apparent at this high level of abstraction, but once they have been revealed at the design level, they must be remedied at both levels. The design artifacts are an exact mapping of the analysis model; therefore the error must have occurred at analysis.

### 4.3. Analysis improvements after iteration

The revised analysis sequence diagram for the “Add New Entry” user story is shown in Figure 4.

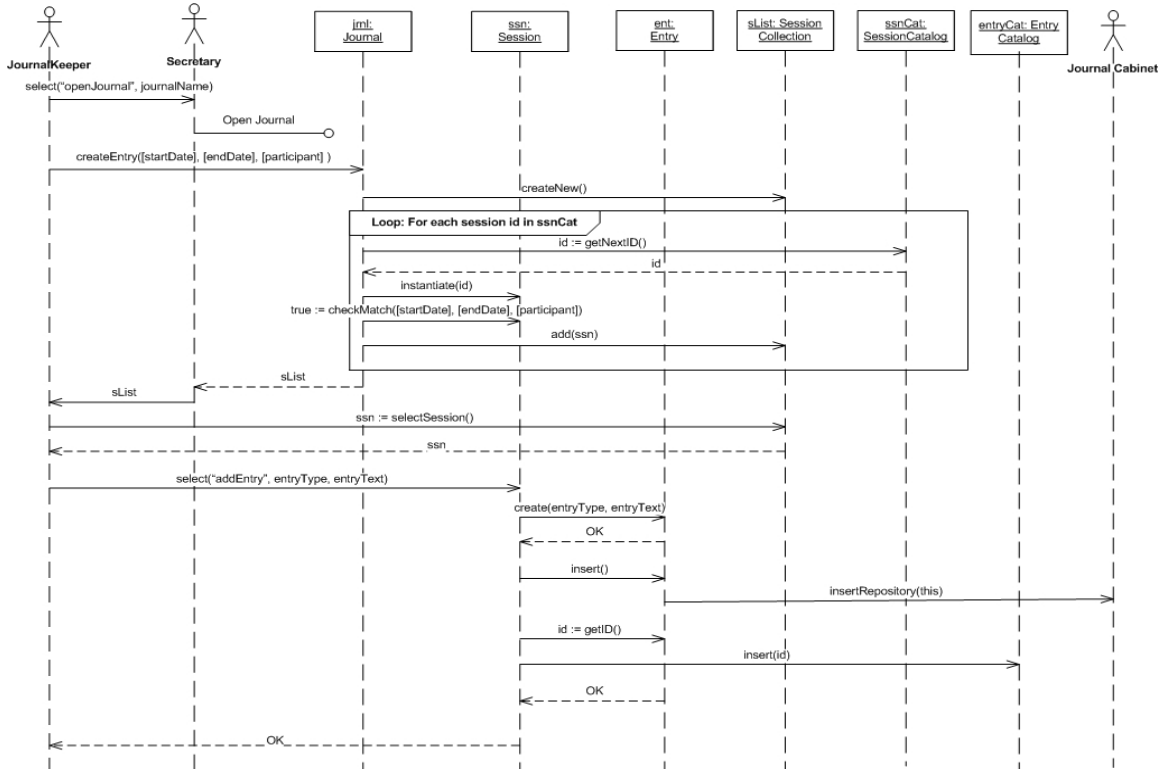


Figure 4. “Add New Entry”, revised analysis sequence diagram

The original analysis model was created by keeping a “smart human” Secretary who does all the work. After some more insight into the system is acquired following design, SEEM suggests the original totally manual workflow, with one “smart human” entity and other inanimate entities, be modified. A technique to consider is that some of the inanimate entities become “smart” and they are now able to respond to commands from the user. Thus, it is not just the Secretary who is smart (because she is the only human), but the Journal folder, the Session and Entry index cards, etc. all become “smart” to some extent. Using this idea, the developer can redo the user story workflow and sequence diagram to send human interactions not just to the Secretary, but to other entities as well. This approach accomplishes a more balanced distribution of responsibilities that will eventually lead to more cohesive and less coupled code when standard patterns are used to transition from high level of abstraction to a lower one.

### 4.4. Design improvements after iteration

A new set of design sequence diagrams is derived based on the revisited analysis artifacts (Figure 5). The same standard design patterns, mainly MVC, are used again to map to design. As Figure 4 indicates, the Secretary is significantly relieved of its previous responsibilities because they are assumed by other entities such as the Journal, the Session, the Entry, etc. All these entities become MVC model entities and have associated views and controllers.

The Secretary is not monolithic anymore, since all new responsibilities will be assumed not by just the Secretary, but also by other entities as well (e.g. Journal, Session, etc.). A much more componentized design is obtained--

before a line of code is written! All the corrective rework has been done by looking at user story descriptions, sequence diagrams and PRC cards!

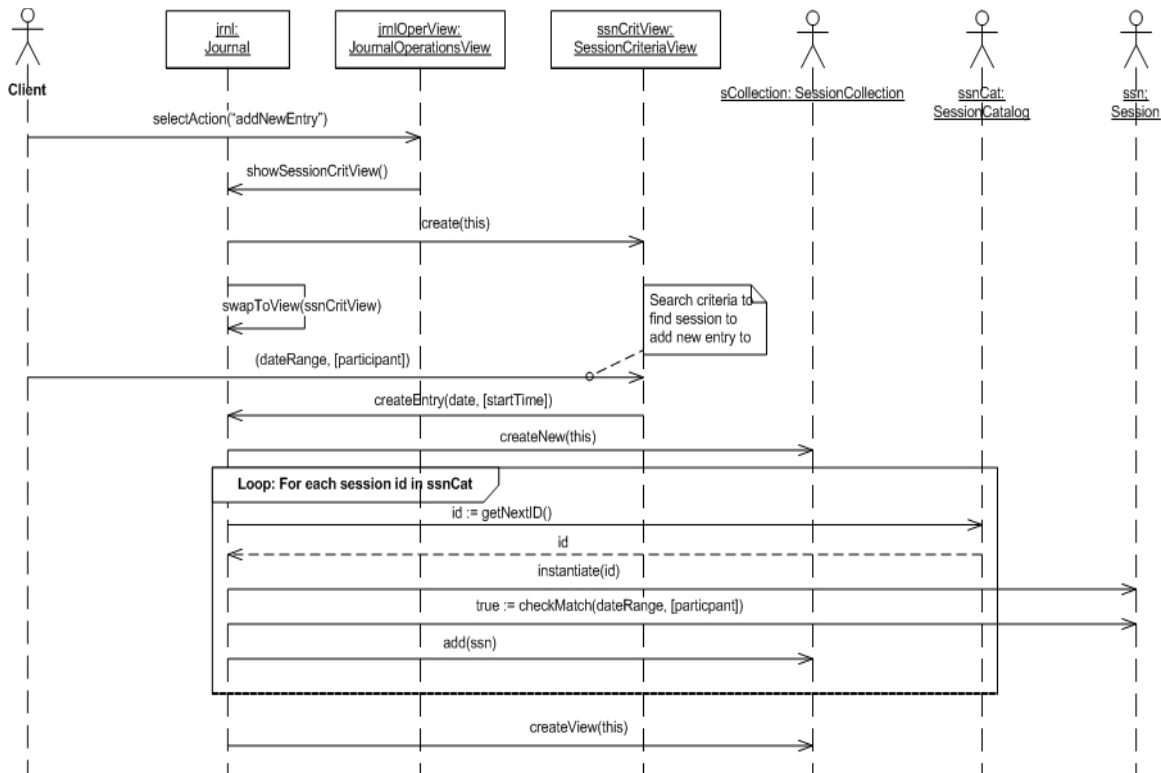


Figure 5. “Add New Entry”, revised design sequence diagram (for the Journal sub-system)

An interesting observation to make is that the old, monolithic Secretary is not even present at the design sequence diagram shown in Figure 5. Firstly, due to the fact that the Secretary’s entire set of responsibilities is spread among other entities, the Secretary is left with just a couple of basic responsibilities such as creating a journal, opening a journal and starting up the application. But it should be noted that based on the modified analysis artifacts, *several* design sequence diagrams are created, one for each of the analysis level entities (Secretary, Journal, Session, Entry, etc.). Each of those entities becomes a *subsystem* by itself in the new design model. The set of design sequence diagrams for each such subsystem is indicated by the set of incoming arrows for that particular subsystem shown in the sequence diagrams of the original analysis model.

Each such design level subsystem may have its own subsystems, thus facilitating hierarchical development as advocated by Grady Booch<sup>11</sup>, who noted it as one of the five basic attributes of complex systems. In Figure 5, a design sequence diagram for the Journal subsystem is shown, clearly delineating a view it manages. This view (called JournalOperationsView) enables the handling of the “createEntry” message sent to the Journal entity in Figure 4. The other ‘actors’ in this diagram are peer subsystems from the analysis level, each of which will have its own design artifacts (hence, they are shown as external actors). If the design sequence diagram for the peer subsystem Session is investigated, it will become apparent how it manages its own view to handle the “addEntry” message that would be sent to it (shown in Figure 4).

The original functionality of the system is still maintained because of clear traceability between the analysis and design artifacts. Simultaneously, the level of complexity is reduced by isolating more robust, and at the same time simpler, software *components*. The less complicated design diagrams that correspond to those components do not only lead to a more cohesive, componentized design, but they are also easier to read, understand, and eventually map to code.

## 5. Conclusion

Various techniques have been identified and outlined to allow the software engineer to study artifacts at each stage of the development process and evaluate how well they conform to established architectural principles. Moreover,

this evaluation does not need to be done at the code level; the artifacts can be evaluated and improved at each and every stage of the lifecycle, and through early iteration, the design can be improved before a single line of code is written.

The result is a much more componentized design with reduced development costs because of less rework and code refactoring. This early iteration would allow the software engineer to develop and produce a high-quality first version of the code, requiring less debugging. Furthermore, since the whole procedure is traceable, this software relies on sound architectural paradigms and principles. The resulting code is easier to maintain, and significantly easier to extend due to the distribution of responsibilities and the high cohesion and low coupling of the entities.

## 6. Acknowledgements

The authors wish to express their appreciation to Mr. Timothy Mullins for his help with the development of the graphical user interface and testing the implementation.

## References

---

- 1 ArchSynergy Ltd., 2500 Turk Hill Road, Victor, NY 14564
- 2 Winston W. Royce. 1970. Managing the Development of Large Software Systems: Concepts and Techniques. *WESCON Technical Papers* 14 (8): A/1-1 - A/1-9; Reprinted in 1987 in *Proceedings of the Ninth International Conference on Software Engineering* 9 (3): 328-338.
- 3 Kruchten, Philippe. 1999. *The Rational Unified Process*. (Reading, MA: Addison-Wesley).
- 4 Beck, Kent. 2000. *Extreme Programming Explained: Embrace Change*. (Reading, MA: Addison-Wesley).
- 5 Barry W. Boehm. 1988. A Spiral Model of Software Development and Enhancement. *IEEE Computer* 21 (5): 61-72.
- 6 Fairley, Richard E., and Mary Jane Willshire. 2005. Iterative Rework: The Good, the Bad, and the Ugly. *IEEE Computer* 38 (9): 34-41.
- 7 Martin, Robert C. 2003. *Agile software development: principles, patterns, and practices*. (Upper Saddle River, NJ: Prentice Hall).
- 8 Christov, Stefan C., Borislava I. Simidchieva. 2005. Full and Complete Traceability from the Inception through the Implementation Phase of Software Development. *19<sup>th</sup> National Conference for Undergraduate Research*, Lexington, VA.
- 9 Wirfs-Brock, Rebecca, and Alan McKean. 2003. *Object Design: Roles, Responsibilities, and Collaborations*. (Boston, MA: Addison-Wesley).
- 10 Buschmann, Frank. 1996. *Pattern-oriented software architecture: a system of patterns*. (Chichester, NY: Wiley).
- 11 Booch, Grady. 1993. *Object oriented design with applications*. 2nd ed. (Redwood City, CA: Benjamin/Cummings).