Good Software Architecting: Goals, Objects and Patterns

Lawrence Chung
Dept. of Computer Science
University of Texas at Dallas
chung@utdallas.edu

Sam Supakkul
Titat Software, LLC
Irving, Texas
ssupakkul@computer.org

Anna Yi
Dept. of Computer Science
University of Texas at Dallas
annayi@utdallas.edu

Abstract

Rather amazingly, software architecture has become the norm rather than an exception in just about any software development projects, even with less than a decade of recognition. With a brief historical perspective, this paper first describes some of the key tools and methodologies, that are available commercially and practiced widely as of today, for developing software architectures. This paper then presents how to develop "good" software architectures using goal-driven, object-oriented and pattern-based approach.

1. Introduction

Rather amazingly, software architecture has become the norm rather than an exception in just about any software development projects, even with less than a decade of recognition. With a brief historical perspective, this paper first describes some of the key tools and methodologies, that are available commercially and practiced widely as of today, for developing software architectures. This paper then presents how to develop "good" software architectures using goal-driven, object-oriented and pattern-based approach.

There are several design methods used by practitioners, but the one emerging as the standard is Unified Modeling Language (UML) [11]. UML is the product of the integration effort that combines Ivar Jacobson's OOSE, Rumbaugh's OMT, and Booch methods. It provides diagrams that can be for structural and behavioral modeling. To avoid re-inventing the wheels, software development projects reuse design patterns [9] that are proven to solve common recurring problems in software design instead of designing the system from scratch. However, the common practice often focuses only on the functional aspect of the system. With fierce competition in the marketplace, differentiating and winning factors for software products are often not the functional aspect, but rather the non-functional aspect, such as scalability, high availability, reliability, extendability, maintainability, etc. Therefore, there is increasing need for methods to address non-functional aspect of the system to help derive the optimum and desirable software architecture.

The NFR Framework [12] is a goal-oriented method that allows requirements engineers and software architects to fully capture non-functional requirements and systematically derive an optimum and desirable software architecture.

This paper presents a process to integrate UML, the NFR Framework, and design pattern to address both functional and non-functional aspects of the system and how to derive "good" or optimum and desirable software architecture. The steps in the process, based on an enhancement to the process presented in [13], are as follows:

1. Model functional requirements (FRs).
2. Post non-functional requirements (NFRs).
3. Refine the NFRs and prioritize them, taking into consideration any particular characteristics of the intended domain.
4. Integrate FR and NFR models
5. Consider architectural alternatives, at a macroscopic level, to meet the requirements stated, both functional and non-functional.
6. Consider design patterns, at a more microscopic level, to satisfice the architectural alternatives being considered. This consideration should be done in terms of the context and problems associated with each of the design patterns.
7. Analyze tradeoffs among the alternatives of architectures and their corresponding design patterns, in relation to the adaptability and whatever other NFRs stated, while carrying out impact analysis.
8. Select among the alternatives of architectures and their corresponding design patterns that best satisfice the adaptability and other NFRs in the context of the intended application domain.
9. Compose the selected design patterns into parts of the selected architectural design.

This paper uses Presence and Instant Messaging System (PIMS) as a case study to demonstrate how to use the process described above and how to integrate and use the three tools (UML, The NFR Framework, and design pattern). PIMS is an application that allows its users to subscribe to friends and colleagues (buddies) for their current communication status (e.g. Off-line, On-line, Busy, Available, etc.) and to send short real-time messages to each other (instant messages). Examples of this type of application are ICQ, AOL Instant Messaging (AIM), Microsoft Messenger, and Yahoo Messenger. The model of PIMS described in this paper is based on IETF RFC 2778 [1] and 2779 [2] specifications.

2. Functional and Non-Functional Requirement for PIMS

We first identify the functional and non-functional requirements for the system. The standard modeling language for capturing functional requirements that is widely accepted now in the industry is UML and its Use Case Modeling in particular. The Use Case diagram for PIMS is shown in Figure 1.

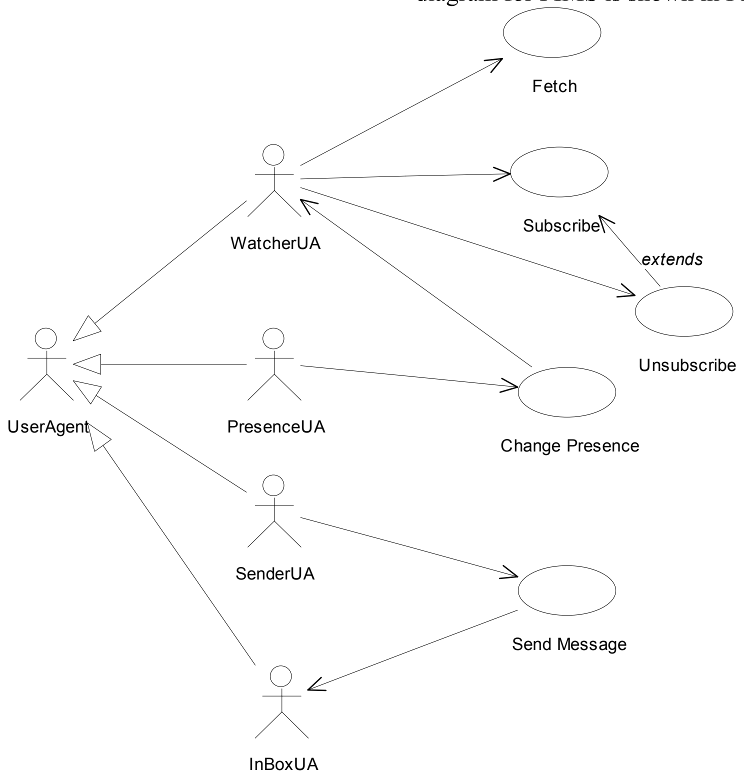

Figure 1. Use Case Diagram for Presence and Instant Messaging System (PIMS)

The external entities or actors of the system are UserAgent, WatcherUA, PresenceUA, SenderUA, and InBoxUA. UserAgent represents an application that provides the user with interface to view and manage his/her presence information, as well as sending and receiving instant messages. Examples of UserAgent are Microsoft MSN Messenger or Yahoo Messenger client applications. The UserAgent may play several roles when communicating with PIMS. Figure 2 shows the roles played by a UserAgent. It may play the role of WatcherUA to explicitly retrieve presence information of a buddy (Fetch Use Case) or subscribe for autonomous notification from PIMS when one of his/her buddy's presence status changes. The subscription may be cancelled (Unsubscribe Use Case). PresenceUA is the role that facilitates the presence information management. The user may change his/her presence information including presence status.

Figure 3 shows the UML state chart of presence status that shows the possible states and their transitions. For instance, once the user is on-line and recognized by the system, the user's presence status changes from Out of Contact to In Contact status. The Open sub-state (accepting messages) is assumed when entering In Contact state, which user may change from Open to Closed (not accepting messages).

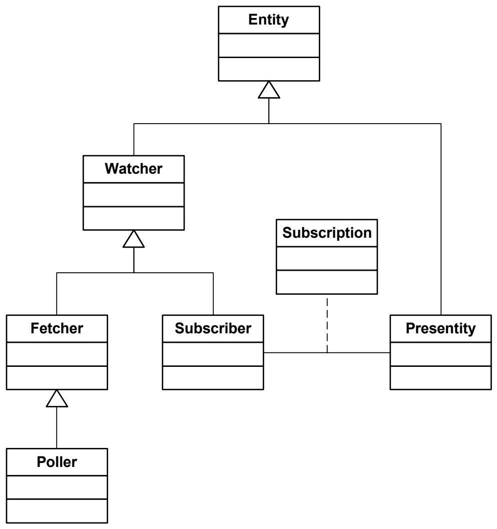

Figure 2. Roles Played by UserAgent

While in Opened status, user may provide more meaningful sub-status, which is implementation specific such as Busy, Available, Out to Lunch. When the user changes his presence status, PresenceUA sends the new status information to the system (Change Presence Use Case). The system then notifies all WatcherUA's that have subscribed for this user's presence status change notification. SenderUA allows user to send short messages to InBoxUA of the target users (Send Message Use Case). InBoxUA receives the messages and delivers them to the user. IETF further defines the role of Watcher user to Subscriber, Fetcher, and Poller. Subscriber is a user who has subscribed for autonomous notification from the PIMS system when one of his/her buddies changes presence status. The association between Subscriber and the Presentity (Presence + Entity) is maintained by the system as Subscription information. Fetcher is the Watcher who does not subscribe for notification, but may explicitly request for presence information of another user on demand. The Poller is the Fetcher who may automatically (via the UserAgent) request for presence information.

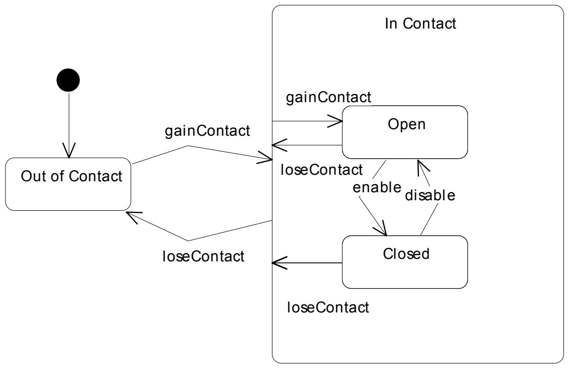

Figure 3. Presence Status State Chart

Figure 4 shows a SIG for simplified non-functional requirements. We can easily see that the PIMS must be scalable (Scalability[PIMS] Softgoal with very critical priority) [2], secured (Security[PIMS] Softgoal with critical priority) [2], and its code should be maintainable (Maintainability[PIMS] with neutral priority). We further decompose the Scalability Softgoal to reflect the definition made by Menasce [5] that a system is considered scalable when it is able to provide acceptable performance (Performance [PIMS] Softgoal) under increased load (Sustainability[PIMS], which is further decomposed to Sustainability[Increased Load, PIMS]).

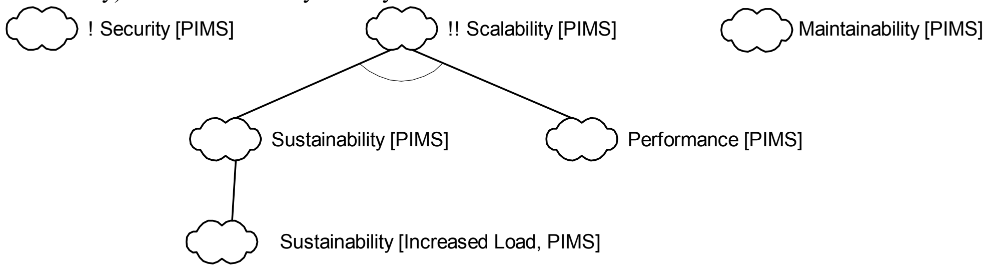

Figure 4. SIG for PIMS

3. Integrating FRs and NFRs

In general, requirements of a system can be categorized into either product related or process related. Functional requirements are mainly product related. They describe the functionality to be provided by the system. Non-functional requirements can be either product or processed related. For NFRs presented as Softgoals in Figure 4, scalability and security are considered product related NFRs as they are associated with certain functionality of the product, while maintainability is process related as it is the quality associated with the software engineering process and the overall quality of the code that implements the system. An example of product related requirements is that PIMS must be able to support large number of users (scalability NFR) that may be on-line exchanging presence status and instant messages (presence tracking and instant messaging delivery functionality). Therefore, it is necessary to associate product related FRs and NFRs to provide a comprehensive context for the design effort. We may use FR driven approach or NFR driven approach. Since majority of the practitioners are more familiar with UML, which is function oriented. We propose here a function oriented approach, more specifically Use Case driven approach. Dimitrov [7] associated timing performance specification with the Actor-Use Case association. We enhance the idea further by associating NFRs with any Use Case model elements, including Actor, Use Case, and Actor-Use Case association. We use Use Case diagram as the context for identifying relevant NFRs for each Use Case. In Figure 5, we say that Change Presence Use Case must provide speedy performance by associating Speed[Presence Notification] Softgoal (B) with PresenceUA-Change Presence Use Case association. We also say, by associating Security[Presence Info Transmission] Softgoal (C) with the association between Change Presence Use Case and PresenceUA/WatcherUA, that the communication between PresenceUA and WatcherUA for exchanging presence information must be secured. The diagram also indicates that the system must support large number of users with the association between Scalability[Presence Service with Large Number of Users] Softgoal (A) and UserAgent Actor. For this Change Presence Use Case, we have a complete view of both functional and non-functional aspects of the system that says the Change Presence function must provide speedy performance, secured transmission, and support large number of users. Using this Use Case driven, we have a comprehensive view of related FR and NFRs for guiding the design effort to achieve all the requirements for a given functionality. This process helps ensure that we have considered NFRs from all aspect related to a functionality.

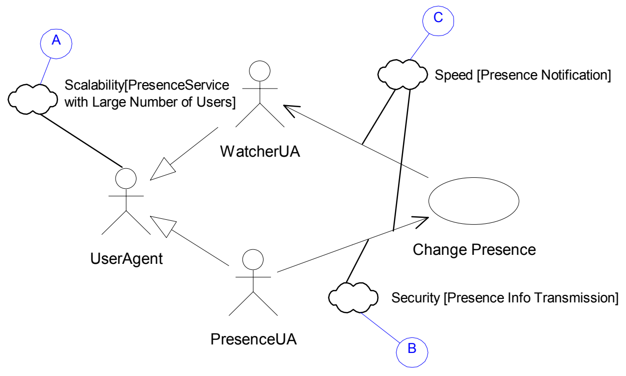
A
Scalability[PresenceService with Large Number of Users]
C
Speed [Presence Notification]
WatcherUA
UserAgent
Change Presence
PresenceUA
Security [Presence Info Transmission]
B

Figure 5. FRs and NFRs Integration for Change Presence Use Case

We repeat this Use Case driven NFR identification process with all other Use Cases. Once all NFRs have been identified in the context of the Use Cases, we integrate them into the original SIG to give a comprehensive view for all NFRs as with the Use Case model does for FRs. During this integration process, we could either decompose the top Softgoals we previously defined to sub-Softgoals (top-down approach) or generalize similar Softgoals into a high level Softgoal (bottom-up approach). We may employ both techniques (middle-out approach) until all Softgoals are fully integrated and a complete SIG is developed. Figure 6 shows the result of the integration process. The NFRs identified with (A), (B), and (C) are NFRs that are carried over from the Use Case driven FR and NFR association diagram. Softgoal (D) and (E) are decomposed from Softgoal (A). Softgoal (A) and (F) can be generalized as Softgoal (G).

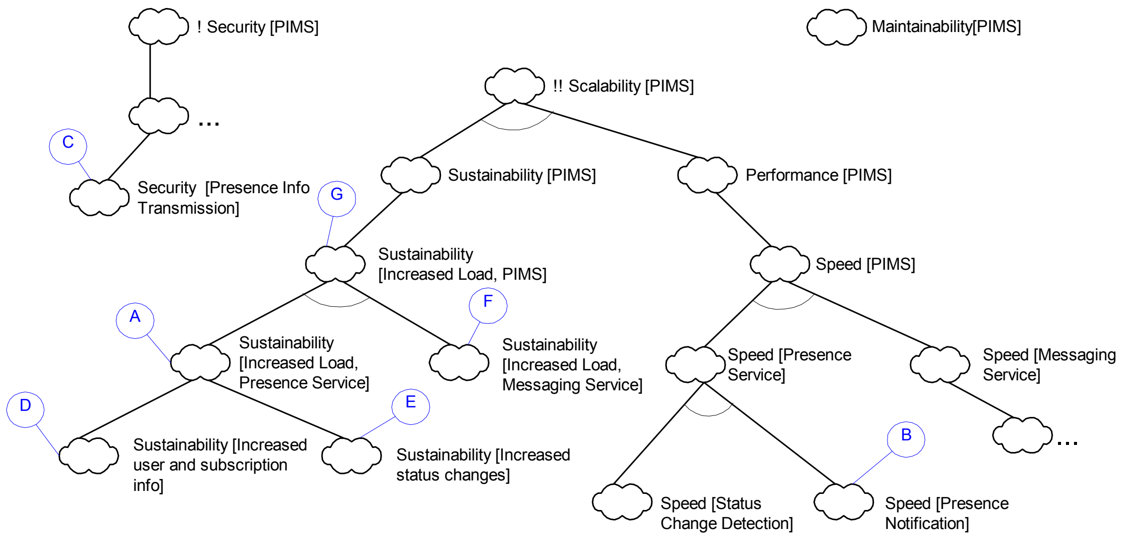
! Security [PIMS]
...
C
Security [Presence Info Transmission]
Maintainability[PIMS]
!! Scalability [PIMS]
Sustainability [PIMS]
Performance [PIMS]
G
Sustainability [Increased Load, PIMS]
Speed [PIMS]
A
Sustainability [Increased Load, Presence Service]
F
Sustainability [Increased Load, Messaging Service]
Speed [Presence Service]
Speed [Messaging Service]
D
Sustainability [Increased user and subscription info]
E
Sustainability [Increased status changes]
B
...
Speed [Status Change Detection]
Speed [Presence Notification]

Figure 6. Refined SIG for NFRs from FR and NFR Integration

4. Operationalizing FRs and NFRs Using Architectures

We now consider architectural alternatives at a macroscopic level to meet both functional and non-functional requirements stated and modeled in Use Cases and SIG. We first identify tasks needed to implement each Use Case. The tasks are used as the goals for operationalizing the FRs. For NFRs, we operationalize the Softgoals. Figure 7 shows the result of the operationalization. The use case is decomposed to tasks and problem domain objects [10] involved with the tasks. Dark clouds represent architectural decisions/alternatives. We mark those operationalizations that are chosen or given higher priority with an exclamation (!). For Change Presence Use Case, we identify the major tasks as Determine Target Subscribers and Notify Subscribers tasks. We enhance the concept proposed in [8] further by associating the tasks with any problem domain objects effected by or involved with the tasks. We then identify architectural decisions and alternatives for the tasks and SIG Softgoals. For architectural/design decisions, we associate the decision with AND relationship to indicate that all is selected using the method described in [14]. For architectural alternatives, we associate them with OR relationship to indicate one of them is required. Each of the alternatives is assigned with a +/- to denote the degree of positive or negative contribution to the task or Softgoal. There is no need for a contribution denotation for architectural decisions as they are all required regardless of the contribution.

5. Operationalizing FRs and NFRs Using Design Patterns

In this step, we consider design patterns at a more microscopic level. We go through design pattern catalog and select design patterns to satisfice the architectural decisions and alternatives. The defined tasks and problem domain objects are used to help guide our design patterns selection. We look for design patterns in the Behavioral Patterns section [9] to satisfice the identified tasks and other architectural decisions/alternatives. We consider design patterns from the Structural Patterns section [9] for tasks that involve problem domain objects.

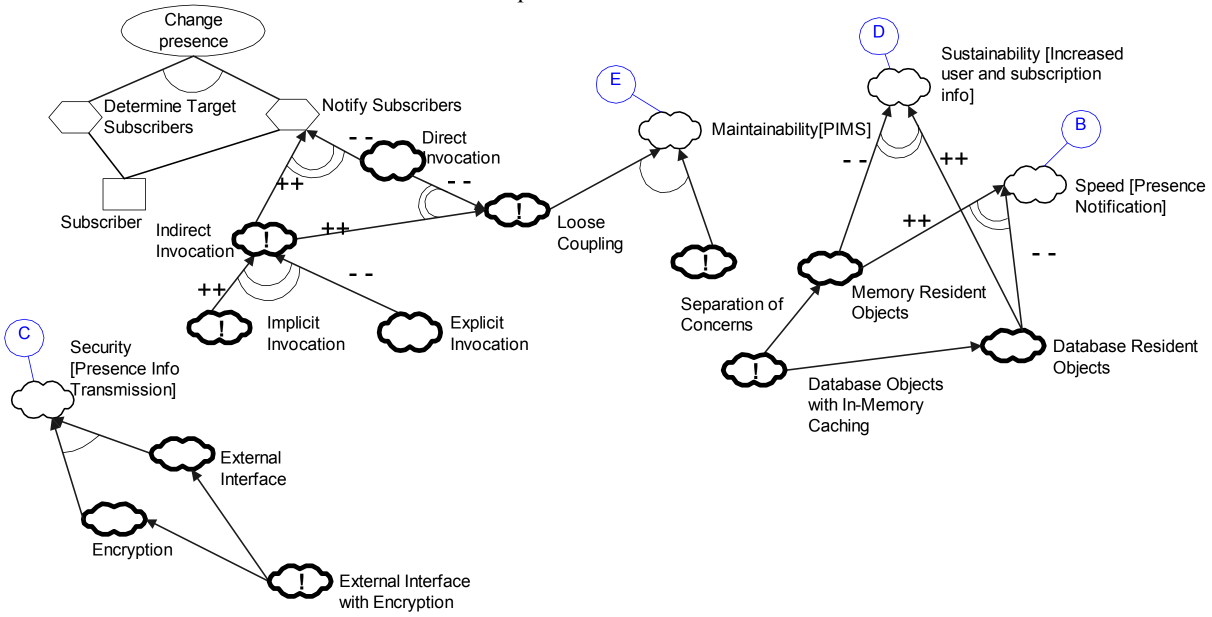

Figure 7. Architectural Operationalizations for Change Presence Use Case.

Figure 8 shows the end result of this process. The dark shaded clouds represent the selected design patterns. Here we say Proxy Pattern with Smart Reference pattern [9] and Factory Pattern [9] satisfice the Database Objects with In-Memory Caching architectural decision. The Proxy is used to provide access to database objects and depends on the Factory Pattern to provide the caching. The Factory Pattern in turn depends on Singleton Pattern [9] to provide convenient access to the Factory object. The Observer Pattern is selected to satisfice Implicit Invocation Softgoal and used as entity objects in the RUP Analysis Model pattern. RUP Analysis Model Pattern [3][4] is selected to provide general partitioning of the design objects.

The design pattern inter-dependency can be used as foundation for "pattern set" solution, as in chip set solution used in the semiconductor industry. It provides higher level solution than the individual design pattern level. It is based on the fact that several design patterns are often used together to realize an architectural decision. With pattern set solution, patterns are preassembled to satisfice certain architectural decisions, and selected using the operationalization process. Using the Change Presence as an example, if we present Implicit Invocation as selection criteria, the pattern selection process (which could be automated) may return a pattern set that is composed of Observer Pattern and Factory Pattern. But if we also add Database Objects with In-memory Caching, the selection process may return a different pattern set that is composed of Observer Pattern, Proxy Pattern with Smart Reference, and Factory Pattern. This high level abstraction is possible because the design patterns are pre-defined with dependency to satisfice certain architectural decisions.

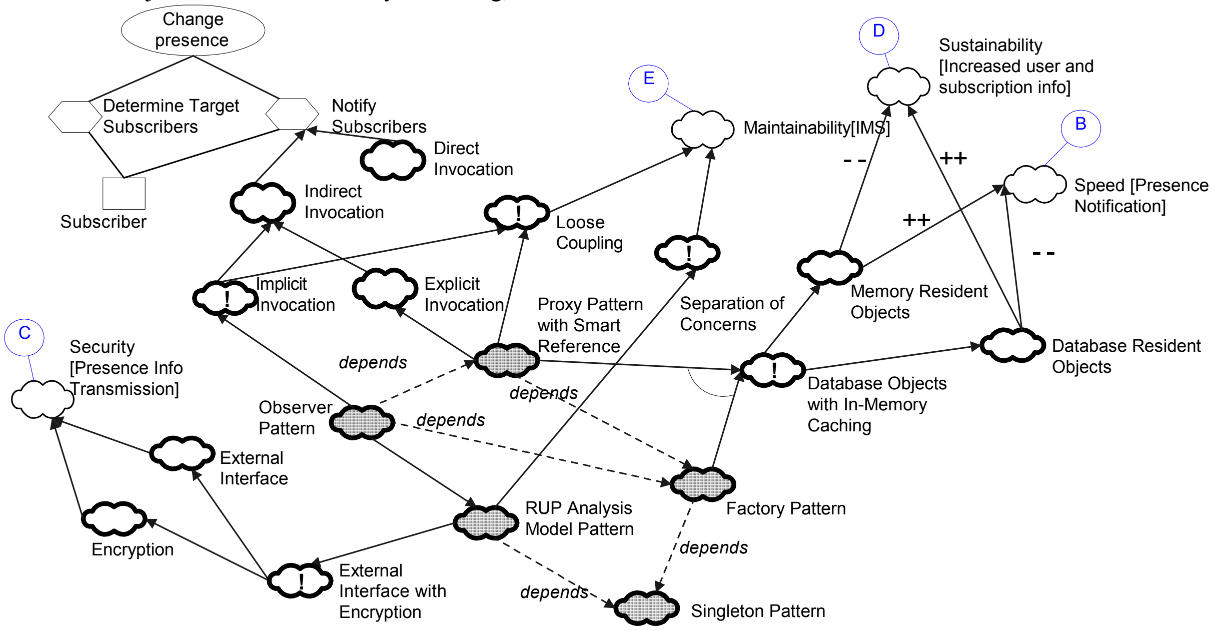

Figure 8. Operationalization with Design Patterns for Change Presence Use Case

The next step is to develop the class diagram to implement the system using the selected design patterns. In general, the objects included in the design serve two purposes. One purpose is to represent the problem domain objects and functionality allocated for those objects, the other is to support the execution of the solution domain. The selected design patterns would involve objects from these two groups as depicted in Figure 9. Pattern 1 consists of objects representing the problem domain entities (shaded area) as well as objects defined to support the run-time execution (non-shaded area). Pattern 2 consists of objects in both areas and also share some objects with Pattern 1. Pattern 3 consists of only objects supporting the execution.

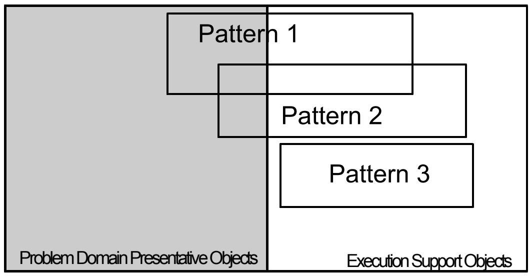
Pattern 1
Pattern 2
Pattern 3
Problem Domain Presentative Objects
Execution Support Objects

Figure 9. Object Participation in the Design Model.

Figure 10 is the class diagram that realizes the SIG in Figure 8. The shaded classes (e.g. Entity class) represent objects representing the problem domain entities. The non-shaded classes are classes defined to support the execution. Here we see the Observer Pattern contains Subject, Observer, PresentityProxy, SubscriberProxy, and Entity classes. PresentityProxy, SubscriberProxy, and Entity objects also participate in Proxy Pattern, Factory Pattern, and RUP Analysis Model Pattern. The EntityFactory class serves as the factory for Entity objects, which also caches recently used Entity objects to improve performance. The

PresenceService serves as the control object and UAInterface serves as the boundary object in the RUP Analysis Model Pattern.

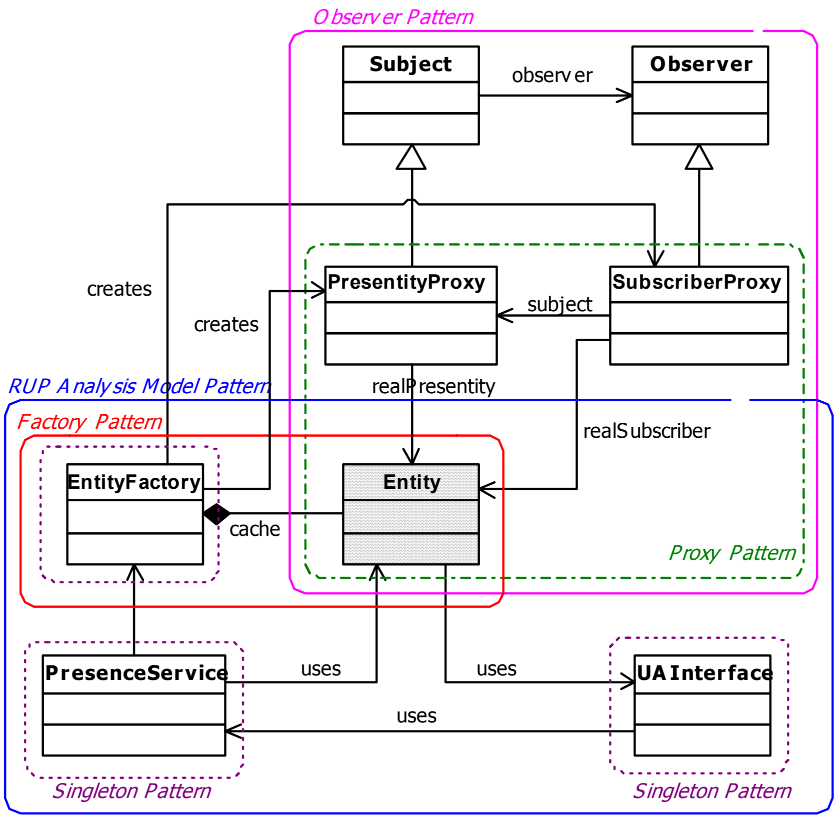

Figure 10. Design Class Diagram for Change Presence Use Case

Figure 11 shows the sequence diagram that describes the interaction among objects to implement the Change Presence Use Case. In some situations, collaboration is more suitable than sequence diagram, for instance when the sequence diagram is complex that involves a lot of objects and messages passed among them. Since collaboration does not use vertical bar to represent points in time, it is free to organize the objects based on structural organization of the objects or other meaningful ways. For objects with states that influence the behavior of the objects based on the state the objects, state chart may be used to model the life cycle of the states and their transitions as shown in Figure 3.

Once the structural model has been developed using UML class diagram, a behavioral model is then developed to implement the functionality described in the Use Cases. UML provides several diagrams for behavior modeling, including sequence diagram, collaboration diagram, activity diagram, and state chart. Among them, sequence diagram is the most popular tool for this purpose. In this step, we look at the Use Cases one at a time, and then develop a sequence diagram to realize the Use Case.

6. Conclusions

This paper has demonstrated how to develop “good” software architecture and design using goal-driven, object-oriented and pattern-based approach. Using the step-by-step process described in this paper, software architects are able to capture both functional and non-functional requirements in the same integrated Use Case driven model. The integrated model is then analyzed to get complete picture of functional and non-functional goals that are later used as guiding criteria for design patterns selection. The goals are then operationalized using architectures and design patterns, and realized using UML class diagram and sequence diagram. This paper also proposes the concept of design pattern dependency which can be used as foundation for “pattern set” solution, which is similar to chip set concept used in the semiconductor industry. It provides higher level solution than the individual design pattern level. Future work may include developing a knowledge base of design pattern compositions for pattern set solution. A CASE tool could be developed to automate the pattern set selection process presented in this paper.

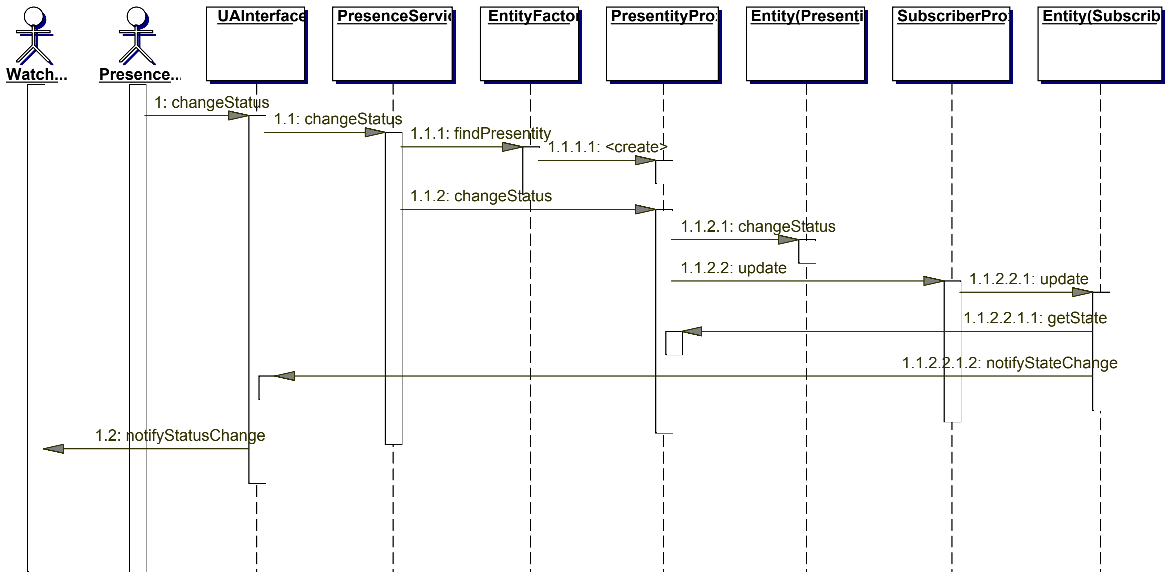

Figure 11. Sequence Diagram for Change Presence Use Case

References

1. M. Day, J. Rosenberg, H. Sugana, A Model for Presence and Instant Messaging, *RFC 2778*, The Internet Society, 2000.

2. M. Day, S. Aggarwal, G. Mohr, J. Vincent, Instant Messaging/Presence Protocol Requirements, *RFC 2779*, The Internet Society, 2000.

3. I. Jacobson, M. Christerson, P. Jonsson, G. Overgaard, *Object-Oriented Software Engineering*, Addison-Wesley, 1992.

4. I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley, 1999.

5. D. Menasce, V. Almeida, Scaling *for E-Business: Technologies, Models, performance, and Capacity Planning*, Prentice-Hall PTR, 2000

6. M. B. Cox, D. Ellsworth, Application-Controleed Demand Paging for Out-of-Core Visualization", *IEEE Visualization*, 1997

7. E. dimitrov, A, Schmistendorf, R. Dumks, UML-based Performance Engineering Possibilities and Techniques, *IEEE Software*, January/February, 2002.

8. D. Gross, E. Yu, From Non-Functional Requirements to Design through Patterns, *Requirement Engineering*, 2001.

9. E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design *Patterns: Elements of Reusable Object-Oriented Software*, Assison-Wesley, 1995.

10. B. Kovitz, *Practical Software Requirements: A Manual of Content and Style*, Manning, 1999.

11. G. Booch, J. Rumbaugh, I. Jacobson, *The Unified Modeling Language User Guide*, Addison-Wesley, 1999.

12. L. Chung, B. A. Nixon, E. Yu and J. Mylopoulos, *Non-Functional Requirements in Software Engineering*, Kluwer Acedemic Publishing, 2000.

13. L. Chung, K. Cooper and A. Yi, "Developing Adaptable Software Architectures for Real-Time Systems Using Design Patterns," To appear in *Proc., Int. Conf. on Software Engineering Practises and Research*, June 24-27, Las Vegas, Nevada.

14. J. Mylopoulos, L. Chung, S. S. Y. Liao, H. Wang and Eric Yu, "Extending Object-Oriented Analysis to Explore Alternatives," *IEEE Software*, January/ February, 2001. pp. 2-6.