Artifacts as Interface: Reasserting the Fundamentals of Software Systems Analysis and Design

Musa J. Jafar
mjafar@wtamu.edu

Jeffry S. Babb, Jr.
jbabb@wtamu.edu

Computer Information & Decision Management,
West Texas A&M University
Canyon, TX 79119

Abstract

In this paper we present an artifacts-based approach to teaching our Object Oriented Analysis and Design course. We focus on (1) the ability to define the system and its boundaries, (2) the separation between business needs and technology requirements artifacts, (3) the clear separation between analysis and design (business-domain models vs. analysis models vs. design models), (4) the evolution of artifacts from domain artifacts, to analysis artifacts and to design artifacts, and (5) the application of abstractions, formal methods and patterns to produce the necessary design artifacts. We assert that the qualities of the design artifacts which result from the process of systems analysis and design convey the essentials of understanding, which elevate the IS discipline as a whole. As we engage the artifacts of our designing, we converse with the problem space in a manner which strengthens our command of the interface between information systems and organizations.

Keywords: Object Oriented, Analysis, Design, use case, object model, sequence diagram, artifacts

1. INTRODUCTION

Software systems analysis and design is a core concern for Information Systems programs. While the content of the course and the depth of content have always been debatable, the course has always been influenced by: (1) the structure of the academic program; (2) the skill set of the faculty teaching the course; (3) the experience of the faculty in software development; (4) the set of tools used in the course; (5) the paradigm used to teach the course (Object-Oriented, structured, etc.); and (6) the prerequisite material for the course (Russell, Tastle, & Pollacia, 2003).

Generally, our concern with software systems analysis and design is in developing (1) an in-depth understanding of the application domain; and (2) communicating descriptions among team members in the same discipline or across disciplines (Analysis, Design, Construction, Testing and deployment). These elements have been well-articulated as "To program is to understand: The development of an information system is not just a matter of writing a program that does the job. It is of the utmost importance that development of this program has revealed an in-depth understanding of the application domain; otherwise, the

information system will probably not fit into the organization. During the development of such systems, it is important that descriptions of the application domain are communicated between system specialists and the organization" (Madsen, Møller-Pedersen, & Nygaard, 1993, p.3).

In addition to systems analysis and design topics, faculty also tend to cover as many software engineering topics as possible in a course on Software systems analysis and design. Some of these topics are hard for students to comprehend. Furthermore, some related subjects, such as user interface design and database design, often require separate courses. Similarly, preparing the deployment environment, designing for scalability, designing for quality assurance, and configuration management are hard to teach in a classroom and typically require many years of experience and on-the-job training. Accordingly, educators need to be very selective of the content they teach and the prerequisites needed as they need to concentrate on the core topics of analysis and design.

To teach students how to analyze, design, build and maintain useful and usable software system products (Brooks, 1995), many IS programs offer a system analysis and design course that focuses on requirements gathering, analysis, and high-level design as an essential element of the undergraduate curriculum. Also, if complemented by a capstone "finishing" and synthesizing course, a course in software systems analysis and design can also focus on low-level design, construction, testing, deployment, and packaging. These two courses cover the major aspects of the factory-life phases of a software system product in contrast to its lifetime in use. Throughout this curriculum thread, students learn about the tools, processes, artifacts and the quality assurance aspects of what is needed to build a software system product (Brooks, 1995; Gupta and Wachter, 1998).

From a system analysis perspective, we attempt to answer the following questions: (1) where do we start? (2) What are the activities that we perform? (3) What are the artifacts that we need to produce? (4) What are the dependencies between the different artifacts?

In our program, our first course in systems analysis and design is a junior/senior level course. For a textbook, we have used "Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design" by (Larman, 2005), w supplemented by other course materials and Microsoft Word document templates from IBM Rational. For software tools we use IBM Rational Architect. As reference texts, we use "Requirements Management Using IBM Rational Requisite Pro" (Zielczynski, 2008), "Visual Modeling with IBM Rational Software Architect" (Quatrani & Palistrant, 2006) and "UML and IBM Rational Unified Process Reference and Certification Guide" (Shuja & Krebs, 2008). All IBM Rational software and educational materials are available free of charge for participating academic programs.

The course has object-oriented programming and database design as pre-requisites. For the homework assignments, students are required to produce the necessary analysis and design artifacts using a combination of Microsoft word documents (using IBM Rational document templates) and UML models using IBM Rational Software Architect. For the final project, students work in teams to produce the complete analysis and design artifacts (Word documents, UML models, and prototype demos).

The authors have been teaching object-oriented analysis and design for more than 10 years. For the seven years before that the authors used object-oriented paradigms to support the analysis, design and development of software products in leading software companies in the U.S. In summary, we watched object-oriented methodologies and their symbolic notations evolve from the early days - the Objectory (Jacobson, Christerson, & Overgaard, 1992), Object Modeling Techniques (OMT), and design patterns (Gamma, Helm, Johnson, & Vlissides, 1995) paradigms - to the current state of UML (Object Management Group, 2010) and Agile development techniques (Steinberg & Palmer, 2004).

In our institution we offer two courses, the focus of this paper is on the first course. In general, we take an artifacts-based approach to teaching the course. Regardless of the software engineering methodology and process model (Agile, Unified, SCRUM, Extreme), we focus on the artifacts that lead to the construction of the product.

In this paper, we share this artifacts-based approach in order to emphasize the role that such a focus plays in strengthening the perceptive skills students require in order to understand the wider process of systems develop-

ment. A focus on the qualities and mechanics of the design and modeling artifacts serves to remind students on the role these artifacts play as an interface between the "… the 'inner' environment, the substance and organization of the artifact itself, and an 'outer' environment, the surroundings in which it operates (Simon, 1996)."

The UML Tool (IBM Rational):

We use IBM Rational Software Architect as a UML-based CASE tool. It is available free-of-charge for academic institutions participating in the IBM academic initiative. The tool provides support for creating, sharing and managing of UML models during analysis and design. It can be used as a repository for the various analysis and design artifacts (model, documents, etc.) (Quatrani & Palistrant, 2006). Figure 1 IBM Rational: User ViewFigure 1 is a screenshot of the tool's frontend.

2. THE ANALYSIS DISCIPLINE

To analyze a system is to build a set of consistent and interrelated models on the basis of which the original system can be designed. During analysis we define: (1) the boundaries of the system modeled as a UML system context model; (2) the users of the system modeled as primary and secondary actors; (3) the functional requirements of the system in association to the primary and secondary actors modeled as a word document; (4) the business logic of the elementary business processes of the system modeled as UML diagrams (use case analysis, system sequence and collaboration diagrams) and a word use case scenarios document; (5) the information models of the system modeled as UML domain object models; (6) the functional architecture of the system modeled as UML functional subcomponents that are dependent on each other and the system (other, non-functional) requirements; and (7) the system, non-functional or other requirements (depending upon what it is named) which is a Word document.

Basically the analysis team produces professional documents using a word processor such as Microsoft Word and rich graphical models using a CASE-tool such as IBM Rational Architect. Accordingly, the analysis team primarily produces artifacts related to documenting an expressive model of the system.

We start by defining the system context artifacts. We use the actors defined in the system context document to define their functional requirements and produce the functional requirements document. We use the functional requirements to detail the use case scenarios and produce the use-cases document, use-case models and system sequence diagrams models. We use the use case scenarios to build the domain object models. We use the domain object model to produce the state transition diagrams. We use the analysis models and system constraints to produce a design models. We use the use cases, system requirements and domain models to produce detailed design.

The System Context

The system context artifact is a UML model that allows us to define the boundaries of the systems. It is an input to the functional requirements of the system. It helps us define (1) business primary actors (both carbon-based and silicon-based) that require services from the system, (2) the primary system actors responsible for administering and maintaining the system due to the imposed system requirements, and (3) the secondary actors (silicon-based) that are in the workflow of the primary actor(s).

When defining primary actors, we have the tendency to ignore the serviceably of the system (system primary actors); we do, however, emphasize that there is always an application administrator actor, a system administrator actor, and in some cases a service layer monitor actor (another system that may have to monitor the health of the application). System primary actors are responsible for the monitoring, operations support, administration, back-up, recovery, maintenance and serviceability of the application. They have their own "System-Level" functional requirements to perform their operations. Using a Student Information System, Figure 2 shows Student(s), Faculty, the Library System, Application Admin and System Admin as primary actors, and the Finance System, the Financial Aid System and the Library System as secondary actors. We are highlighting the Library System as both a primary and a secondary actor to make the point that an actor can be both primary and secondary. Within the UML tool, as illustrated in Figure 1, we can capture the characteristics of each actor group and provide text description within the document editor or attach a document detailing the characteristic of the actor group as a URL.

The Requirements

A requirement is a service the system needs to provide or a capability to which the system needs to conform. Although completely different, requirements are usually divided into (1) the functional requirements that capture the business functions and (2) the systems requirements that provide the scaffolding and the infrastructure support of the business functions of the system. Depending on the software engineering methodology used, the system requirements are also called the nonfunctional, other or supplementary requirements. UML allows for the modeling of functional requirements through use case diagrams, system sequence diagrams, and activity diagrams. UML however, does not provide a framework for modeling system requirements. In summary, the requirements document is a well written word document that includes both the functional and system requirements of the proposed system. It clearly captures the functional and the none-functional requirements of the system. Figure 3 illustrates a sample table-of-content for a requirements document that students use a template.

The Functional Requirements

The functional requirements are the business capabilities the system should provide. They are written in a request for proposal (RFP) format by, or at least with the assistance of, subject matter experts. These requirements are written in clear and unambiguous short paragraphs (in terms such as "should" and "should-not"), with one- or two-paragraph descriptions to provide a high-level understanding of the capability or the restriction.

For each primary actor, we create categorized lists of business functions that reflect the business needs of the actor group. The following is a sample of functional requirements listings:

1.Student Requirements

1 A student should be able to add a course section to their Schedule. During the registration period, using the internet, a student should be able to add a course section to their schedule from the list of open sections as long as it does not exceed the maximum allowed limit for that student.

2 A student should be able to delete a course section from their schedule. During the drop period, using the internet, a student should be able to drop a course section from their schedule as long as they maintain the minimum residency limits.

2.Catalogue dept. Requirements

1 Catalogue dept. should be able to change prerequisites of an existing course. ……

2 Catalogue dept. should be able to assign a course to a degree plan…..

In summary, the functional requirements provide a list of capabilities and restrictions. It is an input to the use case documents.

The System Requirements

The system requirements are capabilities the system needs to conform to. According to Zielczynski (2008), they are all the requirements that cannot be expressed in use cases. System requirements define the scaffolding, the application, and the deployment architecture of the system. They drive the design and specify the system properties. They are categorized into aspects covering security, performance, reliability, usability, testing, technology, external interfaces, operations support, legal, etc. Although two software systems may have totally different functional requirements (Billing vs. HR), it is often the case that they have very similar system requirements. System requirements are usually based on common corporate and industry practices and standards (IEEE Computer Society, 1998). Furthermore, systems requirements are often written by various stakeholder groups according to their interest in the system. For example, security requirements may be written by security engineers to comply with corporate methods and procedures. Maintenance, operations support, and serviceability requirements are written by system administrators, database administrators, and network engineers. Usability requirements are written by the user-centered design (human factors) groups to comply with the look and feel standards of the organization. In general, the systems requirements are conveyed as a simple textual document which highlights the various capabilities that the system needs to comply with.

The system requirements document is an input to the use case details documents used to specify the system needs (usability, data, and performance) for each use case, system architecture document, deployment architecture, and test cases.

The following is a sample of system requirements listings:

1. The System should respond to a user request for a service within 3-5 seconds 90% of the time and no longer than 10 seconds at any time.
2. A user account should become inactive if it has not been used for 90 consecutive days
3. A user should not be able to have more than one concurrent active session
4. The date, time and the IP address of the machine from which a user logged in should be stored into the system.
5. No Open Source code should be used as part of the System
6. All System Windows should have a title that reflects the task at hand, should display the user name and should display the current local date and time.
7. All System windows should have context help.
8. All necessary data should be carried over across multiple active screens
9. Stale records that are more than one year old should be purged out of the system.
10. The System should support single sign-on products
11. Security should be X507 Compliant
12. Client and Server Ports should be configurable

The Use Case Model

Use case modeling is comprised of use case UML diagrams and use case details that are textual documents. Use case diagrams are representations of each actor, their underlying use cases, and the dependencies between use cases (extends and includes). The business logic of functional requirements is detailed in uses-case details document(s). Each functional requirement is traced to one or more concrete use case and each concrete use case is traced back to one or more functional requirement. A use case document details an elementary business process.

<Use Case Name>

1. Brief Description

2. Satisfied Requirements List
   1.<Requirement Name a& Number>
   2.<Another Requirement Name & Number>

3. Actors List
   1.<Actor Group Name>
   2.<Another Actor Group Name>

4. Preconditions
   1.<Precondition>
   2.<Another precondition>

5. Use Case Flow
   1.Basic Main Flow
   2.Alternative Flows
   3. Optional Flows
   4.Exception Flows

6. Post Conditions
   1.<Post Condition>
   2.<Another Post Condition>

7. Included Use Cases
   1.<Use Case Name and Number>
   2.<Another Use Case Name and Number>

8. Extending Use Cases
   1.<Use Case Name and Number>
   2.<Another Use Case Name and Number>

9. Special Requirements
   1.<Special Requirement>
   2.<Another Special Requirement>

10. Assumptions, Open Issues and Comments

It is a coherent set of functions that embodies the business logic needed for the system to provide while moving the system from one consistent state to another consistent state in response to an actor's request for service. During analysis, abstract use cases are extracted from the concrete use cases. Abstract use cases contain reusable business logic components that are common to more than one use case. When a use case is too big, it is also abstracted into a set of simpler use cases to simplify the business logic through abstraction. For example, "check-prerequisites," "get-

probation-status," and "validate-registration-card" are abstract use cases of the "register-for-class" concrete use case, **Error! Reference source not found.**.

Many sources provide templates for use case documents. We use the template from IBM Rational as a skeleton and we modify it as needed (Zielczynski, 2008). See previous chart for an example.

The Domain Object Model

The domain object model is the set of domain objects, the attributes of each object with their constraints and data types, and the set of associations between objects. Associations have cardinality attributes and are regular, aggregation, containment, inheritance or taxonomic. The domain object model is a UML artifact that is comprised of a set of diagrams and the underlying descriptions and semantic content of the object model artifacts. In summary, it is a visual representation of the domain objects of the system, their attributes, constraints and associations with other classes. Each use case scenario exposes certain objects, object attributes and relationships. For example, from a login use case, we learn that a user (student, faculty, staff, etc.) has a user id and a password. From the add class use case scenario, we learn that students have study plans and majors, and courses have prerequisites. By analyzing the use cases, the object model can be built from the ground up. Figure 5 is an example of a mini object model.

The State Diagrams

For each domain object, a state diagram captures the noteworthy, finite, and discrete states of an object. Not every object necessarily has noteworthy states. State transitions of the same object are usually confused with the inheritance hierarchy of an object. For example, a student status as freshman, sophomore, junior, or senior represents the possible state transitions of the undergraduate student object rather than as subclasses of student. Figure 6 is an example of a state transition diagram of the object student

3. THE DESIGN DISCIPLINE

To design a system is to develop a set of design and modeling artifacts – and subsequently an overall system model – from which the original system can be built. Given the set of all the requirements artifacts, time constraints, technology constraints, and financial constraints, the system design is a feasible solution that satisfies these constraints. During design, inputs, processing, data storage, output and communication software artifacts are materialized into a set of layered architectures that are comprised of user-layers, processing layers, data layers, communication layers, security layers, etc. In this sense, designing is about making commitments on the distribution of business logic and the processing of business across the layers. During design, objects are refined and redefined, new objects are also introduced. For each business object, the designers have to define the GUI components or boundary classes (Screen designs and layouts), processing components or processing classes (class responsibilities and collaborations) and data components or entity classes (tables and foreign keys if the underlying data layer is a relational database system). During design, control classes are also defined. During design, students learn how to realize the analysis model (set of analysis artifacts) into a design model with a set of artifacts (Design object models, sequence diagrams, database logical schemas, etc.). Students also learn to create other design objects artifacts such as control classes, listeners, messaging classes, etc. In summary, students learn the various design patterns such as model-view controllers, model/view separation, class responsibilities, and collaborations etc.

During design, use cases are realized into detailed sequence diagrams where commitments are made as to the distribution of processing. For example, given a login use case, should the processing to validate a user be performed in the user layer, the processing layer, or in the data layer through stored procedures? Each one of these designs has its own advantages and disadvantages. During design, a commitment as to how to implement the business logic is clearly outlined.

Using a student login account to the system use case scenario, students learn to identify the design objects of the use case Figure 7. A design commitment needs to be made as to who is responsible for validating the credentials. Students learn to produce detailed sequence diagrams to realize the design of use cases.

In the Figure 8, the "Login-Screen" object basically controls all the authentication operations and the creation of other objects.

However, another sequence diagram could have distributed the logic among the various objects. Accordingly, design is a commitment to a processing logic scenario that is low coupled and highly cohesive.

4. SUMMARY AND CONCLUSIONS

In this paper we presented an outcomes-focused, artifacts-based, hands-on, and disciplined approach to analysis and design. Our objective is to present a disciplined approach to understanding and producing the necessary analysis and design artifacts (documents and models) which consistently leads to a successful system regardless and irrespective of the systems development paradigm, model, and methodology used to build the system. Our premise is that the adherents of a predictive model, such as the Capability Maturity Model, or the adherents of an adaptive model, such as Extreme Programming, should both be equally comfortable with commonly-accepted artifacts of object-oriented systems design modeling. We acknowledge that bridging the gap between process modeling and object-oriented systems modeling remains a challenge; we contend that students will bridge this gap with experience. However, without a solid grounding in the qualities and characteristics of design artifacts themselves, the "craft" of systems design will be elusive. We think of the artisan who must learn the tools of their craft before they worry about the holistic and philosophical concerns of their craft. In this sense, we feel that we are preparing our students to use their knowledge of the characteristics and qualities of design artifacts to then develop their experience.

We foresee that our students will approach their initial years in the profession as an opportunity to learn how their designing of artifacts helps them to understand the systems they build and the context of the organizational problems these systems address. More importantly, by knowing their tools, our students can then focus on what is, and is not, possible as they navigate the complexity of systems specification. As they mature in their profession, our students must develop a sense of how the juxtaposition of the materials of design (the artifacts), the constraints of the design process, and the organizational constraints of the system's intended operational environment, transform their understanding of the analysis and design process. This is so also in a cumulative and iterative tradition: experience is accrued as the design process is continually engaged. We err on the side of the artifacts-based approach as we believe our students are better equipped to learn about the art and craft of systems designing if they are first aware of the indelible truth inherent in the characteristics and qualities of the artifacts of design.

Schön and Bennett (1996) put it well when they described a "reflective conversation with materials" that designers conduct as they reflect on practice. In this case, "practice" is the consistent use of design artifacts, which is only possible when design artifacts (the materials of designing) are well-understood. We see this in other areas which invite mastery: those learning the piano practice and exercise in the structures of chords, notes and scales; those learning to dance exercise in the mechanics of movement; and those learning a team sport exercise in the patterns of play. Accordingly, we have chosen to focus on the artifacts of design in our curriculum. Once armed with the "scales" and building-blocks of design artifacts, we believe that our students can design within the framework of a development model in the same manner that a musician trained in the virtues of sight-reading can work within the context of many styles of music.

Most fundamental to our approach is that the characteristics and qualities of the artifacts of design provide the best interface between the system and those that will use the system. In the artifacts we have a "lingua franca" which allows the realm of Information Technology to understand and accommodate the realm of the organization. This interfacing is at the heart of the Information Systems discipline and is most representative of the skills and knowledge most suited to our students' development.

4. REFERENCES

Alter, S. (2006). Pitfalls in Analyzing Systems in Organizations. *Journal of Information Systems Education*, 17(3), 295-302.

Brooks, F. P. (1995). The Mythical Man-Month. Addison-Wesley.

Cockburn, A. (2001). Writing Effective Use Cases. Addison-Wesley.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1995). Design Patterns, Elements of Reusable Object-Oriented Software. Addison Wesley.

Gupta, J. and Wachter, R. (1998). A Capstone Course in the Information Systems Curriculum. *International Journal of Information Management*, 18(6), 427-441.

IEEE Computer Society. (1998). IEEE Recommended Practice for Software Requirements Specifications. IEEE Computer Society. New York, NY: IEEE Computer Society.

Jacobson, I., Christerson, M., & Overgaard, G. (1992). Object-Oriented Software Engineering, A Use Case Driven Approach. Addison Wesley.

Larman, C. (2005). Applying UML and Patterns. Prentice Hall.

Madsen, O. L., Moller-Pedersen, B., & Nygaard, K. (1993). Object-Oriented Programming in the BETA Programming Language. Retrieved 6 12, 2010, from www.daimi.au.dk/~beta/Books/betabook.pdf:

Object Management Group. (2010, May 03). OMG Unified Modeling Language (OMG UML) Infrastructure, Version 2.3. Retrieved June 05, 2010, from http://www.omg.org/spec/UML/2.3/

Quatrani, T., & Palistrant, J. (2006). Visual Modeling with IBm Rational Software Architect. Upper Saddle River, New Jersey: IBM Press, Pearson Ed.

Rumbaugh, J., Jacobson, I., & Booch, G. (2005). The Unified Modeling Language Reference Manual. Addison Wesley.

Russell, J., Tastle, W., & Pollacia, L. (2003). The State of Systems Analysis and Design. ISECON. San Diego.

Schon, D.A. & Bennett, J. (1996). Reflective Conversation with Materials. Bringing Design to Software, Winograd, T. (Ed.). New York: ACM Press.

Shuja, A. K., & Krebs, J. (2008). IBM Rational Unified Process Reference and Certification Guide. IBM Press, Pearson Ed.

Simon, H.A. (1996). The Sciences of the Artificial, 3rd Edition. Cambridge: MIT Press.

Steinberg, D. H., & Palmer, D. W. (2004). Extreme Software Development. Pearson Prentice Hall.

Zielczynski, P. (2008). Requirements Management Using IBM Rational RequisitePro. Upper Saddle River, New Jersey: IBM Press, Pearson Ed.

APPENDIX

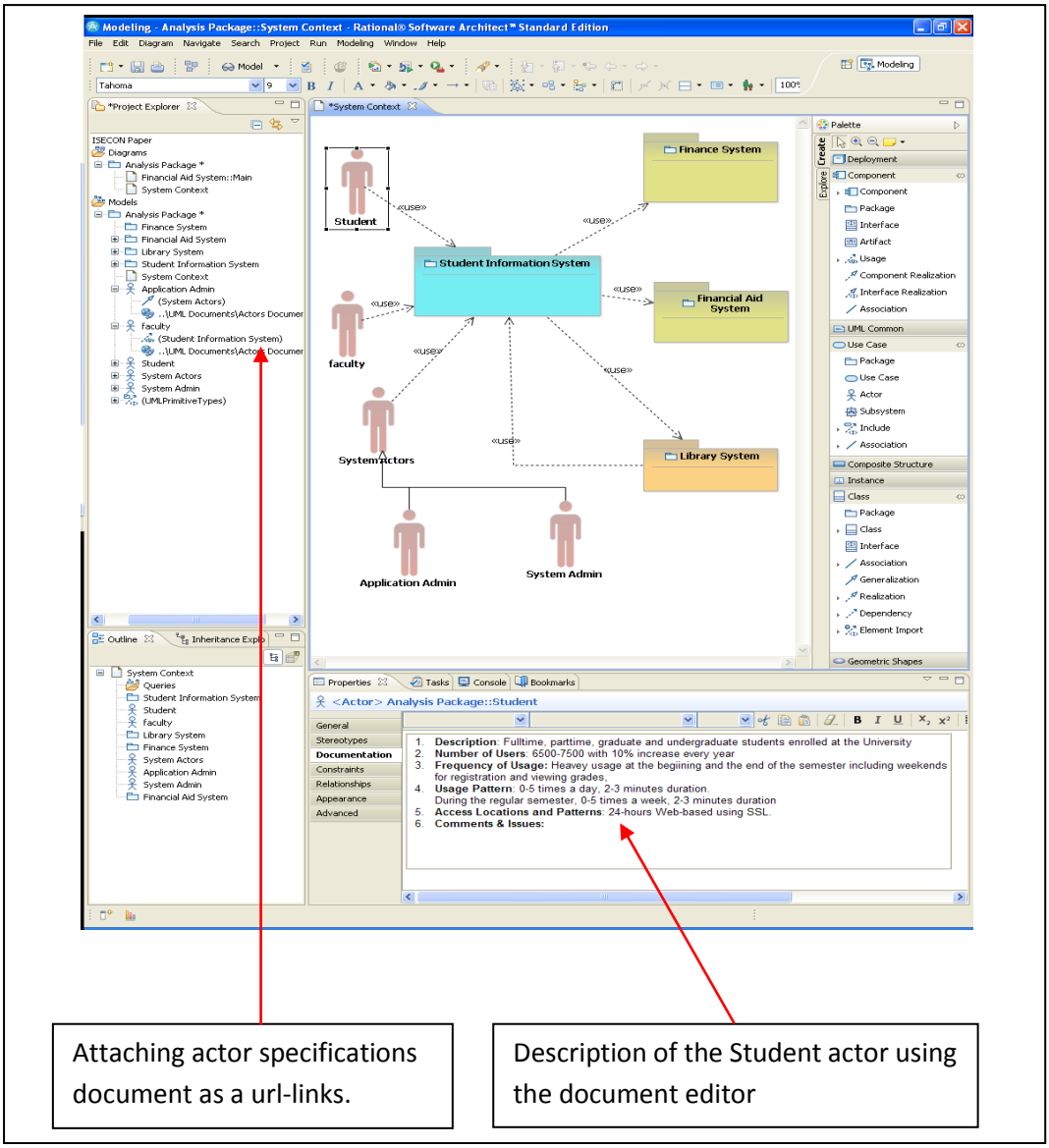

Figure 1 IBM Rational: User View

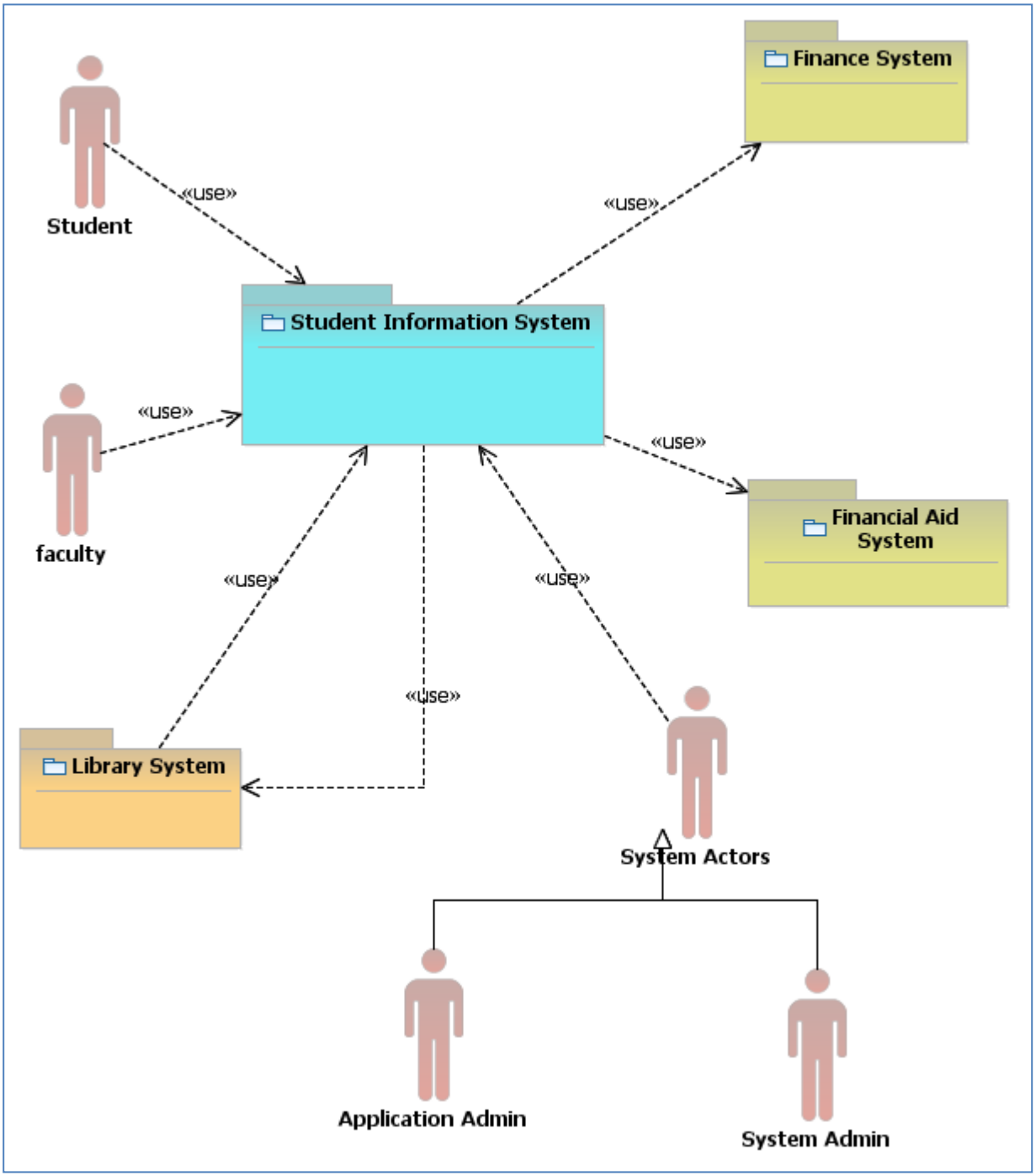

Figure 2 A System Context Diagram

Figure 3 The Table of Contents for A Requirements Document

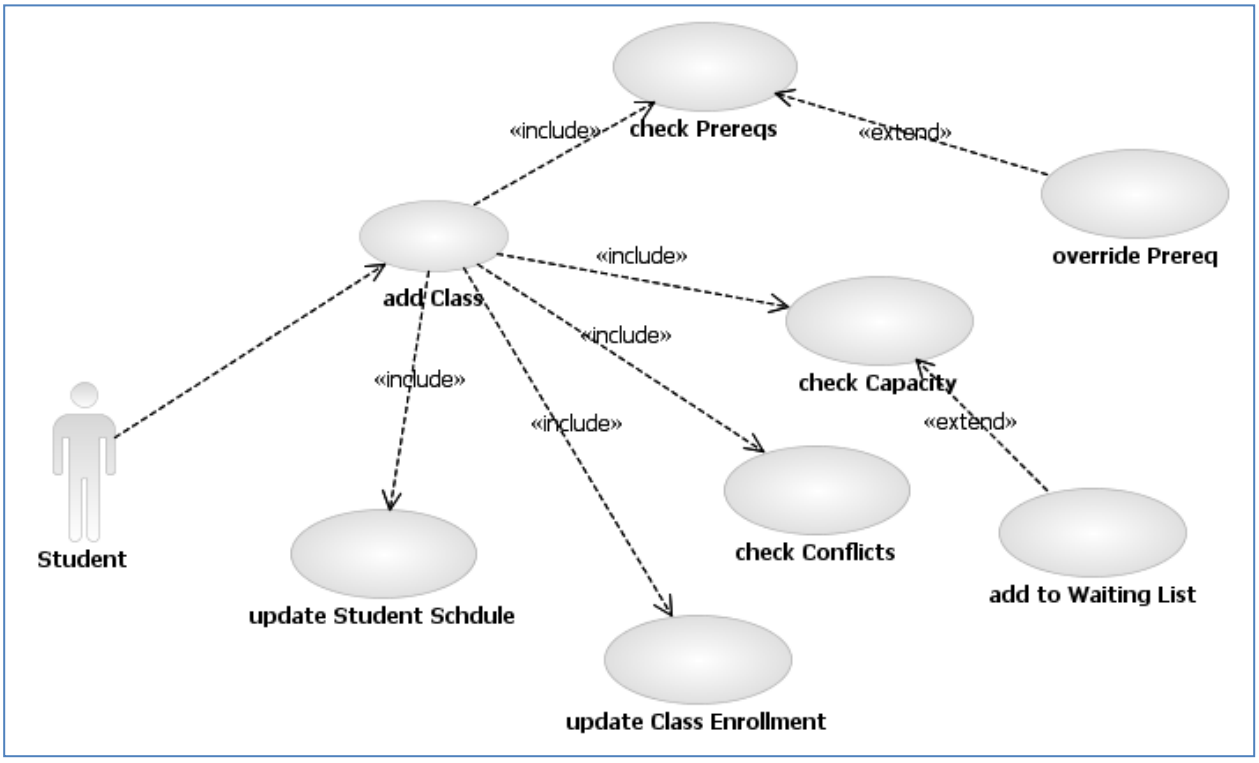

Figure 4 A Use Case Analysis UML Diagram

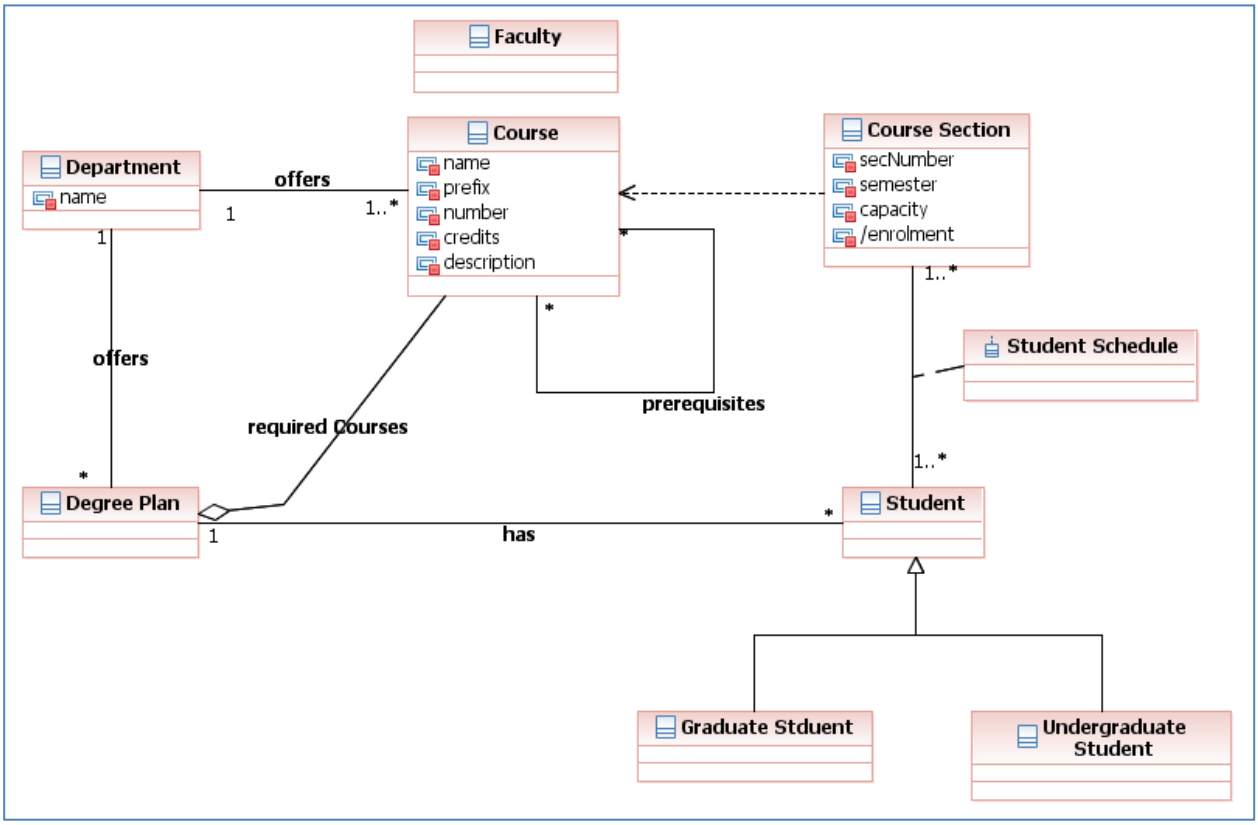

Figure 5 A Simple Domain Object Model

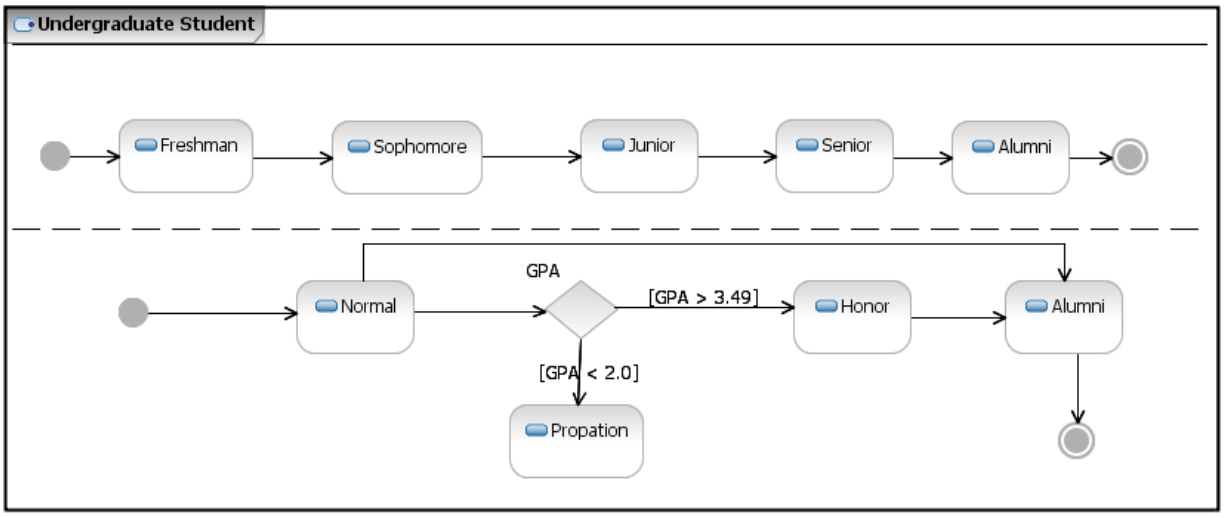

Figure 6 State Transitions of an Undergraduate Student

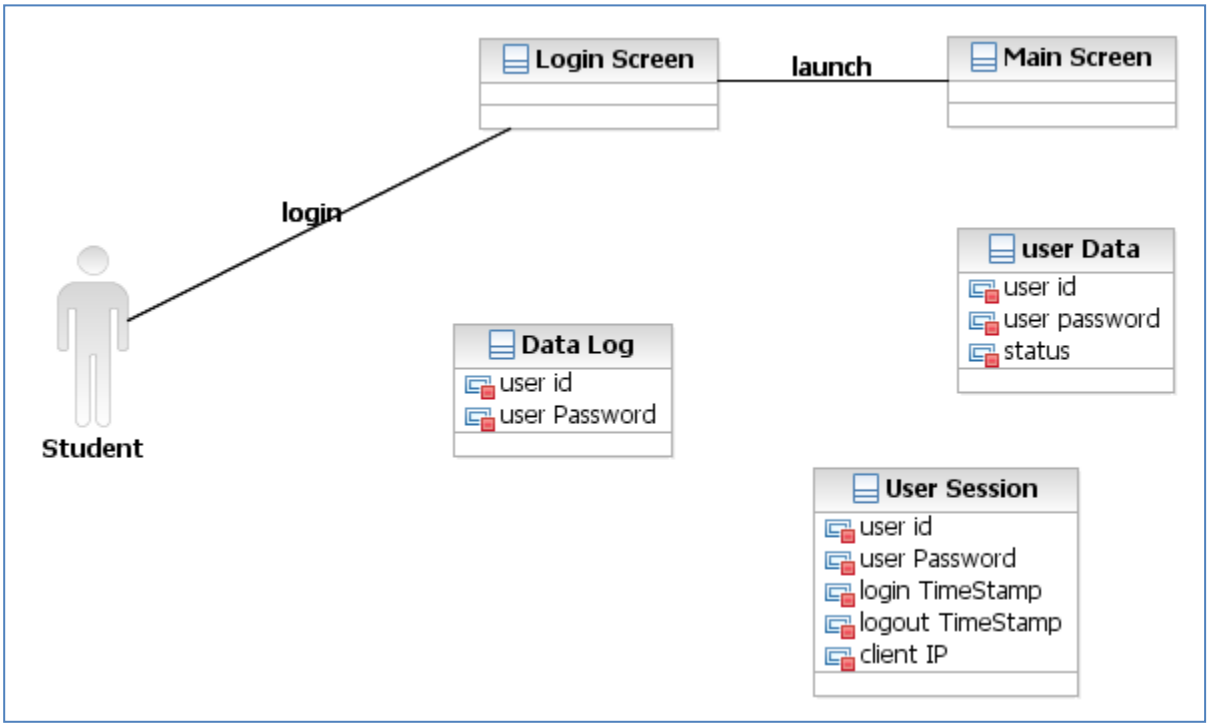

Figure 7 A design Object Model of Login Use Case

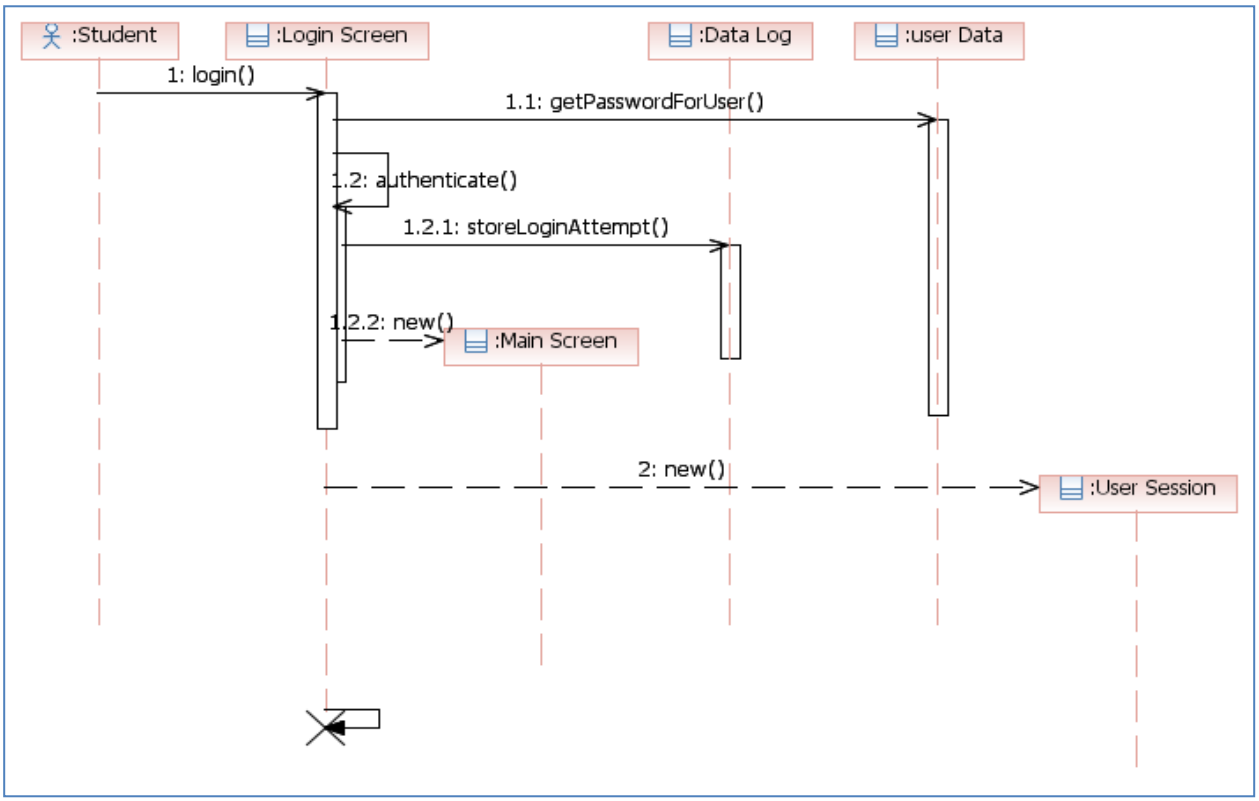

Figure 8 Login Sequence Diagram One