Chapter - Three
Gathering user requirements and Reusable technologies

Introduction

An important part of software development is to explore the requirements for your system. Usage modeling explores how people work with a system, vital information that you require if you are going to successfully build something that meets their actual needs. You cannot successfully build a system if you do not know what it should do, and a critical aspect of this is exploring how people will actually use the system.

What is Requirement?

Requirement in any organization is user needs which they get from service of any kind of software product.

- ➢ Requirement can be divided into two. These are:

1. Fictional Requirement
2. Non-Functional requirement

Functional Requirement

- Something which is intended to be used or practical
- Specify what a system do
- What the system does, or.
- Captures behavioral aspects
- Backbone of the software requirement.
- It depends on the complexity of the software system.
- Define functions and functionality.

Examples of FR: **Library Management system**

What are functional requirements?

- ➢ Issue a book

- Return a book
- Membership facility
- Reservation of books
- Recording member's data
- Visiting book status

Non-Functional Requirements

- **Implicit expectations from the product**
- **Expected features and specifically documented requirements, known as** "quality attributes" of a system.
- Also "qualities", "quality goals", "quality of service requirements", "constraints" and "non-behavioral requirements".

Examples of NFR: Based on the above Library Management system

What are non-functional requirements?

1. Security: Security levels (users, librarian administrative)
2. Usability: all members and users should effectively and easily use the system (user friendly)
3. Availability: the system gives service 8/24
4. Scalability: the system should be enough to accommodate to the changes
5. Reliability: the system should be enough to handle the those problems
6. Efficiency: response time

3.1 Techniques (Methodology) of Data Gathering

You should strive to be adept at three fundamental information gathering skills:

- **Interviewing**
- **Observation: (Ethnography)** open and Close
- **Questioner**
- **Manual(case study)**
- **Workshop**
- **Group discussion (Brain storming)**: technique where groups of people discuss a topic and say anything that comes into their minds about it.

3.1.1 Interviewing

Although you should have one or more project stakeholders actively involved in your project, and therefore readily accessible to you, you will still find that you need to obtain information from others from time to time. A common way to do this is via interviews. When interviewing someone you have several potential goals to accomplish:

- Broaden your understanding of the business domain;
- Determine whom to invite to become active stakeholders; and
- Identify new or existing requirements directly for the application.

Interviewing is a skill that takes years to master, one that cannot possibly be taught in a few paragraphs. There are, however, a few helpful pointers to help you to improve your interviewing skills:

1. Send ahead an agenda to set expectations and thus allow your interviewee to prepare for the interview. An agenda can be something as simple as a short e-mail.
2. Verify a few hours ahead of time that the interviewee is still available because schedules may change.
3. Thank the interviewee for taking time out of their busy day.
4. Tell the interviewee what the project is about and how input is important.
5. Summarize the issues you want to discuss and verify how long the interview will take. This helps to set expectations and enables the interviewee to help you manage the time. This should mirror the agenda sent in step1.
6. State and verify all assumptions. Do not assume that each stakeholder holds the same assumptions.
7. Ask the interviewee whether you have missed anything or whether anything should be added. This gives him a chance to voice concerns and often opens new avenues of questioning.
8. Practice active listening skills—ask open-ended questions. Do not assume you know everything, especially if you think you have already heard it before. Your stakeholders rarely have a consistent view of the world and part of the requirements definition process is to understand where everyone is coming from. If everyone has the same view on

everything, that is great, but it is incredibly rare. Do not shut down your users with a comment like "I have already heard this before. . . ."

9. End the interview by summarizing the main points. Discuss any misunderstandings and clear them up at this point. This gives you a chance to review your notes and ensure you understood everything.
10. Thank the person at the end of the interview. Follow up with a thank-you note too.
11. If you formalize the interview notes, send them to the interviewee for review. This helps to put your interviewees at ease because they know the input was not taken out of context. It also helps to improve the quality of your notes because they will provide feedback.
12. Interviewing is more about listening than talking. Talking starts the process, but the information you want comes from listening to responses. Even when you ask an important question, e.g., "This system needs to be restricted to field reps, right?", the important part of the interview is the answer, e.g., "Yes" or "No" or "Well it's actually like this. . . ."

3.1.2 Observation

Make a habit to spend a day or two with my direct end users simply to sit and observe what they do. One of the problems with interviewing people is they leave out important details, details that you may not know to ask about because they know their jobs so well. Another advantage of observing your users is you see the tools they use to do their jobs. Perhaps they use a key reference manual or use a notepad to write reminder notes for themselves and/or their co-workers. Often they do things differently than the official manual tells you it should be done. Taking the time to observe users doing their work can give you insight into the requirements for the application you are building.

3.1.3 Brainstorming

Brainstorming is a technique where groups of people discuss a topic and say anything that comes into their minds about it. The rules are simple; all ideas are

- Good—ideas are not judged by the group;

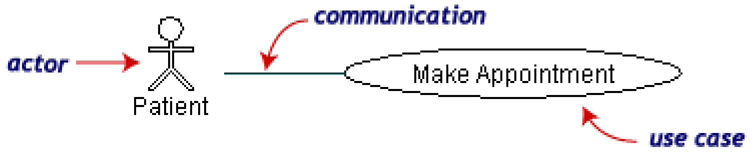
- Owned by the group, not by the individual; and
- Immediately public property: anybody is allowed to expand upon them.

The basic idea is that someone, referred to as a facilitator, leads the group through brainstorming. The facilitator starts by explaining the rules of brainstorming and explaining what issues are to be discussed. All people present must understand and abide by these rules. A good idea is to give everyone a copy of the brainstorming rules before a brainstorming session so they are aware of them. When someone suggests an idea, it should be immediately recorded onto a publicly visible area, such as a flip chart paper or a white- board.

3.2 Essential Use Case Modeling/Diagram

Now let us see what the simple diagram potentially evolve. The use case diagram in Fig. 3.1 provides an example of an UML use case diagram. Use case diagrams depict:

- **Use cases.** A use case describes a ***sequence of actions*** that provide something of measurable value to an actor and is drawn as a **horizontal ellipse**.
- **Actors.** An actor is a ***person, organization, or external system*** that plays a role in one or more interactions with your system. Actors are drawn as **stick figures**.



- **Associations.** Associations between actors and use cases are indicated in use case diagrams by **solid lines**. An association exists whenever an actor is involved with an interaction described by a use case. Associations can also exist between use cases and even between actors, although this is typically an issue for system use case models.
- **System boundary boxes (optional).** You can draw a rectangle around the use cases, called the system boundary box, to indicate the scope of your system. Anything within the box represents functionality that is in scope, and anything outside the box is not.

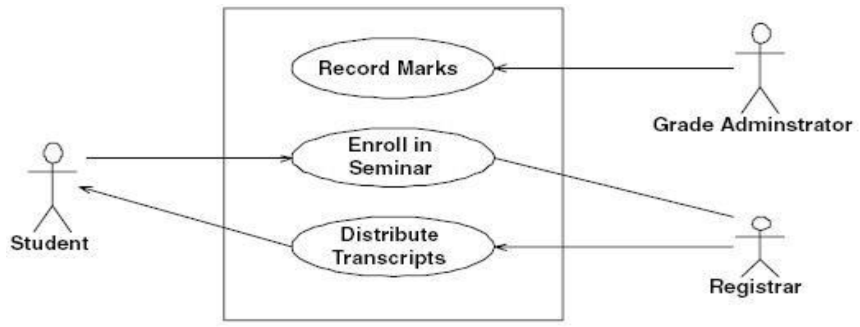


Figure 3.1: A simple use case diagram for a university.

In the example depicted in Fig. 3.2, students are enrolling in courses with the potential help of registrars. Professors input the marks students earn on assignments and registrars authorize the distribution of transcripts (report cards) to students. Note how for some use cases there is more than one actor involved. Moreover, note how some associations have arrowheads—any given use case association will have a zero or one arrowhead. The association between *Student* and *Enroll in Seminar* indicates this use case is initially invoked by a student and not by a registrar (the *Registrar* actor is also involved with this use case). Understanding that associations do not represent flows of information is important; they merely indicate an actor is somehow involved with a use case. Information is flowing back and forth between the actor and the use case; for example, students would need to indicate which seminars they want to enroll in and the system would need to indicate to the students whether they have been enrolled. However, use case diagrams do not model this sort of information. The line between the *Enroll in Seminar* use case and the *Registrar* actor has no arrowhead, indicating it is not clear how the interaction between the system and registrars start. Perhaps a registrar may notice a student needs help and offers assistance, whereas other times, the student may request help from the registrar, important information that would be documented in the description of the use case. Actors are always involved with at least one use case and are always drawn on the outside edges of a use case diagram.

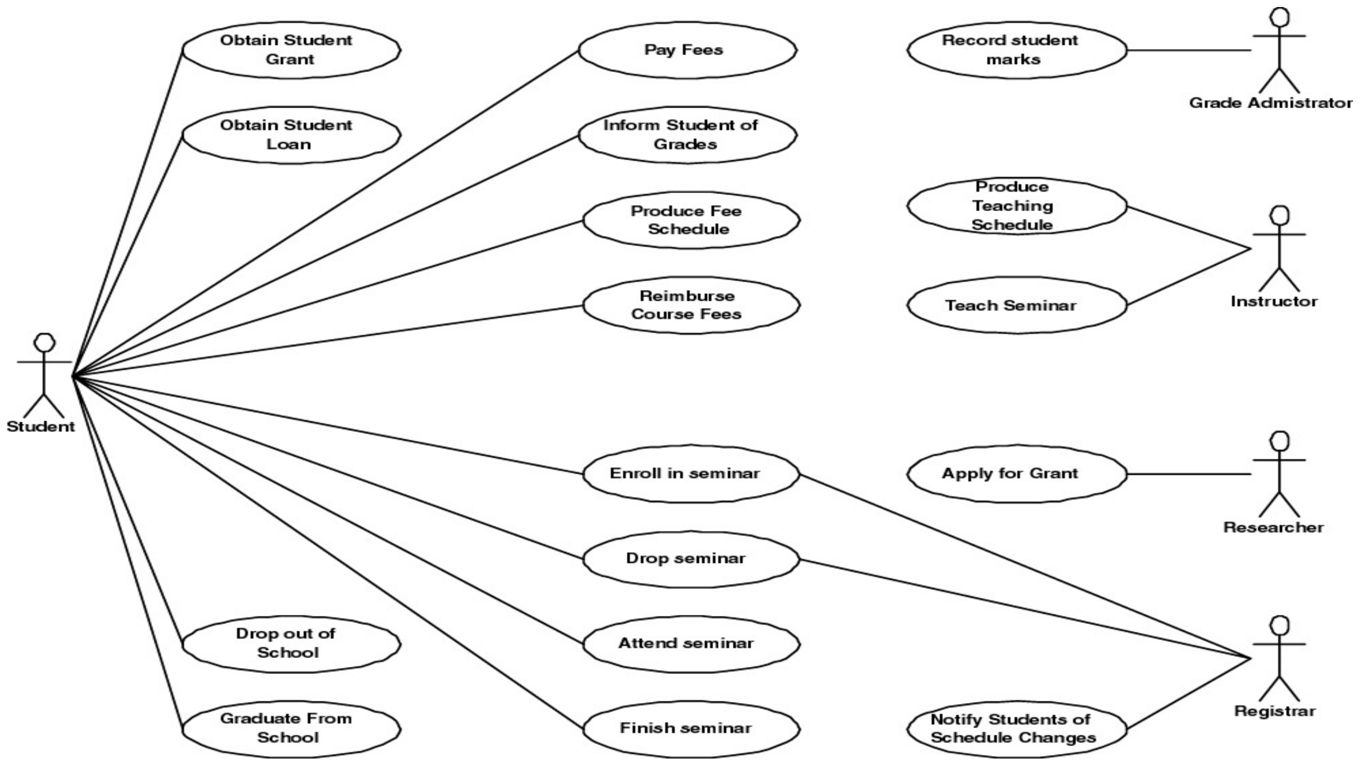


Figure 3.2: A more complex use case diagram for the same university.

Use case diagrams give you a very good overview of the requirements. You often draw a use case diagram while you are identifying use cases, actors, and the associations among them.

You should ask how the actors interact with the system to identify an initial set of use cases. Then, on the diagram, you connect the actors with the use cases with which they are involved. If actor supplies information, initiates the use case, or receives any information as a result of the use case, then there should be an association between them.

The goal of software development is to build working software that meets the needs of your project stakeholders. If you do not know what those needs are then you cannot possibly succeed.

There are several interesting things to note about the use cases of Fig. 3.2:

1. **These are just preliminary use cases.** As you continue to model, you will refractor existing use cases, break them apart, combine them, introduce new use cases, and remove ones that do not make sense. Your models evolve over time: they are never "carved in stone."

2. **No time ordering is indicated between use cases.** For example, you need to enroll in a seminar before you attend it, and you need to attend it before you pass it. Although this is important information, it does not belong in a use case diagram.
3. **Customer actors are usually involved in many use cases.** The customer of the university, in this case *Student*, is the center of much of the action in the use case diagram. This is a normal phenomenon—after all; the main goal of most organizations is to provide services to their customers.
4. **Use cases are not functions.** Begin by listing a series of tasks/business processes that the system needed to support, and then use this information to formulate the use case diagram. There is not one-to-one mapping of processes to use cases, however. For example, the need for an available seminar list is identified as a function, but is modeled simply as a reference in a use case statement. Use cases must provide something of measurable value to an actor. Failing to do so may indicate that the task/process is a step in a use case instead of a use case by itself.
5. **No arrowheads are on the associations.** As we discussed earlier, showing arrowheads on use case associations can be confusing for many people and, frankly, they do not add much value anyway.
6. **Use cases should be functionally cohesive.** A use case should encapsulate one service that provides value to an actor. Accordingly, separate use cases exist for dropping a seminar and enrolling in a seminar. These are two separate services offered to students and, therefore, they should have their own use cases.
7. **Use cases should describe something of business value.** Use cases describe something of value to one or more actors.
8. **Every actor is involved with at least one use case, and every use case is involved with at least one actor.** Remember the definition of a use case: it provides a service of value to an actor. If a use case does not provide service to an actor, then why have it? If an actor not involved in a use case exists, why model it?
9. **Repetitive actions need not be expressed within a single use case.** For example, a student will typically enroll in many seminars each semester. It is not necessary to express this fact within the use case, unless it makes sense to do so, as an actor may invoke the single *Enroll in Seminar* use case multiple times.

3.2.1 Identifying Actors

An actor represents anything or anyone that interfaces with your system. This may include people (not just the end user), external systems, and other organizations. Actors are always external to the system being modeled; they are never part of the system. To help find actors in your system, you should ask yourself the following questions:

- Who is the main customer of your system?
- Who obtains information from this system?
- Who provides information to the system?
- Who installs the system?
- Who operates the system?
- Who shuts down the system?
- What other systems interact with this system?
- Does anything happen automatically at a preset time?
- Who will supply, use, or remove information from the system?
- Where does the system get information?

When you are essential use case modeling, your goal is to use actors to model roles and not the physical, real-world people, organizations, or systems. For example, Fig. 3.1 shows that the *Student* and *Registrar* actors are involved with the use case *Enroll in Seminar*. Yes, it is extremely likely that students are people, but consider *Registrar*. Today, at most modern universities, people are in the role of *Registrar*. But does it really need to be like this? Consider what registrars do: they mediate the paperwork between a university and a student, they validate the information a student submits, and they provide advice to students regarding which seminars to enroll in.

Name: Grade Administrator **Description**: A Grade Administrator will input, update, and finalize in the system the marks that students receive on assignments, tests, and exams. Deans, professors (tenured, assistant, and associate), and teaching assistants are all potential grade administrators

Figure 3.3: Description of the Grade Administrator actor.

To describe an actor you want to give it a name that accurately reflects its role within your model. Actor names are usually singular nouns, such as *Grade Administrator*, *Customer*, and *Payment Processor*. You also want to provide a description of the actor—a few sentences will usually do—and, if necessary, provide real-world examples of the actor. Figure 3.3 provides an example of how you might describe the *Grade Administrator* actor.

3.2.2 Identifying Use Cases

How do you go about identifying potential use cases? One way to identify essential use cases, or simply to identify use cases, is to ***identify potential services*** by asking your stakeholders the following questions from the point of view of the actors:

- What are users in this role trying to accomplish?
- To fulfill this role, what do users need to be able to do?
- What are the main tasks of users in this role?
- What information do users in this role need to examine, create, or change?
- What do users in this role need to be informed of by the system?
- What do users in this role need to inform the system about?

For example, from the point of view of the *Student* actor, you may discover that students

- Enroll in, attend, drop, fail, and pass seminars.
- Need a list of available seminars.
- Need to determine basic information about a seminar, such as its description and its prerequisites.
- Obtain a copy of their transcript, their course schedules, and the fees due.
- Pay fees, pay late charges, receive reimbursements for dropped and cancelled courses, receive grants, and receive student loans.
- Graduate from school or drop out of it.

- Need to be informed of changes in seminars, including room changes, time changes, and even cancellations.
- Provide fundamental information about themselves such as their name, address, and phone number.

Similarly, another way to identify use cases is to ask your stakeholders to brainstorm the various scenarios, often called usage scenarios, which your system may or may not support. A usage scenario is a description of a potential business situation that may be faced by the users of a system—the focus is on behavioral requirements issues, not technical design issues. For example, the following would be considered ***use case scenarios*** for a university information system:

- A student wants to enroll in a seminar, but the registrar informs him that he does not have the prerequisites for it.
- A student wanted to enroll in a seminar that she does have the prerequisites for and seats are still available in the seminar.
- A professor requests a seminar list for every course he teaches.
- A researcher applies for a research grant, but only receives partial funding for her project.
- A professor submits student marks to the system. These marks may be for exams, tests, or assignments.
- A student wants to drop a seminar the day after the drop date.
- A student requests a printed copy of his transcript, so he can include copies of it with his résumé.

You can take either approach, or combine the two, if you like, but the main goal is to end up with a lot of information regarding the behavioral aspects of your system. The next step is to group these aspects, by similarity, into use cases. ***Remember that a use case provides a service of measurable value to an actor.***

Each service should be cohesive; in other words, it should do one thing that makes sense. For example, you would not want a use case called *Support Students* that does everything student needs, such as letting them enroll in courses, drop courses, pay fees, and obtain course information. That is simply too much to handle all at once. Instead, you should identify several use cases, one for each service provided by the system.

3.2.3 Writing/Documenting an Essential Use Case

A use case is a sequence of actions that provide a measurable value to an actor. Another way to look at it is a use case describes a way in which a real-world actor interacts with the system. An essential use case, sometimes called a ***business use case***, is a simplified, abstract, generalized use case that captures the intentions of a user in a technology- and implementation-independent manner. A business use case is often more focused on the business process and existing technology concerns are often brought into them. Think of them as somewhere in between essential and system use cases.

Figure 3.4 presents a fully documented essential use case. Notice how brief and to the point the language is. There is not a lot of detail because you only need to get the basic idea across. The language of the application domain and of users is used, comprising a simplified, generalized, abstract, technology- free and implementation-independent description of one task or interaction. **An essential use case** is complete, meaningful, and well designed from the point of view of users in some role or roles in relation to a system and that embodies the purpose or intentions underlying the interaction.

Enroll in Seminar

ID: UC 1

Preconditions:

- The student is enrolled in the university.

Postconditions:

- None

Actor(s)	**Response**
Student identifies himself	Verifies eligibility to enroll via *BR1 Determine Eligibility to Enroll*. Indicate available seminars
Choose seminar	Validate choice via *BR2 Determine Student Eligibility to Enroll in*

	a Seminar. Validate schedule fit via *BR3 Validate Student Seminar Schedule* Calculates fees via *BR 4 Calculate Student Fees* and *BR5Calculate Taxes for Seminar*. Summarize fees Request confirmation
Confirm enrollment	Enroll student in seminar Add fees to student bill Provide confirmation of enrollment

Figure 3.4: *Enroll in Seminar* as an essential use case.

Advanced Example:

Preconditions describe the state or status the system must be in, prior to the execution of a use case. What must be true before the use case can execute?

Post conditions describe the state or status of the system as a result of the execution of the use case.

Use Case ID	UC-100
Use Case	Withdraw Funds
Actors	(P) Customer
Description	Customer logs in, selects withdraw funds, enters an amount and receives cash and receipt
Pre-conditions	Welcome screen is on
Flow of Events	1. Customer slides card in and out 2. Machine prompts customer for password 3. Customer enters password 4. System authenticates customer 5. System presents user with a choice menu 6. Customer selects Withdraw Funds option 7. System asks customer to select account

	8. Customer selects account 9. System asks customer for amount to withdraw 10. Customer enters amount 11. System dispenses cash and prints receipt 12. System logs customer out
Post-conditions	Welcome screen is back on
Alternative Flows	Customer may not be authenticated; customer may not have sufficient funds; machine may not have enough cash
Priority	High
Non-Functional Requirements	Cash dispensed within 10 seconds after amount is entered
Assumptions	Customer speaks English
Source	Bank's Operational Procedures Manual

Essential use cases (Use case description) are typically written in a two-column format: the column on the left indicating what the actors do, and the column on the right, the response to their actions. The actor(s) will do something and receive one or more responses to that action. As you can see the flow of the use case is apparent from the spacing of the actions and responses, although you may decide to number the steps to make it more apparent.

Fig. 3.4 and Fig.3.5 indicates a unique identifier for the use case as well as its preconditions and post conditions. These three pieces of information are optional although very useful.—for example, the **use case** itself references business rules, each of which has a unique identifier. The **preconditions**, if any, indicate what must be true before this use case is allowed to run. The **post conditions**, if any, indicate what will be true once the use case finishes successfully.

3.3 Essential User Interface Prototyping

The UI is the portion of software with which a user directly interacts. An essential UI prototype is a low-fidelity model, or prototype, of the UI for your system. It represents the general ideas behind the UI, but not the exact details. Essential UI prototypes represent UI requirements in a technology-independent manner, just as essential use case models do for behavioral requirements. An ***essential UI prototype is effectively the initial state—the beginning point—of the UI prototype for your system.*** It models UI requirements; requirements evolved through analysis and design to result in the final user interface for your system.

Two basic differences exist between essential UI prototyping and traditional UI prototyping. First, with essential UI modeling the goal is to focus on your users and their usage of the system, not system features. This is one of the reasons you want to perform essential use case modeling and essential UI prototyping in tandem: they each focus on usage. Second, your prototyping tools are simple, including whiteboards, flip chart paper, and sticky notes. Right now, you should be focused on requirements, not design; therefore, you do not currently want to use technology-based prototyping tools. Understand the problem first, and then solve it.

Tip **Active Stakeholder Participation Is Crucial** Project stakeholders must be available to provide requirements, to prioritize them, and to make decisions in a timely manner. It is critical that your project stakeholders understand this concept and are committed to it from the beginning of any project.

So how do you use sticky notes and flip chart paper to create an essential UI prototype? Let us start by defining several terms. A major UI element represents a large-grained item, potentially a screen, HTML page, or report. A minor UI element represents a small-grained item, widgets such as user input fields, menu items, lists, or static text fields such as labels. When a team is creating an essential UI prototype, it iterates between the following designs:

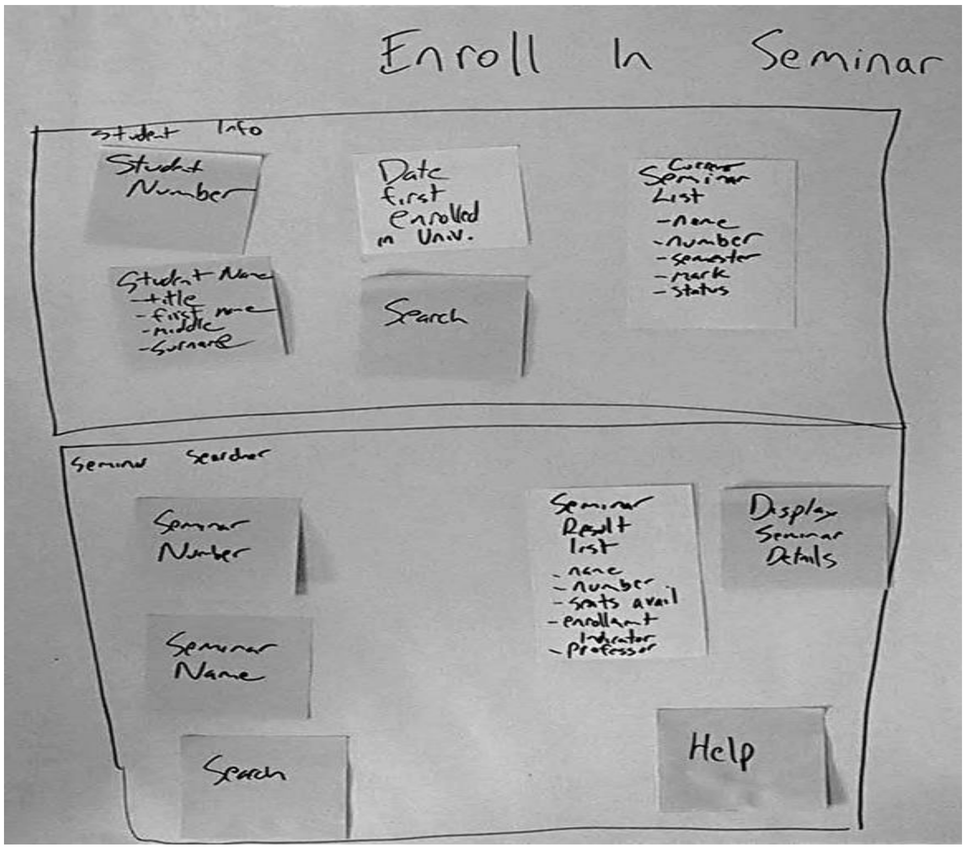
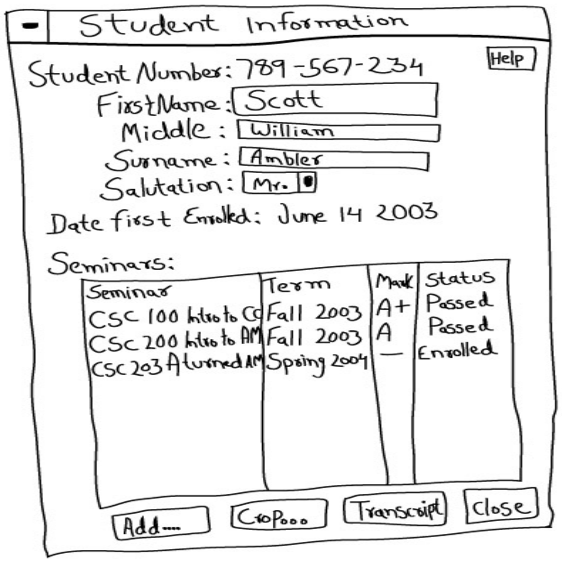


Figure 3.5: An essential UI prototype to enroll a student in a seminar.



Student Information

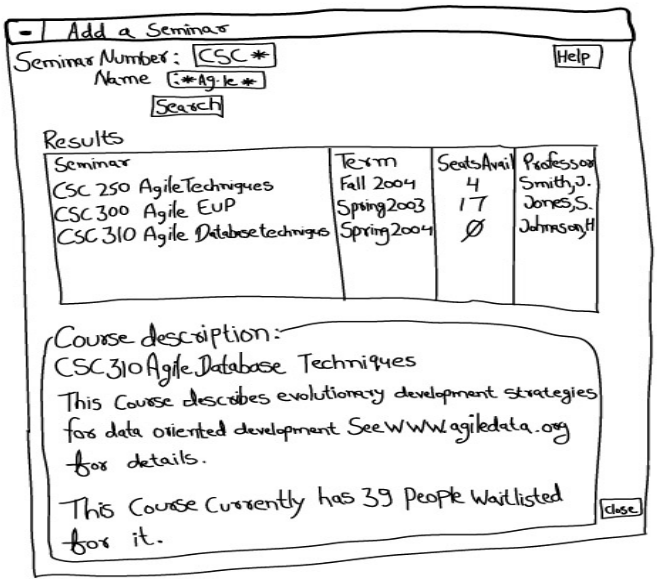
Help

Student Number: 789-567-234
FirstName: Scott
Middle: William
Surname: Ambler
Salutation: Mr.
Date first Enrolled: June 14 2003

Seminars:

Seminar	Term	Mark	Status
CSC 100 Intro to C(	Fall 2003	A+	Passed
CSC 200 Intro to AM	Fall 2003	A	Passed
CSC 203 Advanced AM	Spring 2004	—	Enrolled

Add... Drop... Transcript Close



Add a Seminar

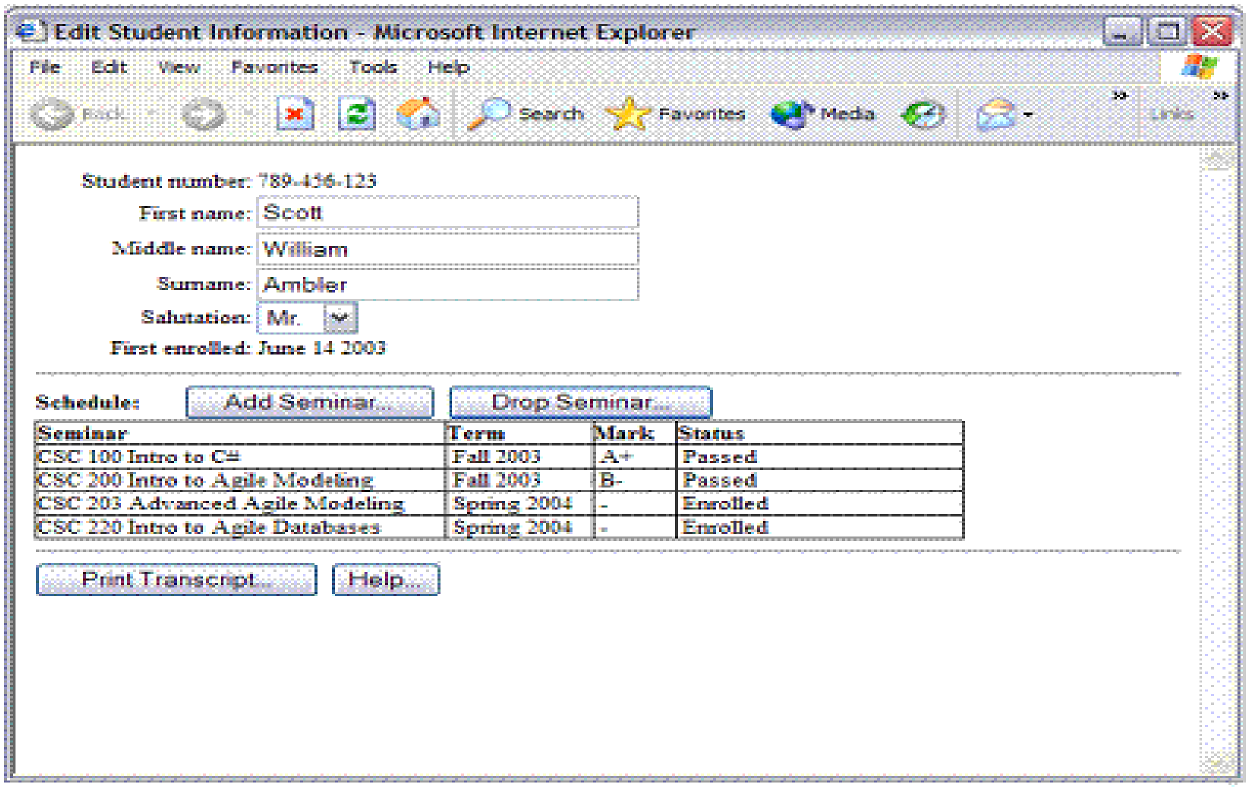
Seminar Number: CSC*
Name: *Agile*
Search
Help

Results

Seminar	Term	Seats Avail	Professor
CSC 250 Agile Techniques	Fall 2004	4	Smith,J.
CSC 300 Agile EUP	Spring 2003	17	Jones,S.
CSC 310 Agile Database techniques	Spring 2004	Ø	Johnson,H

Course description:
CSC310 Agile Database Techniques
This Course describes evolutionary development strategies for data oriented development See WWW.agiledata.org for details.
This Course Currently has 39 People Waitlisted for it.

Close


Edit Student Information - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Media Links
Add Seminar...
Drop Seminar...
Print Transcript...
Help...

Student number:	789-456-123
First name:	Scott
Middle name:	William
Surname:	Ambler
Salutation:	Mr.
First enrolled:	June 14 2003

Schedule:

Seminar	Term	Mark	Status
CSC 100 Intro to C#	Fall 2003	A+	Passed
CSC 200 Intro to Agile Modeling	Fall 2003	B-	Passed
CSC 203 Advanced Agile Modeling	Spring 2004	-	Enrolled
CSC 220 Intro to Agile Databases	Spring 2004	-	Enrolled

Figure 3.6: Revised essential UI prototype to enroll a student in a seminar.

Always remember that your project stakeholders are the official source of requirements, not developers. If you identify some new functionality that you think is required you need to convince your stakeholder that it is a good idea, have them prioritize it, and add the new requirement to the stack.

3.4 Domain modeling with class responsibility collaborator (CRC) cards

A class responsibility collaborator model is a collection of standard index cards that have been divided into three sections. A **class** represents a collection of similar objects, a **responsibility** is something that a class knows or does, and a **collaborator** is another class that a class interacts with to fulfill its responsibilities.

Figure 3.7 depicts a small CRC model for the university information system. These cards were identified by working closely with a project stakeholder who understood this part of the domain.

Because CRC cards are a simple technique you will quickly discover that your stakeholders will learn how to work with them, particularly for high-level conceptual modeling.

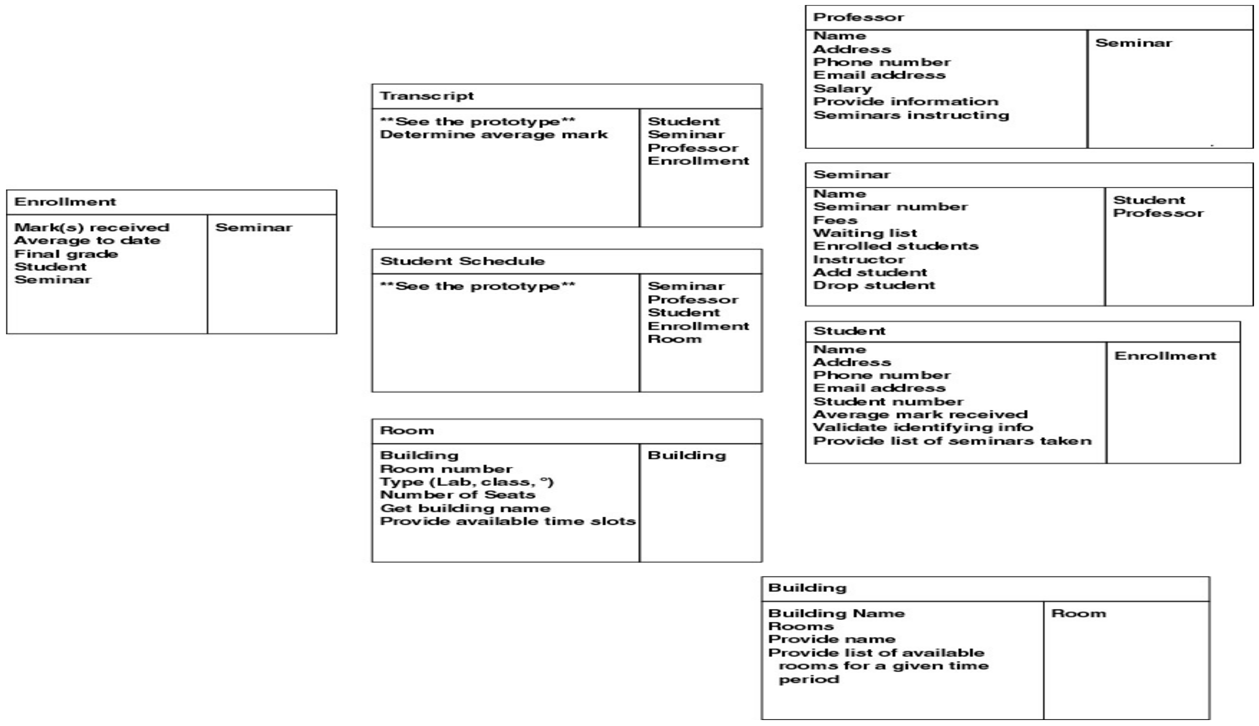


Figure 3.7: Some of the CRC cards for a university information system.

CRC modeling often becomes more hands-on because we are using physical cards. The cards often work better than the diagrams because they are easy to understand and work with.

3.5 Developing a supplementary Specification

The true goal of requirements engineering is not to create documentation; it is to convey ideas from project stakeholders to developers.

A supplementary specification is a document that contains requirements not contained directly in your use cases. This often includes business rules, technical requirements, and constraints. The best way to think of a supplementary specification is that it is a container into which you place other requirements. In my opinion supplementary specifications are not models in their own right but instead are the documentation of models.

Modern-day businesses are complex, as are the systems that support them. To make matters worse, there are various aspects to the complexities. For example there are business complexities that you need to understand, technical complexities, even constraints imposed on you by outside authorities. The implication is that you need techniques with which to explore these complexities, techniques that produce something that we call "supplementary requirements" in Table 3.1 summarizes the supplementary requirements artifacts described in this chapter.

Table 3.1: Supplementary Requirements Artifacts

Artifact	**Description**
Business rules	A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business. Business rules often supplement usage or user interface requirements.
Constraints	A constraint is a restriction on the degree of freedom you have in providing a solution. Constraints will supplement other development artifacts, in particular architecture and design-oriented models.
Glossary	A glossary is a collection of definitions that supplements a wide range of development artifacts by defining a common business and technical vocabulary.
Technical requirements	A technical requirement pertains to the technical aspects that your system must fulfill, such as performance-related, reliability, and availability issues. Technical requirements are often the main driver of your technical architecture.

3.5.1 Business Rules

A business rule defines or constrains one aspect of your business that is intended to assert business structure or influence the behavior of your business. Business rules often focus on access control issues; for example, professors are allowed to input and modify the marks of the students taking the seminars they instruct, but not the marks of students in other seminars. Business rules may also pertain to business calculations, for example, how to convert a percentage mark (for example, 91 percent) that a student receives in a seminar into a letter grade (for example, A-). Some business rules focus on the policies of your organization; perhaps the university policy is to expel for one year anyone who fails more than two courses in the same semester.

Figure 3.8 summarizes several examples of business rules. Notice how each business rule has a unique identifier. The convention is to use the format of BR#, but you are free to set your own numbering approach. The unique identifier enables you to refer easily to business rules in other development artifacts, such as class models and use cases.

- BR7 Tenured professors may administer student grades.
- BR8 Teaching assistants who have been granted authority by a tenured professor may administer student grades.
- BR9 Table to convert between numeric grades and letter grades.
- BR6 All master's degree programs must include the development of a thesis.

Figure 3.8: Example business rules (summarized).

In some situations you will discover that business rules can be described very simply, perhaps with a single sentence. In other situations this is not the case. Figure 3.8 presents one way to fully document BR12. There are several sections in this figure:

Name:	Tenured professors may administer student grades
Identifier:	BR7
Description:	Only tenured professors are granted the ability to initially input, modify, and delete grades students receive in the seminars that they and they only instruct. They may do so only during the period a seminar is active.
Example:	Dr. Bruce, instructor of "Biology 301 Advanced Uses of Gamma Radiation," may administer the marks of all students enrolled in that seminar, but not those enrolled in "Biology 302 Effects of Radiation on Arachnids," which is taught by Dr. Peters.
Source:	University Policies and ProceduresDoc ID: U1701Publication date: August 14, 2000

Figure 3.9: A fully documented business rule.

- **Name.** The name should give you a good idea about the topic of the business rule.

- **Description.** The description defines the rule exactly. Use text to describe this rule it is quite common to see diagrams such as flow charts or UML activity diagrams used to describe an algorithm.
- **Example (optional).** An example of the rule is presented to help clarify it.
- **Source (optional).** The source of the rule is indicated so it may be verified (it is quite common that the source of a rule is a person, often one of your project stakeholders, or a team of people).

Basic principles of the business rule approach

- Be written and made explicit;
- Be expressed in plain language;
- Exist independent of procedures and workflows (e.g., multiple models);
- Build on facts, and facts should build on concepts as represented by terms
- Be motivated by identifiable and important business factors;
- Be accessible to authorized parties (e.g., collective ownership);
- Be single sourced;
- Be specified directly by those people who have relevant knowledge (e.g., active stakeholder participation); and

3.5.2 Technical Requirements

A technical requirement pertains to the technical aspects that your system must fulfill, such as performance-related issues, reliability issues, and availability issues. These types of requirements are often called service-level requirements or nonfunctional requirements. Examples of technical requirements are presented in Fig. 3.10. As you can see, technical requirements are summarized in a manner similar to that of business rules: they have a name and a unique identifier (the convention is to use the format TR#, where TR stands for technical requirement). You document technical requirements in the same manner as business rules, including a description, an example, a source, references to related technical requirements, and a revision history.

Technology changes quickly and often requirements based on technology change just as quickly. An example of a pure technical requirement is that an application be written in Java or must run on the XYZ computer. Whenever you have a requirement based purely on technology, try to determine the real underlying business needs being expressed. To do this, keep asking why your application must meet a requirement. For example, when asked why your application must be written in Java, the reply was it must run on the Internet. When asked why it must run on the Internet, the reply was your organization wants to take orders for its products and services on the Internet. The real requirement is to sell things to consumers at their convenience; one technical solution to this need (and a good one) is to write that component in Java that can be accessed via the Internet. A big difference exists between having to write the entire application in Java and having to support the sales of some products and services to consumers over the Internet.

- TR34 The system shall be available 99.99 percent of the time for any 24- hour period.
- TR78 A seminar search will occur within less than three seconds 95 percent of the time.
- TR79 A seminar search will occur within no more than ten seconds 99 percent of the time.

Figure 3.10: Potential technical requirements for the university system.

3.5.3 Constraints

A constraint is a restriction on the degree of freedom you have in providing a solution. Constraints are effectively global requirements, such as limited development resources or a decision by senior management that restricts the way you develop a system. Constraints can be economic, political, technical, or environmental and pertain to your project resources, schedule, target environment, or to the system itself. Figure 3.11 presents several potential constraints for the university system. Like business rules and technical requirements, constraints are documented in a similar manner.

- C24 The system will work on our existing technical infrastructure—no new technologies will be introduced.
- C56 The system will only use the data contained in the existing corporate database.
- C73 The system shall be available 99.99 percent of the time for any 24-hour period.
- C76 All master's degree programs must include the development of a thesis.

Figure 3.11: Potential constraints for the university system.

An interesting thing about Fig. 3.11 is that it contains two constraints, C73 and C76, which were previously identified as a technical requirement and a business rule, respectively. Constraints can be a little confusing because of their overlap with business rules and technical requirements. Don't worry about it. The important thing is that you have identified the requirement.

As with business rules, you identify constraints as you are developing other artifacts, such as your use case model and user-interface model.

3.5.4 Glossaries

A glossary is a collection of definitions of terms. You may want to include both technical and business terms in your glossary. Although you may understand what terms such as XP, C#, J2EE, and application server all mean your stakeholders likely do not. Similarly your stakeholders may understand what business terms such as convocation, grant, and transcript mean, but some developers may not. A glossary that includes both the relevant technical and business terminology goes a long way to improving the communications between developers and users—if you do not understand each other's language you cannot communicate effectively.

3.5.5 Identify and documenting Change Cases

Change cases are used to describe new potential requirements for a system or modifications to existing requirements. Change cases are modeled in a simple manner. You describe the potential change to your existing requirements, indicate the likeliness of that change occurring, and indicate the potential impact of that change. Figure 3.12 presents three change cases, two potential changes motivated by technical innovation—in this case the need to support several platforms and the transition to a new database vendor—and a third by a change in your business environment. Notice how both change cases are short and to the point, making them easy-to-understand. The name of a change case should describe the potential change itself.

Change case: Need to support both Linux and Microsoft platforms.

Likelihood: Very likely to happen for application and database servers within six months; medium probability that Linux will be adopted by the school for desktop machines.

Impact: Unknown. Currently application servers for other applications run Microsoft-based operating systems and will likely continue to do so. New servers will likely have Linux installed. Desktop impact is hard to quantify as the Linux market is evolving rapidly. Should be re-evaluated six months from now.

Change case: Need to support new database vendor.
Likelihood: Medium. The school is currently renegotiating the contract with our existing vendor.
Impact: Potentially large if SQL is hard-coded into the application.
Change case: The University will open a new campus.
Likelihood: Certain. It has been announced that a new campus will be opened in two years across town.
Impact: Large. Students will be able to register in classes at either campus. Some instructors

will teach at both campuses. Some departments, such as Computer Science and Philosophy, are slated to move their entire programs to the new campus. The likelihood is great that most students will want to schedule courses at only one of the two campuses, so we will need to make this easy to support.

Figure 3.12: Examples of change cases for the university.

Change cases can be identified throughout the course of your overall development efforts. Change cases are often the result of brainstorming with your project stakeholders.