Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 4:

Developing Requirements

Lecture 4

4.1 Domain Analysis

The process by which a software engineer learns about the domain to better understand the problem:

- The *domain* is the general field of business or technology in which the clients will use the software
- A *domain expert* is a person who has a deep knowledge of the domain

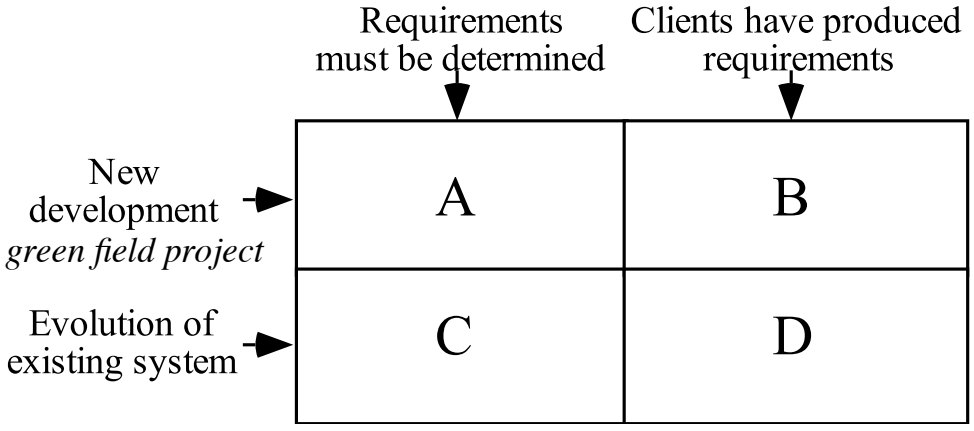
Benefits of performing domain analysis:

- Faster development
- Better system
- Anticipation of extensions

Domain Analysis document

A. Introduction
B. Glossary
C. General knowledge about the domain
D. Customers and users
E. The environment
F. Tasks and procedures currently performed
G. Competing software
H. Similarities to other domains

4.2 The Starting Point for Software Projects



4.3 Defining the Problem and the Scope

A problem can be expressed as:

- A *difficulty* the users or customers are facing,
- Or as an *opportunity* that will result in some benefit such as improved productivity or sales.

The solution to the problem normally will entail developing software

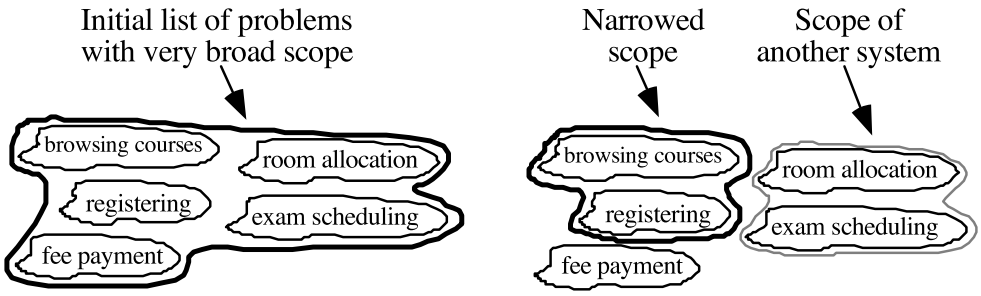
A good problem statement is short and succinct

Defining the Scope

Narrow the *scope* by defining a more precise problem

- List all the things you might imagine the system doing
  - —Exclude some of these things if too broad
  - —Determine high-level goals if too narrow

Example: A university registration system



4.4 What is a Requirement ?

It is a statement describing either

- 1) an aspect of what the proposed system must do,
- or 2) a constraint on the system's development.
- In either case it must contribute in some way towards adequately solving the customer's problem;
- the set of requirements as a whole represents a negotiated agreement among the stakeholders.

A collection of requirements is a *requirements document*.

4.5 Types of Requirements

Functional requirements

- Describe *what* the system should do

Quality requirements

- C*onstraints* on the design to meet specified levels of quality

Platform requirements

- C*onstraints* on the environment and technology of the system

Process requirements

- C*onstraints* on the project plan and development methods

Functional Requirements

- What *inputs* the system should accept
- What *outputs* the system should produce
- What data the system should *store* that other systems might use
- What *computations* the system should perform
- The *timing and synchronization* of the above

Quality Requirements

All must be verifiable

Examples: Constraints on

- Response time
- Throughput
- Resource usage
- Reliability
- Availability
- Recovery from failure
- Allowances for maintainability and enhancement
- Allowances for reusability

4.6 Use-Cases: describing how the user will use the system

A *use case* is a typical sequence of actions that a user performs in order to complete a given task

- The objective of *use case analysis* is to model the system from the point of view of

  … how users interact with this system

  … when trying to achieve their objectives.

  It is one of the key activities in requirements analysis
- A *use case model* consists of
  - a set of use cases
  - an optional description or diagram indicating how they are related

Use cases

A use case should

- Cover the *full sequence of steps* from the beginning of a task until the end.
- Describe the *user's interaction* with the system ...
  - Not the computations the system performs.
- Be written so as to be as *independent* as possible from any particular user interface design.
- Only include actions in which the actor interacts with the computer.
  - Not actions a user does manually

Scenarios

A scenario is an *instance* of a use case

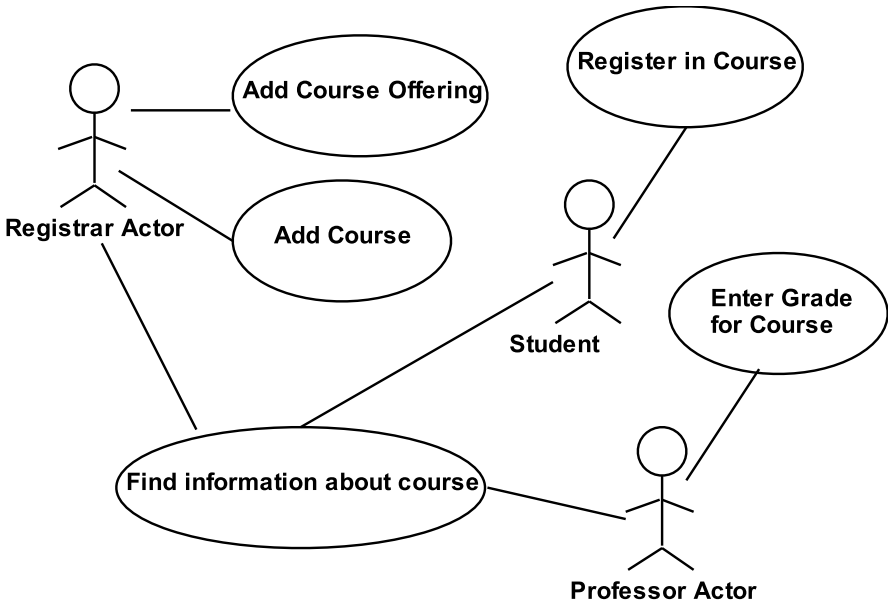
- A *specific occurrence* of the use case
  - —a specific actor ...
  - —at a specific time ...
  - —with specific data.

How to describe a single use case

A. Name: Give a short, descriptive name to the use case.
B. Actors: List the actors who can perform this use case.
C. Goals: Explain what the actor or actors are trying to achieve.
D. Preconditions: State of the system before the use case.
E. Summary: Give a short informal description.
F. Related use cases.
G. Steps: Describe each step using a 2-column format.
H. Postconditions: State of the system in following completion.

A and G are the most important

Use case diagrams



Extensions

- Used to make *optional* interactions explicit or to handle *exceptional* cases.
- Keep the description of the basic use case simple.

Generalizations

- Much like superclasses in a class diagram.
- A generalized use case represents *several similar* use cases.
- One or more specializations provides details of the similar use cases.

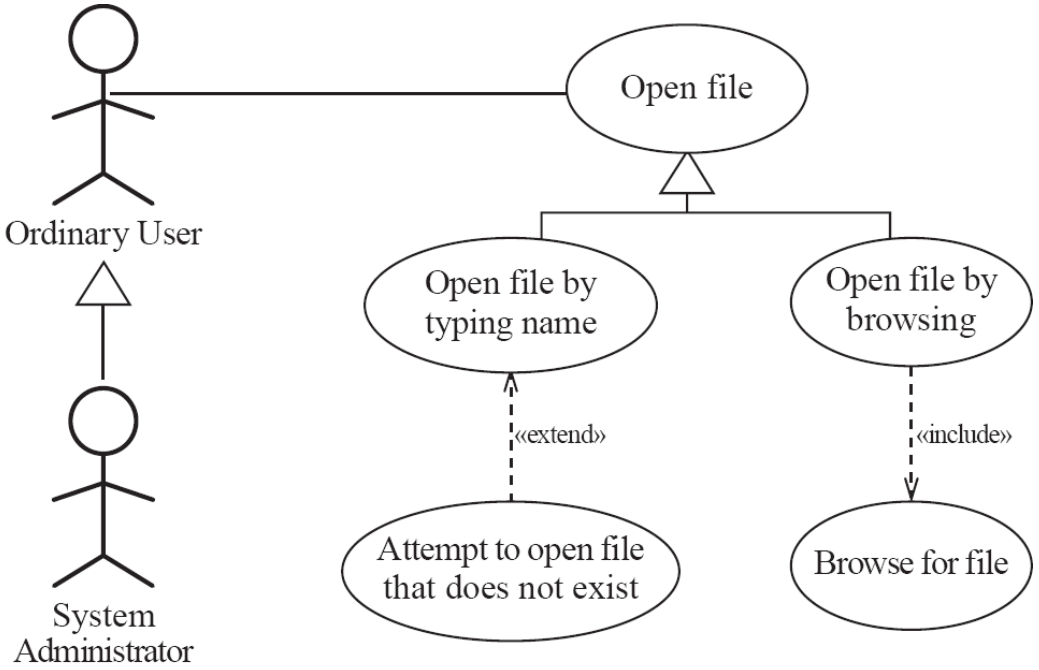
Inclusions

- Allow one to express *commonality* between several different use cases.

- Are included in other use cases
  - —Even very different use cases can share sequence of actions.
  - —Enable you to avoid repeating details in multiple use cases.

- Represent the performing of a *lower-level task* with a lower-level goal.

Example of generalization, extension and inclusion



Example description of a use case

Use case: **Open file**

Related use cases:
Generalization of:
- Open file by typing name
- Open file by browsing

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3. Specify filename	
4. Confirm selection	5. Dialog disappears

Example (continued)

Use case: **Open file by typing name**

Related use cases:
Specialization of: Open file

Steps:

Actor actions	**System responses**
1. Choose ‘Open…’ command	2. File open dialog appears
3a. Select text field	
3b. Type file name	
4. Click ‘Open’	5. Dialog disappears

The modeling processes: Choosing use cases on which to focus

- Often one use case (or a very small number) can be identified as *central* to the system
  - —The entire system can be built around this particular use case
- There are other reasons for focusing on particular use cases:
  - —Some use cases will represent a high *risk* because for some reason their implementation is problematic
  - —Some use cases will have high political or commercial value

The benefits of basing software development on use cases

They can

- Help to define the *scope* of the system
- Be used to *plan* the development process
- Be used to both develop and validate the requirements
- Form the basis for the definition of test cases
- Be used to structure user manuals

Use cases must not be seen as a panacea

- The use cases themselves must be validated
  - —Using the requirements validation methods.
- Some aspects of software are not covered by use case analysis.
- Innovative solutions may not be considered.

4.7 Some Techniques for Gathering and Analysing Requirements

Observation

- Read documents and discuss requirements with users
- Shadowing important potential users as they do their work
  - —ask the user to explain everything he or she is doing
- Session videotaping

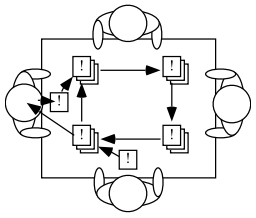
Interviewing

- Conduct a series of interviews
  - —Ask about specific details
  - —Ask about the stakeholder's vision for the future
  - —Ask if they have alternative ideas
  - —Ask for other sources of information
  - —Ask them to draw diagrams

Gathering and Analysing Requirements...

Brainstorming

- Appoint an experienced moderator
- Arrange the attendees around a table
- Decide on a 'trigger question'
- Ask each participant to write an answer and pass the paper to its neighbour



Joint Application Development (JAD) is a technique based on intensive brainstorming sessions

Gathering and Analysing Requirements...

Prototyping

- The simplest kind: *paper prototype*.
  - —a set of pictures of the system that are shown to users in sequence to explain what would happen
- The most common: a mock-up of the system's UI
  - —Written in a rapid prototyping language
  - —Does *not* normally perform any computations, access any databases or interact with any other systems
  - —May prototype a particular aspect of the system

Gathering and Analysing Requirements...

Use case analysis

- Determine the classes of users that will use the facilities of this system (actors)
- Determine the tasks that each actor will need to do with the system

4.8 Types of Requirements Document

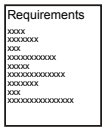
Two extremes:

An informal outline of the requirements using a few paragraphs or simple diagrams

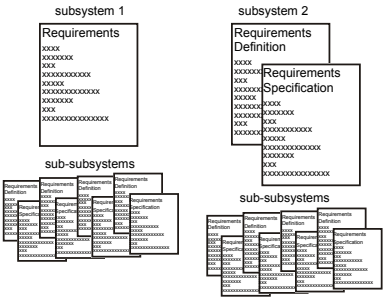
requirements *definition*

A long list of specifications that contain thousands of pages of intricate detail

requirements *specification*



- Requirements documents for large systems are normally arranged in a hierarchy



Level of detail required in a requirements document

- How much detail should be provided depends on:
  - —The size of the system
  - —The need to interface to other systems
  - —The readership
  - —The stage in requirements gathering
  - —The level of experience with the domain and the technology
  - —The cost that would be incurred if the requirements were faulty

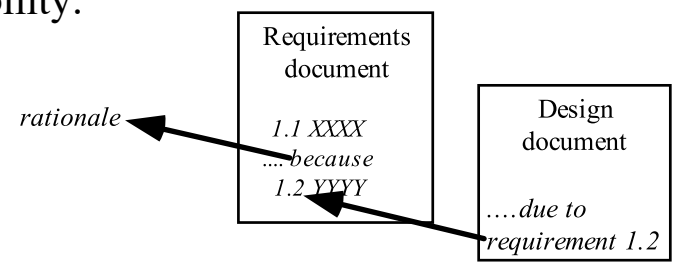
4.9 Reviewing Requirements

- **Each individual requirement should**
  - Have **benefits that outweigh the costs** of development
  - Be **important** for the solution of the current problem
  - Be expressed using a **clear and consistent notation**
  - Be **unambiguous**
  - Be **logically consistent**
  - Lead to a system of **sufficient quality**
  - Be **realistic** with available resources
  - Be **verifiable**
  - Be uniquely **identifiable**
  - **Not over-constrain the design** of the system

Requirements documents...

- The document should be:
  - sufficiently complete
  - well organized
  - clear
  - agreed to by all the stakeholders

- Traceability:



Requirements document...

A. Problem
B. Background information
C. Environment and system models
D. Functional Requirements
E. Non-functional requirements

4.10 Managing Changing Requirements

Requirements change because:

- Business process changes
- Technology changes
- The problem becomes better understood

Requirements analysis never stops

- Continue to interact with the clients and users
- The benefits of changes must outweigh the costs.
  - —Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.
  - —Larger-scale changes have to be carefully assessed
    - Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery
- Some changes are enhancements in disguise
  - —Avoid making the system *bigger*, only make it *better*

4.13 Difficulties and Risks in Domain and Requirements Analysis

- Lack of understanding of the domain or the real problem
  - —*Do domain analysis and prototyping*
- Requirements change rapidly
  - —*Perform incremental development, build flexibility into the design, do regular reviews*
- Attempting to do too much
  - —*Document the problem boundaries at an early stage, carefully estimate the time*
- It may be hard to reconcile conflicting sets of requirements
  - —*Brainstorming, JAD sessions, competing prototypes*
- It is hard to state requirements precisely
  - —*Break requirements down into simple sentences and review them carefully, look for potential ambiguity, make early prototypes*