Softwaretechnik

Lecture 03: From Requirements to Definition

Peter Thiemann

University of Freiburg, Germany

SS 2012

Requirements Engineering

comprises methods, means of description, and tools to discover, analyze, and formulate requirements of software systems

- **requirements analysis** (*Systemanalyse*)
- **requirements specification** (*Produktdefinition*)

Requirements

- **Functional requirements**
  - inputs and their constraints
  - functions of the system
  - outputs (reactions)
- **Nonfunctional requirements**
  - runtime
  - memory
  - standards

Requirements

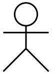
- **Requirements on realization**
  - software / hardware
  - devices
  - interfaces
  - facilities (OS, computers, ...)
  - documentation
- **Requirements on testing, installation, support**
- **Requirements on construction of the system**
  - approach
  - resources (personal, cost, deadlines)
  - rules, standards

Systematic Investigation of Functional Requirements

- Inside-out methods
  modeling starts from product internals
  (rarely applicable for new products)
- Outside-in methods
  modeling starts from environment of product
  - actors and use cases (use case diagram, UML)
  - interfaces and data flows (context diagram)

Use Cases and Use Case Diagrams

Jacobson, UML

Actor 

- participates directly in a process
- stands for a role
    - natural person
    - unit of organization
    - external system

Use Cases

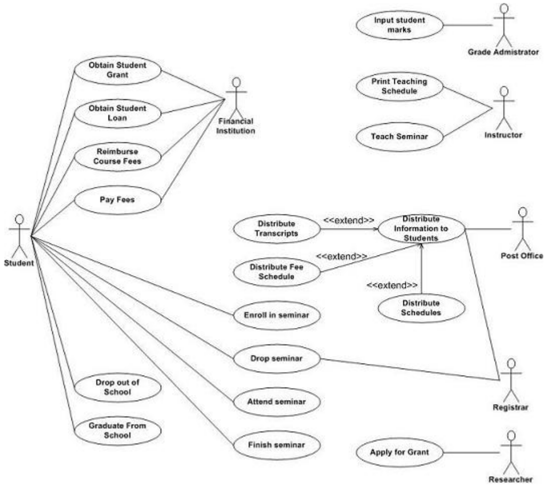
Use case [Definition]

- a sequence of actions
- performed by one actor
- to achieve a particular goal

two forms:

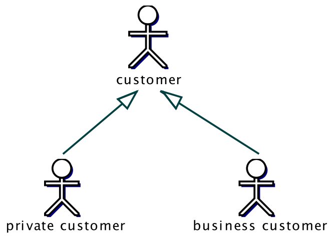
- graphical (UML diagram)
- textual (with templates)

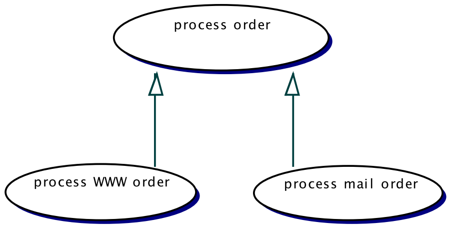
Example Use Case Diagram



Source: http://www.agilemodeling.com/images/models/useCaseDiagram.jpg

Generalization





- generalization
- concrete and abstract use cases
- concrete and abstract actors

Use Case Textual Template

Use case:	name
Goal:	achieved by successful execution
Category:	primary, secondary, optional
Precondition:	
Postcondition/success:	
Postcondition/failure:	
Actors:	
Trigger:	
Description:	numbered tasks
Extensions:	wrt previous tasks
Alternatives:	wrt tasks

Use Case Guidelines

- Outside view — System as black box
- No implementation specifics
- No UI specifics
- **Primarily text**

Tools

- `http://www.umlet.com/`
  UML diagram drawing — standalone and in Eclipse
- `http://yuml.me/`
  online drawing of use case and class diagrams (UML)
- `http://www.gliffy.com/flowchart-software/`
  flowcharts and DFD

Related Approaches

User Stories

A user story is a very high-level definition of a requirement, containing just enough information so that the developers can produce a reasonable estimate of the effort to implement it. [Scott Ambler `http://www.agilemodeling.com/artifacts/userStory.htm`]

- Very slim, very high-level, often just one sentence.
- Informal, but proposed formal style [Mike Cohn]:
  As a (role) I want (something) so that (benefit).

Example User Stories

- Students can purchase monthly parking passes online.
- Parking passes can be paid via credit cards.
- Professors can input student marks.
- Students can obtain their current seminar schedule.
- Students can order official transcripts.
- Students can only enroll in seminars for which they have prerequisites.

- As a student I want to purchase a monthly parking pass so that I can drive to school.
- As a student I want to obtain my current seminar schedule so that I can follow my classes.

User Stories Guidelines

- Authors
- Tools
- Size
- Priority
- Traceability

Related Approaches

Usage Scenarios

A usage scenario, or scenario for short, describes a real-world example of how one or more people or organizations interact with a system. They describe the steps, events, and/or actions which occur during the interaction. Usage scenarios can be very detailed, indicating exactly how someone works with the user interface, or reasonably high-level describing the critical business actions but not the indicating how they are performed. [Scott Ambler `http://www.agilemodeling.com/artifacts/usageScenario.htm`]

- Further elaboration of a use case.
- Scenario $\sim$ path through a use case.

Example High-Level Scenario

Scenario: ATM banking for the week

1. Sally Jones places her bank card into the ATM.
2. Sally successfully logs into the ATM using her personal identification number.
3. Sally deposits her weekly paycheck of \$350 into her savings account.
4. Sally pays her phone bill of \$75, her electricity bill of \$145, her cable bill of \$55, and her water bill of \$85 from her savings account.
5. Sally attempts to withdraw \$100 from her savings account for the weekend but discovers that she has insufficient funds.
6. Sally withdraws \$40 and gets her card back.

Source: agilemodeling.com

Sorry, Sally...

Source: http://www.autograph-gallery.co.uk/acatalog/F79_Sally_Field.html

Example Detailed Scenario

Scenario: A successful withdrawal attempt at an automated teller machine (ATM)

1. John Smith presses the 'Withdraw Funds' button
2. The ATM displays the preset withdrawal amounts ($20, $40, . . .)
3. John chooses the option to specify the amount of the withdrawal
4. The ATM displays an input field for the withdrawal amount
5. John indicates that he wishes to withdraw $50 dollars
6. The ATM displays a list of John's accounts, a checking and two savings accounts
7. John chooses his checking account
8. The ATM verifies that the amount may be withdrawn from his account
9. The ATM verifies that there is at least $50 available to be disbursed from the machine
10. The ATM debits John's account by $50
11. The ATM dispenses $50 in cash
12. The ATM displays the 'Do you wish to print a receipt' options
13. John indicates 'Yes'
14. The ATM prints the receipt

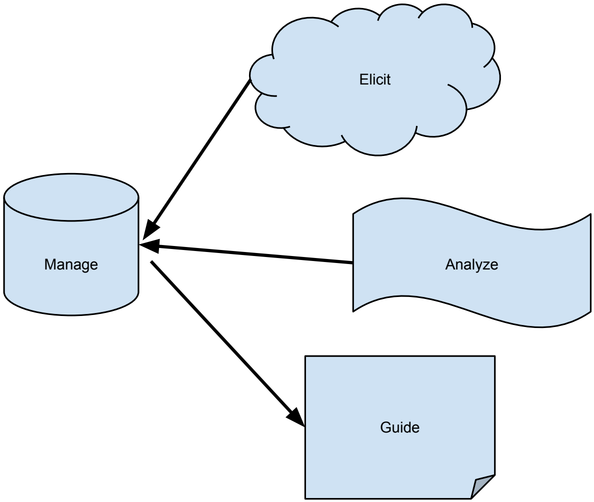
Source: agilemodeling.com

Perspective on Changing Requirements

- Produce high quality requirements (see checklist in CC2)
- Advertize the cost of requirements changes
- Establish a change-control procedure
- Anticipate changes
- Consider the business value of requirements
- Cancel a project with bad or frequently changing requirements

Functional Requirements

Four Activities



The Analysis Problem

- Is it complete?
- Is it sound?
- Did I really understand it?
- Do different people say different things?
- Am I too abstract?
- Not sufficiently abstract?

Formal Methods for Analysis

What are Formal Methods?

- Formal = Mathematical
- Methods = Structured Approaches

Formal Methods in Industrial Use

- Hardware: no major chip is developed without it
- Software
  - software verification and model checking
  - Design by Contract
  - Blast, Atelier B, Boogie
- Design: UML's OCL, BON, Z, state charts
- Testing
  - automatic test generation
  - parallel simulation

What's Disturbing about the 'M' Word?

- Very abstract
- Too many greek letters
- Difficult to learn and read
- Cannot communicate with normal person

Useful Mathematics for Requirements

- Set theory
- Functions and relations
- First-order predicate logic
- Before-after predicates

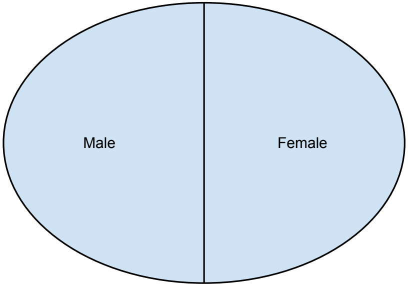
Set Theory

All dogs are male or female

$\text{Dogs} = \text{Male} \cup \text{Female}$

No dog is male and female at once

$\text{Male} \cap \text{Female} = \emptyset$



Functions and Relations

Every customer must have a personal assistant

attendants : Customers $\rightarrow$ Employees

Every customer has a set of accounts

accountsOf : Customers $\rightarrow$ P(Accounts)

First-Order Predicate Logic

Everybody who works on a Sunday must have a permit

$\forall$ p$\in$Employees. workOnSunday(p) $\Rightarrow$ hasPermit(p)

Every customer must have at least one account

$\forall$ c$\in$Customers. $\exists$ a$\in$Accounts. a$\in$accountsOf(c)

Before-After Predicates

People who enter the building must carry their ID with them. When entering, they have to leave their ID at the registration desk.

```
enterBuilding(p) =
PRE
        hasAuthorization(p)
        carriesPassport(p)
THEN
        peopleInBuilding := peopleInBuilding $\cup$ { p }
        passportsAtDesk := passportsAtDesk $\cup$ { passportOf(p) }
        not carriesPassport(p)
END
```

Advantanges of Using Mathematics

- Short notation
- Forces precision
- Identifies ambiguity
- Clean form of communication
- Makes you ask the right questions

Mathematical Notation is Concise

For every ticket that is issued, there has to be a single person that is allowed to enter. This person is called the owner of the ticket.

vs.

$ticketOwner : IssuedTickets \rightarrow Persons$

Mathematical Notation Enforces Precision

On red traffic lights, people normally stop their cars.

- What is the meaning of normally?
- Is it possible to build a system on this statement?
- What happens when people do not stop their cars?

No basis for formalization.

Mathematical Notation Avoids Ambiguity

When the temperature is too high, the ventilation has to be switched on or the maintenance staff has to be informed.

Is it ok to switch on ventilation **and** inform the maintenance staff?

$$\text{temperatureHigh} \Rightarrow \text{informStaff } \textbf{or} \text{ ventilationOn}$$

or

$$\text{temperatureHigh} \Rightarrow \text{informStaff } \textbf{xor} \text{ ventilationOn}$$

Mathematical Communication

- Mathematical notation has precise semantics
- New concepts can be defined in terms of old ones
- Mathematical notation is international: no language skills required

Completeness

Every customer is either billed or prepaid.

$$\forall\ c \in \text{Customers}.\ \text{billed}(c)\ \text{xor}\ \text{prepaid}(c)$$

For a purchase through the internet, a person is automatically registered as a new customer.

$$\text{internetPurchase}(c) = \text{Customers} := \text{Customers} \cup \{\ c\ \}$$

Is the new customer billed or prepaid?

Remarks

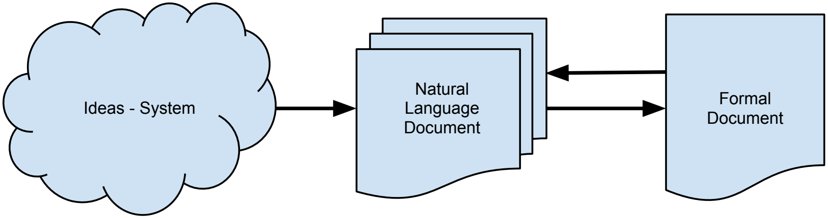
Not Programming

- Programming is constructive
- Programming describes a solution

Not Design

- Description of the entire system
- No description of the software
- No distinction between software and environment
- Incremental approach
- Goal: understanding the system

General Approach



Different Notations

Starting Point: Formal Document

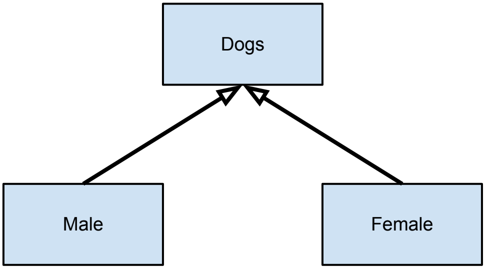
- Transform to natural language
- Transform to graphical notation

Objectives for Choosing a (Graphical) Notation

- Well-defined semantics?
- Is it helpful? Does it make things clearer?

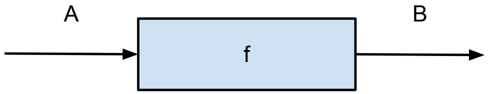
Graphical Notation

Sets and Subset / Classes and Subclasses



Graphical Notation

Function f : $A \rightarrow B$



Example Problem

The system should control the temperature of the room. It can read the current temperature from a thermometer. If the temperature falls below a lower limit, then the heater should be switched on to raise the temperature. If it rises above an upper limit, then the air condition system should be switched on to lower the temperature.

Safety condition

The heater and the air condition should never be switched on at the same time.

Specification

roomTemperature : INTEGER
lowerLimit : INTEGER
upperLimit : INTEGER

Specification (cont)

airCondition : { on, off }
heater : { on, off }

$(\text{airCondition} = \text{on}) \Rightarrow (\text{heater} = \text{off})$
$(\text{heater} = \text{on}) \Rightarrow (\text{airCondition} = \text{off})$

Specification (cont)

Turn on AC

```
startCooling =
PRE
    airCondition = off
    roomTemperature > upperLimit
THEN
    airCondition := on
END
```

Tools

- Pretty printer, Editor
- Syntax checker
- Type checker
- Execizers
- Model checkers
- Interactive provers
- Automatic provers

Languages
The Z Notation

- Developed in the late 1970 at Oxford
- ISO Standard since 2002 (ISO/IEC 13568:2002)
- Support of large user community
- Large number of tools available

Languages
The B Method

- Simplified version of Z
- Goal: Provability
- Introduction of Refinement
- Industrial strength proof tools
- Methodological Approach
- Can also be used for Design and Implementation

Languages

Other Languages

- . . . numerous!
- Most tools come with their own language
- Nearly all based on same underlying concepts
- Main difference: syntax. . .

ProB

- ProB is an animator and model checker for the B-Method
- Fully automatic animation of many B specifications
- Systematically checks a specification for a range of errors
- Supports model finding, deadlock checking and test-case generation
- Further languages supported: Event-B, CSP-M, TLA+, and Z.
- Developed by Michael Leuschel (Uni Düsseldorf) `http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page`

Summary

Requirements Engineering Using Formal Methods

- Advantanges
  - Clear and precise notation
  - Makes you understand you problem
  - Discoveres contradictions
  - Helps you to merge requirements
  - Makes you ask the right questions
- Disadvantages
  - Notation requires some skills to master
  - Not suitable for non-functional requirements