Software Engineering
Prof. Dr. Bertrand Meyer
Dr. Manuel Oriol
Dr. Bernd Schoeller

Lectures 3/4: Requirements

Topics

Part 1: Overview
Part 2: Standards & methods
Part 3: Requirements elicitation
Part 4: Tools
Part 5: Object-oriented requirements, abstract data types
Part 6: Case study
Part 7: Formal requirements
Part 8: Non-functional requirements
Part 9: Conclusion
Complementary material: Bibliography
The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.

*For sources cited, see bibliography*
When not done right

80% of interface fault and 20% of implementation faults due to requirements (Perry & Stieg, 1993)

48% to 67% of safety-related faults in NASA software systems due to misunderstood hardware interface specifications, of which 2/3rds are due to requirements (Lutz, 1993)

85% of defects due to requirements, of which: incorrect assumptions 49%, omitted requirements 29%, inconsistent requirements 13% (Young, 2001).

Numerous software bugs due to poor requirements, e.g. Mars Climate Orbiter

Topics

Part 1: Overview
Part 2: Standards & methods
Part 3: Requirements elicitation
Part 4: Tools
Part 5: Object-oriented requirements, abstract data types
Part 6: Case study
Part 7: Formal requirements
Part 8: Non-functional requirements, conclusion
Part 9: Conclusion
Complementary material: Bibliography
A small case study

Consider a small library database with the following transactions:

3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

Source*: Wing 88

Another case study

The aim of the proposed system is to support project managers and developers in keeping track of the advancement of a project and react to unforeseen circumstances. The system must support distributed projects (i.e. projects whose members are in separate physical locations, connected only by the Internet); all interactive capabilities must be usable through the World-Wide Web.

The system must support the management of software projects. Applicability to other kinds of projects is not required.

The functionalities shall include:

- Defining tasks and subtasks
- Defining dependencies between tasks
- Assigning time estimates to tasks
- Assigning people to tasks (one person may be assigned to multiple tasks, and one task to multiple people)
- Assigning availability levels to people (e.g. number of hours per week)
- Changing any previous assignment
- Reporting completion of a task
- Estimating the completion time of a task, on the basis of timing estimates for subtasks, dependencies between tasks, project members' assignments and availability, completion data.
- Providing output in various forms including individual project member schedules, overall project schedules, PERT, Gantt
- "What-if?" scenarios: assessing the results of hypothetical changes.
- User login with various privileges, including at least "manager" and "project member"

For activities that it does not itself cover, the system shall provide interfaces to other tools, e.g. Bugzilla or another for bug tracking, Sourceforge or another for source code hosting, CVS or another for configuration management.
Part 1: Overview of the requirements task

Definition

“A requirement” is a statement of desired behavior for a system

“The requirements” for a system are the collection of all such individual requirements
Goals of performing requirements

- Understand the problem or problems that the eventual software system, if any, should solve
- Prompt relevant questions about the problem & system
- Provide basis for answering questions about specific properties of the problem & system
- Decide what the system should do
- Decide what the system should not do
- Ascertain that the system will satisfy the needs of its stakeholders
- Provide basis for development of the system
- Provide basis for V & V* of the system

*Validation & Verification, especially testing

Products of requirements

- Requirements document
- Development plan
- V&V plan (especially test plan)
Practical advice

Don’t forget that the requirements also determine the test plan

Possible requirements stakeholders

- Clients (tailor-made system)
- Customers (product for general sale)
- Clients’ and customers’ customers
- Users
- Domain experts
- Market analysts
- Unions?

- Legal experts
- Purchasing agents
- Software developers
- Software project managers
- Software documenters
- Software testers
- Trainers
- Consultants
Your turn!  Who are the stakeholders?

Consider a small library database with the following transactions:

3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

Practical advice

Identify all relevant stakeholders early on
## Requirements categories

<table>
<thead>
<tr>
<th>Functional</th>
<th>Non-functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full system</td>
<td>Software only</td>
</tr>
<tr>
<td>Procedural</td>
<td>Object-oriented</td>
</tr>
<tr>
<td>Informal</td>
<td>Formal</td>
</tr>
<tr>
<td>Textual</td>
<td>Graphical</td>
</tr>
<tr>
<td>Executable</td>
<td>Non-executable</td>
</tr>
</tbody>
</table>

## Components of requirements

- Domain properties
- Functional requirements
- Non-functional requirements (reliability, security, accuracy of results, time and space performance, portability...)
- Requirements on process and evolution
15 quality goals for requirements

- Justified
- Correct
- Complete
- Consistent
- Unambiguous
- Feasible
- Abstract
- Traceable

- Delimited
- Interfaced
- Readable
- Modifiable
- Verifiable
- Prioritized*
- Endorsed

Marked attributes are part of IEEE 830, see below
* "Ranked for importance and/or stability"

Difficulties of requirements

- Natural language and its imprecision
- Formal techniques and their abstraction
- Users and their vagueness
- Customers and their demands
- The rest of the world and its complexity
Bad requirements

The Background Task Manager shall provide status messages at regular intervals not less than 60 seconds.

Better:

The Background Task Manager (BTM) shall display status messages in a designated area of the user interface

1. The messages shall be updated every 60 plus or minus 10 seconds after background task processing begins.
2. The messages shall remain visible continuously.
3. Whenever communication with the background task process is possible, the BTM shall display the percent completed of the background task.

Bad requirements

The XML Editor shall switch between displaying and hiding non-printing characters instantaneously.

Better:

The user shall be able to toggle between displaying and hiding all XML tags in the document being edited with the activation of a specific triggering mechanism. The display shall change in 0.1 second or less.
Bad requirements

The XML parser shall produce a markup error report that allows quick resolution of errors when used by XML novices.

Better:

1. After the XML Parser has completely parsed a file, it shall produce an error report that contains the line number and text of any XML errors found in the parsed file and a description of each error found.
2. If no parsing errors are found, the parser shall not produce an error report.

The two constant pitfalls

- Committing too early to an implementation
  Overspecification!

- Missing parts of the problem
  Underspecification!
A simple problem

Given a text consisting of words separated by BLANKS or by NL (new line) characters, convert it to a line-by-line form in accordance with the following rules:

1. Line breaks must be made only where the given text has BLANK or NL;
2. Each line is filled as far as possible as long as:
3. No line will contain more than MAXPOS characters

See discussion at se.ethz.ch/~meyer/publications/ieee/formalism.pdf

“Improved”

The program’s input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. Characters are classified as:

- Break characters — BL (blank) and NL (new line);
- Nonbreak characters — all others except ET;
- The end-of-text indicator — ET.

A word is a nonempty sequence of nonbreak characters. A break is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possibly leading and trailing breaks, and ending with ET.

The program’s output should be the same sequence of words as in the input, with the exception that an oversize word (i.e. a word containing more than MAXPOS characters, where MAXPOS is a positive integer) should cause an error exit from the program (i.e. a variable, Alarm, should have the value TRUE). Up to the point of an error, the program’s output should have the following properties:

1. A new line should start only between words and at the beginning of the output text, if any.
2. A break in the input is reduced to a single break character in the output.
3. As many words as possible should be placed on each line (i.e., between successive NL characters).
4. No line may contain more than MAXPOS characters (words and BLs).
The program's input is a stream of characters whose end is signaled with a special end-of-text character, ET. There is exactly one ET character in each input stream. Characters are classified as:

- **Break characters** — BL (blank) and NL (new line);
- **Nonbreak characters** — all others except ET;
- The end-of-text indicator — ET.

A **word** is a nonempty sequence of nonbreak characters. A **break** is a sequence of one or more break characters. Thus, the input can be viewed as a sequence of words separated by breaks, with possibly leading and trailing breaks, and ending with ET.

The program's output should be the same sequence of words as in the input, with the exception that an oversize word (i.e., a word containing more than MAXPOS characters, where MAXPOS is a positive integer) should cause an error exit from the program (i.e., a variable, Alarm, should have the value TRUE). Up to the point of an error, the program's output should have the following properties:

1. A new line should start only between words and at the beginning of the output text, if any.
2. A break in the input is reduced to a single break character in the output.
3. As many words as possible should be placed on each line (i.e., between successive NL characters).
4. No line may contain more than MAXPOS characters (words and BLs).

The formal specification

\[
\text{TRIMMED}(b) = (x \in \text{EQUIVCLASS}(b) \land \text{MAXPOS} = \text{length}(x))
\]

\[
\text{FEWEST LINES}(\text{SSC}) = \text{MIN}\_\text{SET}(\text{SSC}, \text{number of new lines})
\]

where the function \text{number of new lines} is defined by:

\[
\text{number of new lines}(i) = \text{card}([1 \leq j \leq \text{length}(x) \land \text{ET}(j - 1) \neq \text{new line}])
\]

\[
\text{and card}(X), \text{defined for any finite set } X, \text{ is the number of elements (cardinality) of } X.
\]

The basic relation. The above definitions allow us to define the basic relation of the problem, relation \text{goal}, precisely. Relation \text{goal}(a, b) holds between input \text{i} and output \text{a}, both of which are sequences of characters, if and only if:

\[
\text{FEWEST LINES}(\text{TRANS}(i)) = \text{MIN}\_\text{SET}(\text{SSC}, \text{number of new lines})
\]

with:

\[
v = \text{limited length} \times \text{short breaks}
\]

The dot operator denotes the composition of relations (one-to-one). A look at

**Contradiction**  **Noise**  **Ambiguity**  **Overspecification**  **Remorse**  **Forward reference**
Given are a non-negative integer $\text{MAXPOS}$ and a character set including two "break characters" blank and new_line.

The program shall accept as input a finite sequence of characters and produce as output a sequence of characters satisfying the following conditions:

- It only differs from the input by having a single break character wherever the input has one or more break characters.
- Any $\text{MAXPOS} + 1$ consecutive characters include a new_line.
- The number of new_line characters is minimal.
- If (and only if) an input sequence contains a group of $\text{MAXPOS} + 1$ consecutive non-break characters, there exists no such output. In this case, the program shall produce the output associated with the initial part of the sequence up to and including the $\text{MAXPOS}$-th character of the first such group, and report the error.

**Practical advice**

Don't underestimate the potential for help from mathematics
15 quality goals for requirements

- Justified
- Correct
- Complete
- Consistent
- Unambiguous
- Feasible
- Abstract
- Traceable
- Delimited
- Interfaced
- Readable
- Modifiable
- Testable
- Prioritized
- Endorsed

Verifiable requirements

Non-verifiable:
- The system shall work satisfactorily
- The interface shall be user-friendly
- The system shall respond in real time

Verifiable:
- The output shall in all cases be produced within 30 seconds of the corresponding input event. It shall be produced within 10 seconds for at least 80% of input events.
- Professional train drivers will reach level 1 of proficiency (defined in requirements) in two days of training.

Adapted from: IEEE
Practical advice

Favor precise, falsifiable language over pleasant generalities

Complete requirements

Complete with respect to what?

Definition from IEEE standard (see next):

An SRS is complete if, and only if, it includes the following elements:

- All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.
- Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.
- Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.
Completeness cannot be “completely” defined.

But (taking advantage of the notion of sufficient completeness for abstract data types) we can cross-check:

- Commands x Queries

...to verify that every effect is defined.

Practical advice

Think negatively
The two parts of requirements

Purpose: to capture the user needs for a “machine” to be built

Jackson’s view: define success as

\[ \text{machine specification} \land \text{domain properties} \Rightarrow \text{requirement} \]

- **Domain properties**: outside constraints (e.g. can only modify account as a result of withdrawal or deposit)
- **Requirement**: desired system behavior (e.g. withdrawal of $n$ francs decreases balance by $n$)
- **Machine specification**: desired properties of the machine (e.g. request for withdrawal will, if accepted, lead to update of the balance)

Domain requirements

Domain assumption: trains & cars travel at certain max speeds

**Requirement**: no collision in railroad crossing
Consider a small library database with the following transactions:

3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

Practical advice

Distinguish machine specification from domain properties
Part 2: Standards and Methods

The purpose of standards

Software engineering standards:

- Define common practice.
- Guide new engineers.
- Make software engineering processes comparable.
- Enable certification.
IEEE 830-1998

“IEEE Recommended Practice for Software Requirements Specifications”

Approved 25 June 1998 (revision of earlier standard)

Descriptions of the content and the qualities of a good software requirements specification (SRS).

Goal: “The SRS should be correct, unambiguous, complete, consistent, ranked for importance and/or stability, verifiable, modifiable, traceable.”

15 quality goals for requirements

- Justified
- Correct
- Complete
- Consistent
- Unambiguous
- Feasible
- Abstract

- Traceable
- Delimited
- Interfaced
- Readable
- Modifiable
- Testable
- Prioritized
- Endorsed
IEEE Standard: definitions

**Contract:**
A legally binding document agreed upon by the customer and supplier. This includes the technical and organizational requirements, cost, and schedule for a product. A contract may also contain informal but useful information such as the commitments or expectations of the parties involved.

**Customer:**
The person, or persons, who pay for the product and usually (but not necessarily) decide the requirements. In the context of this recommended practice the customer and the supplier may be members of the same organization.

**Supplier:**
The person, or persons, who produce a product for a customer. In the context of this recommended practice, the customer and the supplier may be members of the same organization.

**User:**
The person, or persons, who operate or interact directly with the product. The user(s) and the customer(s) are often not the same person(s).

IEEE Standard

Basic issues to be addressed by an SRS:

- Functionality
- External interfaces
- Performance
- Attributes
- Design constraints imposed on an implementation
Recommended document structure:

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, acronyms, and abbreviations ← Glossary!
   1.4 References
   1.5 Overview

2. Overall description
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 Constraints
   2.5 Assumptions and dependencies

3. Specific requirements

Appendixes
Index

Practical advice

Use the recommended IEEE structure
Practical advice

Write a glossary

Recommended document structure

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, acronyms, and abbreviations
   1.4 References
   1.5 Overview

2. Overall description
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 Constraints
   2.5 Assumptions and dependencies

3. Specific requirements
Appendixes
Index
Example section: **scope**

- Identify software product to be produced by name (e.g., Host DBMS, Report Generator, etc.)
- Explain what the product will and will not do
- Describe application of the software: goals and benefits
- Establish relation with higher-level system requirements if any

Example section: **product perspective**

Describe relation with other products if any. Examples:

- System interfaces
- User interfaces
- Hardware interfaces
- Software interfaces
- Communications interfaces
- Memory
- Operations
- Site adaptation requirements
Example section: **constraints**

Describe any properties that will limit the developers’ options

Examples:

- Regulatory policies
- Hardware limitations (e.g., signal timing requirements)
- Interfaces to other applications
- Parallel operation
- Audit functions
- Control functions
- Higher-order language requirements
- Reliability requirements
- Criticality of the application
- Safety and security considerations

---

**Recommended document structure**

1. Introduction
   1.1 Purpose
   1.2 Scope
   1.3 Definitions, acronyms, and abbreviations
   1.4 References
   1.5 Overview

2. Overall description
   2.1 Product perspective
   2.2 Product functions
   2.3 User characteristics
   2.4 Constraints
   2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index
Specific requirements (section 3)

This section brings requirements to a level of detail making them usable by designers and testers.

Examples:

- Details on external interfaces
- Precise specification of each function
- Responses to abnormal situations
- Detailed performance requirements
- Database requirements
- Design constraints
- Specific attributes such as reliability, availability, security, portability

Possible section 3 structure

3. Specific requirements
   3.1 External interfaces
      3.1.1 User interfaces
      3.1.2 Hardware interfaces
      3.1.3 Software interfaces
      3.1.4 Communication interfaces
   3.2 Functional requirements
   3.3 Performance requirements
   3.4 Design constraints
   3.5 Quality requirements
   3.6 Other requirements
Requirements under agile methods

Under XP: requirements are taken into account as defined at the particular time considered
Requirements are largely embedded in test cases

Benefits:
- Test plan will be directly available
- Customer involvement

Risks:
- Change may be difficult (refactoring)
- Structure may not be right
- Test only cover the foreseen cases

Practical advice

Retain the best agile practices, in particular frequent iterations, customer involvement, centrality of code and testing.

Disregard those that contradict proven software engineering principles.
Some recipes for good requirements

Managerial aspects:
- Involve all stakeholders
- Establish procedures for controlled change
- Establish mechanisms for traceability
- Treat requirements document as one of the major assets of the project; focus on clarity, precision, completeness

Technical aspects: how to be precise?
- Formal methods?
- Design by Contract

Checklist
- Premature design?
- Combined requirements?
- Unnecessary requirements?
- Conformance with business goals
- Ambiguity
- Realism
- Testability

After: Kotonya & Sommerville 98
Using natural language for requirements

Keys are:
- Structure
- Precision (including precise definition of all terms)
- Consistency
- Minimizing forward and outward references
- Clarity
- Conciseness

Advice on natural language

Apply the general rules of “good writing” (e.g. Strunk & White, or Zobel)

Use active form
(Counter-example: “the message will be transmitted...”)
This forces you to state who does what

Use prescriptive language (“shall...”)
Separate domain properties and machine requirements
Take advantage of text processing capabilities, within reason
Identify every element of the requirement, down to paragraph or sentence
For delicate or complex issues, use complementary formalisms:
  - Illustrations (with precise semantics)
  - Formal descriptions, with explanations in English
Even for natural language specs, a mathematical detour may be useful
Advice on natural language

- When using numbers, identify the units
- When introducing a list, describe all the elements
- Use illustrations to clarify
- Define all project terms in a glossary
- Consider placing individual requirements in a separate paragraph, individually numbered
- Define generic verbs ("transmitted", "sent", "downloaded", "processed"...) precisely

Part 3:
Requirements elicitation
Case study questions

- Define stakeholders
- Discuss quality of statements -- too specific, not specific enough, properly scoped
- Discuss completeness of information: what is missing?
- Any contradictions that need to be resolved between stakeholders?
- Identify domain and machine requirements
- Identify functional and non-functional requirements
- Plan for future elicitation tasks

The need for an iterative approach

The requirements definition activity cannot be defined by a simple progression through, or relationship between, acquisition, expression, analysis, and specification.

Requirements evolve at an uneven pace and tend to generate further requirements from the definition processes.

The construction of the requirements specification is inevitably an iterative process which is not, in general, self-terminating. Thus, at each iteration it is necessary to consider whether the current version of the requirements specification adequately defines the purchaser’s requirement, and, if not, how it must be changed or expanded further.

Source: Southwell 87
Before elicitation

At a minimum:

- **Overall project description**
- **Draft glossary**

Requirements elicitation: overall scheme

- **Identify stakeholders**
- **Gather wish list of each category**
- **Document and refine wish lists**
- **Integrate, reconcile and verify wish lists**
- **Define priorities**
- **Add any missing elements and nonfunctional requirements**
The four forces at work

Problem to be solved  Business context

Requirements

Domain constraints  Stakeholder constraints

After: Kotonya & Sommerville 98

The customer perspective

“The primary interest of customers is not in a computer system, but rather in some overall positive effects resulting from the introduction of a computer system in their environment”

Source: Dubois 88
### Stereotypes

<table>
<thead>
<tr>
<th>How developers see users</th>
<th>How users see developers</th>
</tr>
</thead>
<tbody>
<tr>
<td>➢ Don’t know what they want</td>
<td>➢ Don’t understand operational needs.</td>
</tr>
<tr>
<td>➢ Can’t articulate what they want</td>
<td>➢ Too much emphasis on technicalities.</td>
</tr>
<tr>
<td>➢ Have too many needs that are politically motivated</td>
<td>➢ Try to tell us how to do our jobs.</td>
</tr>
<tr>
<td>➢ Want everything right now.</td>
<td>➢ Can’t translate clearly stated needs into a successful system.</td>
</tr>
<tr>
<td>➢ Can’t prioritize needs</td>
<td>➢ Say no all the time.</td>
</tr>
<tr>
<td>➢ &quot;Me first&quot;, not company first</td>
<td>➢ Always over budget.</td>
</tr>
<tr>
<td>➢ Refuse to take responsibility for the system</td>
<td>➢ Always late.</td>
</tr>
<tr>
<td>➢ Unable to provide a usable statement of needs</td>
<td>➢ Ask users for time and effort, even to the detriment of their primary duties.</td>
</tr>
<tr>
<td>➢ Not committed to system development projects</td>
<td>➢ Set unrealistic standards for requirements definition.</td>
</tr>
<tr>
<td>➢ Unwilling to compromise</td>
<td>➢ Unable to respond quickly to legitimately changing needs.</td>
</tr>
<tr>
<td>➢ Can’t remain on schedule</td>
<td></td>
</tr>
</tbody>
</table>

### Requirements elicitation: who?

- Users/customers
- Software developers
- Other stakeholders
- Requirements engineers (analysts)
Requirements elicitation: what?

Example questions:
- What will the system do?
- What must happen if...?
- What resources are available for...?
- What kind of documentation is required?
- What is the maximum response time for...?
- What kind of training will be needed?
- What precision is requested for...?
- What are the security/privacy implications of ...?
- Is ... an error?
- What should the consequence be for a ... error?
- What is a criterion for success of a ... operation?

Requirements elicitation: how?

- Contract
- Study of existing non-computer processes
- Study of existing computer systems
- Study of comparable systems elsewhere
- Stakeholder interviews
- Stakeholder workshops
Building stakeholders’ trust

Future users may be jaded by previous attempts where the deliveries did not match the promises

Need to build trust progressively:
- Provide feedback, don’t just listen
- Justify restrictions
- Reinforce trust through evidence, e.g. earlier systems, partial prototypes
- Emphasize the feasible over the ideal

Study of existing systems

Non-computerized processes
- Not necessarily to be replicated by software system
- Understand why things are done the way they are

Existing IT systems
- Commercial products (buy vs build)
- Previous systems
- Systems developed by other companies, including competitors
Stakeholder interviews

Good questions:
- Are egoless
- Seek useful answers
- Make no assumptions

“Context-free” questions:
- “Where do you expect this to be used?”
- “What is it worth to you to solve this problem?”
- “When do you do this?”
- “Whom should I talk to?” “Who doesn’t need to be involved?”
- “How does this work?” “How might it be different?”

Also: meta-questions: “Are my questions relevant?”

Probe further

What else?
Can you show me?
Can you give me an example?
How did that happen?
What happens next?
What’s behind that?
Are there any other reasons?

“How” rather than “why”:
- What was the thinking behind that decision?
Uncovering the implicit

One analyst didn’t include in his requirements document the database that fed his system. I asked him why. He said, “Everyone knows it’s there. It’s obvious.” Words to be wary of! It turned out that the database was scheduled for redesign.

[Winant]

Implicit assumptions are one of the biggest obstacles to a successful requirements process.

Requirements workshops

Often less costly than multiple interviews

Help structure requirements capture and analysis process
Dynamic, interactive, cooperative

Involve users, cut across organizational boundaries

Help identify and prioritize needs, resolve contentious issues; help promote cooperation between stakeholders

Help manage users’ expectations and attitude toward change
Knowing when to stop elicitation

Keep the focus on scope
Keep a list of open issues
Define criteria for completeness

After elicitation

Examine resulting requirements from the viewpoint of requirements quality factors, especially consistency and completeness

Make decisions on contentious issues
Finalize scope of project
Go back to stakeholders and negotiate
15 quality goals for requirements

- Justified
- Correct
- Complete
- Consistent
- Unambiguous
- Feasible
- Abstract
- Traceable
- Delimited
- Interfaced
- Readable
- Modifiable
- Testable
- Prioritized
- Endorsed

Practical advice

Treat requirement elicitation as a mini-project of its own
Part 4:
Requirement Tools

See Exercises!

Part 5:
Object-Oriented Requirements Analysis &
Abstract Data Types
Use Cases (scenarios)

One of the UML diagram types
A use case describes how to achieve a single business goal or task through the interactions between external actors and the system

A good use case must:
- Describe a business task
- Not be implementation-specific
- Provide appropriate level of detail
- Be short enough to implement by one developer in one release

Use case example

Place an order:
- Browse catalog & select items
- Call sales representative
- Supply shipping information
- Supply payment information
- Receive confirmation number from salesperson

May have precondition, postcondition, invariant
Consider a small library database with the following transactions:

3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

Bertrand’s view

Use cases are a minor tool for requirement elicitation but not really a requirement technique. They cannot define the requirements:

- Not abstract enough
- Too specific
- Describe current processes
- Do not support evolution

Use cases are to requirements what tests are to software specification and design

Major application: for testing
Practical advice

Apply use cases for deriving the test plan, not the requirements

Analysis classes

defered class VAT

inherit TANK

feature
  in_valve, out_valve: VALVE
  fill is  -- Fill the vat.
    require
      in_valve.open
      out_valve.closed
    deferred
      in_valve.closed
      out_valve.closed
      is_full
  end

  empty, is_full, is_empty, gauge, maximum,... [Other features] ...

  invariant
    is_full = (gauge $\geq$ 0.97 * maximum) and (gauge $\leq$ 1.03 * maximum)
end
What is object-oriented analysis?

- **Classes** around object types (not just physical objects but also important concepts of the application domain)
- **Abstract Data Types** approach
- **Deferred** classes and features
- Inter-component relations: “client” and inheritance
- Distinction between **reference** and **expanded** clients
- **Inheritance** — single, multiple and repeated for classification.
- **Contracts** to capture the **semantics** of systems: properties other than structural.
- **Libraries** of reusable classes

Why O-O analysis?

Same benefits as O-O programming, in particular extendibility and reusability

Direct modeling of the problem domain

Seamlessness and reversibility with the continuation of the project (design, implementation, maintenance)
What O-O requirements analysis is not

Use cases

(Not appropriate as requirements statement mechanism)

Use cases are to requirements what tests are to specification and design

Television station example

class SCHEDULE feature
    segments: LIST [SEGMENT]
end

Source: OOSC
Schedules

\begin{itemize}
\item note
description : "24-hour TV schedules"
deferred class SCHEDULE feature
\item segments: LIST [SEGMENT]
  -- Successive segments
deferred
end
\item air_time : DATE is
  -- 24-hour period
  -- for this schedule
deferred
end
\item set_air_time (t: DATE)
  -- Assign schedule to
  -- be broadcast at time t.
  require
    t.in_future
  deferred
  ensure
    air_time = t
  end
print
  -- Produce paper version.
deferred
end
end
\end{itemize}

Contracts

Feature precondition: condition imposed on the rest of the world

Feature postcondition: condition guaranteed to the rest of the world

Class invariant: Consistency constraint maintained throughout on all instances of the class
Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:
  ➢ Committing too early to an implementation
    Overspecification!
  ➢ Missing parts of the problem
    Underspecification!
)

Segment

```plaintext
note
description : "Individual fragments of a schedule"
deferred class SEGMENT feature
  schedule : SCHEDULE deferred end
    -- Schedule to which
    -- segment belongs
  index : INTEGER deferred end
    -- Position of segment in
    -- its schedule
  starting_time, ending_time :
    INTEGER is deferred end
    -- Beginning and end of
    -- scheduled air time
next : SEGMENT is deferred end
  -- Segment to be played
  -- next, if any

sponsor : COMPANY deferred end
  -- Segment's principal sponsor
rating : INTEGER deferred end
  -- Segment's rating (for
  -- children's viewing etc.)

Minimum_duration : INTEGER = 30
  -- Minimum length of segments,
  -- in seconds
Maximum_interval : INTEGER = 2
  -- Maximum time between two
  -- successive segments, in seconds
```

... Commands such as change_next,
    set_sponsor, set_rating omitted ...
Segment (continued)

**invariant**

\[
\begin{align*}
\text{in_list:} & \ (1 \leq \text{index}) \ \text{and} \ (\text{index} \leq \text{schedule.segments.count}) \\
\text{in_schedule:} & \ \text{schedule.segments.item(\text{index}) = Current} \\
\text{next_in_list:} & \ (\text{next} \neq \text{Void}) \ \text{implies} \\
& \ (\text{schedule.segments.item(index + 1)} = \text{next}) \\
\text{no_next_iff_last:} & \ (\text{next} = \text{Void}) = (\text{index} = \text{schedule.segments.count}) \\
\text{non_negative_rating:} & \ \text{rating} \geq 0 \\
\text{positive_times:} & \ (\text{starting_time} > 0) \ \text{and} \ (\text{ending_time} > 0) \\
\text{sufficient_duration:} & \ \text{ending_time} - \text{starting_time} \geq \text{Minimum_duration} \\
\text{decent_interval:} & \ (\text{next.starting_time} - \text{ending_time} \leq \text{Maximum_interval})
\end{align*}
\]

end

Commercial

**note**

\[
\text{description: } "\text{Advertizing segment \"}" \text{ deferred class COMMERCIAL inherit \ SEGMENT \ rename sponsor as advertizer end}
\]

**feature**

\[
\begin{align*}
\text{primary:} & \ \text{PROGRAM deferred} \\
& \ (\text{Program to which this} \\
& \ (\text{commercial is attached}) \\
\text{primary_index:} & \ \text{INTEGER deferred} \\
& \ (\text{Index of primary})
\end{align*}
\]

**set_primary** (p: PROGRAM)

\[
\begin{align*}
\text{\text{-- Attach commercial to p.}} \\
\text{require} \\
& \ \text{program_exists:} \ p \neq \text{Void} \\
& \ \text{same_schedule:} \ p,\text{schedule} = \text{schedule} \\
& \ \text{before:} \\
& \ \text{p.starting_time <= starting_time} \\
\text{deferred} \\
\text{ensure} \\
& \ \text{index_updated:} \\
& \ \text{primary_index} = p,\text{index} \\
& \ \text{primary_updated:} \ primary = p \\
\end{align*}
\]

end

**invariant**

\[
\begin{align*}
\text{meaningful_primary_index:} & \ \text{primary_index} = \text{primary.index} \\
\text{primary_before:} & \ \text{primary.starting_time} \leq \text{starting_time} \\
\text{acceptable_sponsor:} & \ \text{advertizer.compatible(\text{primary.sponsor})} \\
\text{acceptable_rating:} & \ \text{rating} \leq \text{primary.rating}
\end{align*}
\]

end
O-O analysis process

Identify abstractions

- New
- Reused

Describe abstractions through interfaces, with contracts

Look for more specific cases: use inheritance

Look for more general cases: use inheritance, simplify

Iterate on suppliers

At all stages keep structure simple and look for applicable contracts
Your turn! Describe this in an O-O way

Consider a small library database with the following transactions:

3. Get the list of books by a particular author or in a particular subject area.
4. Find out the list of books currently checked out by a particular borrower.
5. Find out what borrower last checked out a particular copy of a book.

There are two types of users: staff users and ordinary borrowers.

Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- All copies in the library must be available for checkout or be checked out.
- No copy of the book may be both available and checked out at the same time.
- A borrower may not have more than a predefined number of books checked out at one time.

Practical advice

Take advantage of O-O techniques from the requirements stage on

Use contracts to express semantic properties
Practical advice

Write ADT specifications for delicate parts of the system requirements

Part 6:
Case Study

See Exercise session!
Part 7:
Formal Methods for Requirements

Overview

- What are Formal Methods?
- Advantages and Disadvantages of Formal Methods
- Formal Methods in the Requirement Process
- Mathematical Formulas and Free Text
- Tools for Formal Methods
- The B Method and Language
  - Analysis of a problem in B
  - Implementation and prove of the model in “Click’n’Prove”
- Summary
What are formal methods?

Formal = Mathematical
Methods = Structured Approaches, Strategies

Using mathematics in a structured way to analyze and describe a problem

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
Resistance to mathematics

- “Very abstract”
- “Lots of Greek letters”
- “Difficult to learn and read”
- “Can’t communicate with a normal person”

Useful mathematics

The type of math required consists of

- Set theory
- Functions and Relations
- First-order predicate logic
- Before-After predicates
Set theory

“All humans are male or female.”

\[
\text{Humans} = \text{Male} \cup \text{Female}
\]

“Nobody is male and female at the same time.”

\[
\text{Male} \cap \text{Female} = \emptyset
\]

Functions and relations

“Every customer must have a personal attendant.”

\[
\text{attendant} : \text{Customers} \rightarrow \text{Employees}
\]

“Every customer has a set of accounts.”

\[
\text{AccountsOf} : \text{Customers} \rightarrow \mathcal{P}(\text{Accounts})
\]
First-order predicate logic

"Everybody who works on a Sunday needs to have a special permit."

\[ \forall p \in \text{Employee}: \text{workOnSunday}(p) \Rightarrow \text{hasPermit}(p) \]

"Every customer must at least have one account."

\[ \forall c \in \text{Customers}: \exists a \in \text{Accounts}: a \in \text{AccountsOf}(c) \]

Before-After predicates

"People can enter the building if they have their ID with them. When entering, they have to leave their ID card at the registration desk."

\[
\text{EnterBuilding}(p) = \\
\text{PRE} \\
\text{hasAuthorization}(p) \\
\text{carriesPassport}(p) \\
\text{THEN} \\
\text{peopleInBuilding}' = \text{peopleInBuilding} \cup \{p\} \\
\text{passportsAtDesk}' = \text{passportsAtDesk} \cup \{\text{passportOf}(p)\} \\
\text{not carriesPassport}(p)
\]
Advantages of formal methods

The advantages of using math for any analytical problem

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

Conciseness

Compare

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

with

\[
\text{TicketOwner: IssuedTickets} \rightarrow \text{Person}
\]
Forced precision

"On red traffic lights, people normally stop their cars."

What does "normally" mean? How should we build a system based on this statement? What are the consequences? What happens in the exceptional case?

Formalization Fails

Identified ambiguity

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

May we do both?

\[
\text{temperature\_is\_high} \implies (\text{notify\_staff} \lor \text{ventilation\_on})
\]

or

\[
\text{temperature\_is\_high} \implies (\text{notify\_staff} \oplus \text{ventilation\_on})
\]
Clean form of communication

- Every mathematical notation has a precise semantic definition.
- New constructs can be added defined in terms of old constructs.
- Math does not need language skills and can be easily understood in an international context.

Asking the right questions

"Every customer is either trusted or untrusted."

\[ \forall c \in \text{customer}: \text{trusted}(c) \text{xor untrusted}(c) \]

"Upon internet purchase, a person is automatically registered as a new customer."

\text{InternetPurchase} (by) =
\text{customers'} = \text{customers} \cup \{by\}

Is the new customer trusted or untrusted ?!
This is indeed requirements

It’s not programming:
- Programming describes a solution and not a problem
- Programming is constructive

It’s not design:
- We do not only describe the software
- We describe the full system (software and environment)
- No separation between software and environment
- We do so in an incremental way
- We want to understand the system

General approach
Merging formal requirements

No natural language?
Graphical notations

- Once we have a formal document
  - we can transform it back into a natural language document.
  - we can also transform it into a graphical document.
- There are many graphical notations out there.
- Be careful when choosing a graphical notation:
  - Does it have a well defined semantics?
  - Does it really make things clearer than the formal or natural description?

Graphical notations (cont.)

- Sets as Classes
- Subsets as Subclasses
Graphical notations (cont.)

- Sets as Classes
- Subsets as Subclasses

![Diagram of sets and subsets]

Graphical notations (cont.)

- Functions

![Diagram of function]

instead of \( f : A \rightarrow B \)
An example problem

"The software should control the temperature of the room. It can read the current temperature from a thermometer. Should the temperature fall below a lower limit, then the heater should be switched on to raise the temperature. Should it rise above an upper limit, then the cooling system should be switched on to lower the temperature."

[...]

"Safety concern: the heater and the cooler should never be switched on at the same time."

Formal specification

current_temperature : INTEGER
lower_limit: INTEGER
upper_limit: INTEGER
Formal specification (cont.)

cooling_system : { on, off }
heating_system : { on, off }

(cooling_system = on) ⇒ (heating_system = off)
(heating_system = on) ⇒ (cooling_system = off)

Switch on event

switch_on_cooling_system =
SELECT
  cooling_system = off &
  current_temperature > upper_limit
THEN
  cooling_system := on
END
Formal specification (cont.)

Switch on event

switch_on_heating_system =
SELECT
  heating_system = off &
  current_temperature < lower_limit
THEN
  heating_system := on
END

Languages for formal methods

How should we formalize the requirements?

The Z notation

- Developed in the late 1970s at Oxford
- Support of large user community
- Large number of tools available
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

Other Candidates

- There are numerous languages out there
- Most tools invent an own language
- (Nearly) all are based on the same mathematical concepts
- Biggest difference: The US keyboard does not have Greek letters.

In the end, it is all just math
Formal methods: an assessment

- New approach for Requirements Engineering
- Powerful tools are currently developed

Pros
- Clear and precise notation
- Makes you understand your problem
- Discovers contradictions
- Helps you to merge requirements
- Makes you ask the right questions

Cons
- Notation requires some skills to master
- Not suitable for non-functional requirements

Practical advice

Learn a formal method thoroughly

Let formal methods inform your practice of requirements
Part 9: Conclusion

Key lessons

Requirements are software
- Subject to software engineering tools
- Subject to standards
- Subject to measurement
- Part of quality enforcement

Requirements is both a lifecycle phase and a lifecycle-long activity

Since requirements will change, seamless approach is desirable

Distinguish domain properties from machine properties
- Domain requirements should never refer to machine requirements!
Key lessons

Identify & involve all stakeholders
Requirements determine not just development but tests
Use cases are good for test planning
Requirements should be abstract
Requirements should be traceable
Requirements should be verifiable (otherwise they are wishful thinking)
Object technology helps
  - Modularization
  - Classifications
  - Contracts
  - Seamless transition to rest of lifecycle

Key lessons

Formal methods have an important contribution to make:
  - Culture to be mastered by requirements engineers
  - Necessary for critical parts of application
  - Lead to ask the right questions
  - Proofs & model checking uncover errors
  - Lead to better informal requirements
  - Study abstract data types
  - Nothing to be scared of
Bibliography (1/4)


Bibliography (2/4)


Bibliography (3/4)


Bibliography (4/4)
