Statements about requirements: Brooks

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*
Statements about requirements: Boehm

Relative cost to correct a defect

Source*: Boehm 81

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
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

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>
</tbody>
</table>

Informal vs Formal

Textual

Executable vs Non-executable

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

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

A \textbf{word} is a nonempty sequence of nonbreak characters. A \textbf{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 \textit{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 \textit{MAXPOS} characters, where \textit{MAXPOS} is a positive integer) should cause an error exit from the program (i.e. a variable, \textit{Alarm}, should have the value \textit{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 \textit{NL} characters).
4. No line may contain more than \textit{MAXPOS} characters (words and \textit{BLs}).
The formal specification

where

\[ \text{FORM81} = \{ i \in \text{EQU81} \mid \text{max\_len}(i) \geq \text{MAXPOS} \} \]

\[ \text{EQU81} = \{ i : \text{CHAR} \mid \text{length}(i) = \text{length}(i) \} \]

\[ \text{form\_len}(i) = \max\{ l \mid \text{length}(i) \} \]

\[ (l \in \text{FORM81}) \text{ and } (l \in \text{char81}) \]

A few explanations may help to understand these definitions. If \( i \) is a sequence of characters, then \( \text{form\_len}(i) \) is the maximum length of a line, as expressed in the maximum number of nonbreak characters, none of which is a one char. In other words, it is the maximum number of characters that will fit on one line. However, even if \( (l \in \text{FORM81}) \), it may still be the case that \( \text{length}(i) \) is less than \( \text{form\_len}(i) \). The reason for this is that \( \text{form\_len}(i) \) is the maximum length of a line, and it may be the case that \( \text{length}(i) \) is less than \( \text{form\_len}(i) \). This is because the length of a line is determined by the maximum number of characters that can fit on the line, and this number is not necessarily equal to the maximum number of characters that can fit on a line. For example, if the maximum number of characters that can fit on a line is 10, and the length of a line is only 5, then the line is not full. However, if the length of a line is greater than the maximum number of characters that can fit on a line, then the line is full. In this case, the line is considered to be full, and the maximum number of characters that can fit on a line is the maximum number of characters that can fit on a line that is full.

The basic relations:

- \( \text{FORM81} \) is a set of sequences which are "equal" in that they have the same length.
- \( \text{EQU81} \) is a set of sequences which are "equal" in that they have the same length.
- \( \text{form\_len}(i) \) is the maximum length of a line, as expressed in the maximum number of nonbreak characters, none of which is a one char. In other words, it is the maximum number of characters that will fit on one line. However, even if \( (l \in \text{FORM81}) \), it may still be the case that \( \text{length}(i) \) is less than \( \text{form\_len}(i) \). The reason for this is that \( \text{form\_len}(i) \) is the maximum length of a line, and it may be the case that \( \text{length}(i) \) is less than \( \text{form\_len}(i) \). This is because the length of a line is determined by the maximum number of characters that can fit on the line, and this number is not necessarily equal to the maximum number of characters that can fit on a line. For example, if the maximum number of characters that can fit on a line is 10, and the length of a line is only 5, then the line is not full. However, if the length of a line is greater than the maximum number of characters that can fit on a line, then the line is considered to be full, and the maximum number of characters that can fit on a line is the maximum number of characters that can fit on a line that is full.

"My" spec, informal from formal

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 MAXPOS-th character of the first such group, and report the error.
Practical advice

Do not 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.

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

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
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

Your turn! Separate machine & domain

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

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

IEEE Standard

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
**Practical advice**

Use the recommended IEEE structure

<table>
<thead>
<tr>
<th>Practical advice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write a glossary</td>
</tr>
</tbody>
</table>
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)

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
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

Source: Southwell 87

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.

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.”

Stereotypes

How developers see users
- Don’t know what they want
- Can’t articulate what they want
- Have too many needs that are politically motivated
- Want everything right now.
- Can’t prioritize needs
- “Me first”, not company first
- Refuse to take responsibility for the system
- Unable to provide a usable statement of needs
- Not committed to system development projects
- Unwilling to compromise
- Can’t remain on schedule

How users see developers
- Don’t understand operational needs.
- Too much emphasis on technicalities.
- Try to tell us how to do our jobs.
- Can’t translate clearly stated needs into a successful system.
- Say no all the time.
- Always over budget.
- Always late.
- Ask users for time and effort, even to the detriment of their primary duties.
- Set unrealistic standards for requirements definition.
- Unable to respond quickly to legitimately changing needs.
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

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
**Your turn!**

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.

---

**Devise use cases**

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.

---

**Discussion**

Use cases are a tool for requirement elicitation but insufficient to define 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 ensure

in_valve.closed

out_valve.closed

is_full

end

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

invariant

is_full = (gauge >= 0.97 * maximum) and (gauge <= 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

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
What is 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)

To be continued: we need abstract data types before continuing the discussion of O-O analysis. See lecture 3.

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)


Michael Jackson: Software Requirements and Specifications, Addison-Wesley, 1996.


Bibliography (4/4)
