Part 2: Requirements engineering
Requirements engineering topics

1: Overview
2: Standards & methods
3: Requirements elicitation
4: Object-oriented requirements
5: Formal requirements
6: 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*
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
Requirements engineering topics

1: Overview
2: Standards & methods
3: Requirements elicitation
4: Tools
5: Object-oriented requirements
6: Formal requirements

Complementary material: Bibliography
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

\(^*\textit{Validation & Verification, especially testing}\)
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
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

- Functional
- Non-functional
- Full system
- Software only
- Procedural
- Object-oriented
- Informal
- Formal
- Textual
- Graphical
- Executable
- Non-executable
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
Requirements vs prototyping

“Prototype” may mean:
1. Pilot project
2. Throw-away development
   Brooks: “Plan to throw one away; you will anyhow"
3. Incremental development
4. Preparing for reusable components
5. Experimentation: user interface, implementation…
6. Experimentation: requirements capture

A prototype (1, 5, 6) may help requirements, but is not a substitute for requirements.
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
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 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, 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).

Source: Goodenough & Gerhart
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.

The program's output should be the same 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. 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.
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 between successive NL characters.
4. No line may contain more than MAXPOS characters (words and BLs).

Source: Meyer 85
The formal specification

where

\[ \text{TRIMMED} (b) = \{ s \in \text{EQUIVALENT} (b) \mid \text{max\_line\_length} (s) \leq \text{MAXPOS} \} \]

\[ \text{EQUIVALENT} (b) = \{ s \in \text{seq}[\text{CHAR}] \mid \text{length} (s) = \text{length} (b) \text{ and } (\forall i \leq \text{length} (b), s(i) = b(i) \Rightarrow s(i) \in \text{BREAK\_CHAR} \text{ and } b(i) \in \text{BREAK\_CHAR}) \} \]

\[ \text{max\_line\_length} (s) = \max \{ (j-i) \mid 0 \leq i \leq \text{length} (s) \text{ and } (\forall k \leq i+1, j, s(k) \neq \text{new\_line}) \} \]

A few explanations may help in understanding these definitions. If \( s \) is a sequence of characters, \( \text{max\_line\_length} (s) \) is the maximum length of a line in \( s \), expressed as the maximum number of consecutive characters, none of which is a new line. In other words, it is the maximum value of \( j - i \) such that \( s(k) \) is not a new line for any \( k \) in the interval \( i \leq k \leq j \). (We will have more to say about this definition below.) \( \text{EQUIVALENT} (b) \) is the set of sequences that are “equivalent” to sequence \( b \) in the sense of being identical to \( b \), except that \( \text{new\_line} \) characters may be substituted for blank characters or vice versa. Finally, \( \text{TRIMMED} (b) \) is the set of sequences which are “equivalent” to \( b \) and have a maximum line length less than or equal to MAXPOS.

Fewest lines. Let \( \text{SSC} \) be a set of sequences of characters. These sequences can be interpreted as consisting of lines separated by \( \text{new\_line} \) characters. We define the set \( \text{FEWEST\_LINES} (\text{SSC}) \) as the subset of \( \text{SSC} \) consisting of those sequences that have as few lines as possible:

\[ \text{FEWEST\_LINES} (\text{SSC}) = \{ s \mid \text{number\_of\_new\_lines} (s) = \text{MIN\_SET} (\text{SSC}, \text{number\_of\_new\_lines}) \} \]

where the function \( \text{number\_of\_new\_lines} \) is defined by:

\[ \text{number\_of\_new\_lines} (s) = \text{card} (\{ i \mid 1 \leq i \leq \text{length} (s) \mid s(i) = \text{new\_line} \}) \]

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

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

\[ o \in \text{FEWEST\_LINES} (\text{TRANSF} (i)) \]

\[ \text{TRANSF} (i) = \{ s \mid \text{TRIMMED} (s) \text{ with } \text{number\_of\_new\_lines} (s) \leq \text{limited\_length} \} \]

with

\[ \text{tr} = \text{limited\_length} \cdot \text{short\_breaks} \]

The dot operator denotes the composition of relations (see box). A look at

\[ \text{dom} (\text{goal}) = \{ s \in \text{seq}[\text{CHAR}] \mid (\forall i \leq \text{length} (s) - \text{MAXPOS}, (\forall j \leq i + \text{MAXPOS}, s(j) \in \text{BREAK\_CHAR}) \} \]

The property expressed by this theorem is that the domain of relation \( \text{goal} \) consists of sequences such that, if a character is followed by MAXPOS other characters, at least one character among \( a \) and the other characters must be a break.

An important property of the above-defined \( \text{goal} \) relation is that, with inputs not in the domain of the \( \text{goal} \) relation—like sequences with overline words—clearly, a robust and complete specification should include (along with \( \text{goal} \)) another relation, say, \( \text{exception\_goal} \), whose domain is \( \text{INPUT} - \text{dom} (\text{goal}) \) (set difference). This relation would supplement \( \text{goal} \) by defining alternative results (usually some kind of error message) for erroneous inputs.

Formal specification of erroneous cases falls beyond the scope of this paper, but a discussion of the problem and precise definitions of terms such as “error,” “failure,” and “exception” can be found in a paper by Cristian.4

Discussion. What we have obtained is an abstract specification—this is, a mathematical description of the problem. It would be difficult to criticize this specification as being oriented toward a particular implementation; if

\[ w = \text{tr} \cdot s \]

January 1985
“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 $\text{MAXPOS}$-th character of the first such group, and report the error.
Practical advice

Don’t underestimate the potential for help from mathematics
Your turn!  
Discuss these requirements

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.
15 quality goals for requirements

<table>
<thead>
<tr>
<th>Right</th>
<th>Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Justified</td>
<td>• Traceable</td>
</tr>
<tr>
<td>• Correct</td>
<td>• Delimited</td>
</tr>
<tr>
<td>• Complete</td>
<td>• Interfaced</td>
</tr>
<tr>
<td>• Consistent</td>
<td>• Readable</td>
</tr>
<tr>
<td>• Unambiguous</td>
<td>• Modifiable</td>
</tr>
<tr>
<td>• Feasible</td>
<td>• Testable</td>
</tr>
<tr>
<td>• Abstract</td>
<td>• Prioritized</td>
</tr>
<tr>
<td></td>
<td>• Endorsed</td>
</tr>
</tbody>
</table>
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.
Practical advice

Think negatively
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.
The two parts of requirements

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

Jackson’s view: define success as

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

- **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!

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
**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
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
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
Your turn! Outline some sections

**Consider a small library database with the following transactions:**

1. **Check out a copy of a book.** Return a copy of a book.
2. **Add a copy of a book to the library.** Remove a copy of a book from the library.
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.
Lifecycle models

Origin: Royce, 1970, Waterfall model

Scope: describe the set of processes involved in the production of software systems, and their sequencing

“Model” in two meanings of the term:
- Idealized description of reality
- Ideal to be followed
Using natural language in 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
Natural language elements to be avoided

- Noise phrases such as “obviously”, “clearly”, “certainly”.
- Ambiguous terms such as “some”, “several”, “many”
- List terminators such as “etc.”, “such as”
- Ambiguous pronouns ( “When module A calls module B its message history file is updated”).
- “To Be Defined”
The waterfall model of the lifecycle
V-shaped

- FEASIBILITY STUDY
- REQUIREMENTS ANALYSIS
- GLOBAL DESIGN
- DETAILED DESIGN
- IMPLEMENTATION
- DISTRIBUTION
- SYSTEM VALIDATION
- SUBSYSTEM VALIDATION
- UNIT VALIDATION
Arguments for the waterfall

(After B.W. Boehm: *Software engineering economics*)

- The activities are necessary
  - (But: merging of middle activities)

- The order is the right one.

Source: Boehm 81
The waterfall model

Feasibility study → Requirements → Specification
Global design → Detailed design → Implementation
V & V → Distribution
Problems with the waterfall

- Late appearance of actual code.
- Lack of support for requirements change — and more generally for extendibility and reusability.
- Lack of support for the maintenance activity (70% of software costs?).
- Division of labor hampering Total Quality Management.
- Impedance mismatches.
- Highly synchronous model.
Quality control?

- Requirement Analysts
  - Designers
  - Implementers
  - Testers
  - Customers
Impedance mismatches

1. As Management requested it.
2. As the Project Leader defined it.
3. As Systems designed it.
4. As Programming developed it.
5. As Operations installed it.
6. What the user wanted.

(Pre-1970 cartoon; origin unknown)
The Spiral model

Figure from: Ghezzi, Jazayeri, Mandrioli, *Software Engineering, 2nd edition*, Prentice Hall
Seamless development:

- Single notation, tools, concepts, principles throughout
- Eiffel is as much for analysis & design as implementation & maintenance
- Continuous, incremental development
- Keep model, implementation and documentation consistent

Reversibility: go back and forth

- Saves money: invest in single set of tools
- Boosts quality

Example classes:

- PLANE, ACCOUNT, TRANSACTION...
- STATE, COMMAND...
- HASH_TABLE...
- TEST_DRIVER...
- TABLE...
Seamless development

Use consistent notation from analysis to design, implementation and maintenance.

Advantages:

- Smooth process. Avoids gaps (improves productivity, reliability).
- Direct mapping from problem to solution, i.e. from software system to external model.
- Better responsiveness to customer requests.
- Consistency, ease of communication.
- Better interaction between users, managers and developers.
Single model principle

Use a single base for everything: analysis, design, implementation, documentation...

Use tools to extract the appropriate views.
Consequences of this discussion

- Requirements are generally viewed as a step, but are better understood as an activity, normally carried out at the beginning but possibly needing to be taken up again later.

- Requirements will change.

- The lifecycle model should support requirements and their continuous adaptation.
Practical advice

If you use a lifecycle model, make sure it integrates change and maintenance
Capability Maturity Model Integration (CMMI)

Process improvement approach
Various aspects: systems engineering, software engineering, integrated product & process development, supplier sourcing

Can be measured using:

- **Maturity levels:** staged (for an organization)
- **Capability levels:** continuous (within a process area)

**Maturity levels:**
1. Initial
2. Managed
3. Defined
4. Quantitatively managed
5. Optimizing
1: Initial

- Processes performed but often ad-hoc or chaotic
- Performance dependent on competence & heroics
- Performance difficult to predict.

Source: SEI
2: Managed

Process properties:
- Planned and executed in accordance with policy
- Employs skilled people having adequate resources to produce controlled outputs
- Involves relevant stakeholders
- Monitored, controlled, reviewed, evaluated for adherence to its process description.
- Institutionalized
- Discipline helps ensure that existing practices are retained during times of stress.
- Status visible to management at defined points.
3: Defined

Managed, plus:

- Description tailored from organization’s set of standard processes according to tailoring guidelines
- This contributes work products, measures, and other process-improvement information
- Standard processes improved over time
4: Quantitatively managed

Defined, plus:

- Controlled using statistical & other quantitative techniques
- Statistical predictability
- Projects use measurable objectives to meet needs of customers, end-users & organization
- Managers & engineers use the data in managing processes & results
5: Optimizing

Quantitatively managed, plus:

- Changed to meet current and projected business objectives
- Focus on continually improving range of process performance through incremental, innovative technological improvements.
- Process improvement is inherently part of everybody’s role, resulting in cycles of continual improvement.
"The project takes appropriate steps to ensure that the agreed-upon set of requirements is managed to support the planning and execution needs of the project. When a project receives requirements from an approved requirements provider, the requirements are reviewed with the requirements provider to resolve issues [...]. Commitment to the requirements is [then] obtained from the project participants. [...]"
Specific Practices:
- Obtain an understanding of requirements
- Obtain commitment to requirements
- Manage requirements changes
- Maintain bidirectional traceability of requirements
- Identify inconsistencies between project work and requirements
ISO 9001:2000

Quality rules:
1. Develop quality management system
2. Implement quality management system
3. Improve quality management system
4. Develop, control and maintain quality documents

Management rules:
1. Promote & manage quality quality
2. Satisfy customers (identify requirements...)
3. Establish quality policy
4. Carry out quality planning
5. Control quality system
English or Metric - why Mars Climate Orbiter was lost!

Lord Wodehouse <w0400@ggr.co.uk> Fri, 01 Oct 1999

The following quoted from NASA's press release shows that for the second time a mix-up in units resulted in an experiment failure, but this time it was a spacecraft.

"The peer review preliminary findings indicate that one team used English units (e.g., inches, feet and pounds) while the other used metric units for a key spacecraft operation. This information was critical to the maneuvers required to place the spacecraft in the proper Mars orbit."

On the other hand...
Rational Unified Process

Defines phases of the software process:

- Inception
- Elaboration
- Construction
- Transition

Requirements are part of inception. Includes:

- Business case (business context, success factors)
- Financial forecast
- Use case model
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
Our 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
Literate Programming

First version called “WEB” was developed by Donald E. Knuth for the TeX word processing system.

Key characteristics:

- Regards programming as the transformation of a document into code.
- Natural language text and code are unified into one document. Automatic tools are used to extract the code and generate the program.
- During development, parts can be left unspecified.
- Opens the “Analyse, Design, Implement”-Triathlon
64. A storage *get* method must return zero if either the column or the row indexes are zero. In this way it is possible for a structure to force certain entries to be zero. In a diagonal matrix, for example, the structure *preprocess* method should assign zero to the index whenever \( row \neq col \) (and also return *false*).

```
(Dense storage methods 29) +≡
299  const element_type get(const index &row, const index &col) const
300  {
301   if (¬row \lor ¬col) return element_type(0);
302   return data[col][row];
303  }
```
The anti-process movement

“eXtreme Programming” (XP), “Agile” methods

Crystal (Cockburn), Scrum

Test-driven development
Recommended practices, e.g. Pair Programming
Short iteration cycles

“The revenge of the cubicles”
Agile Methods

Example: Extreme Programming
Combination of various ideas:

- Rejects full requirements analysis
- Frequent releases (cf. Microsoft, daily build)
- Recommends rapid prototyping
  - “Code as fast as possible”
- “User stories” captured on story cards
- “Planning game” to counter mistrust between customer & supplier
- “Pair programming”
- User stories are converted into test cases (“test-driven development”)
- Tight integration of customer into software process.
**Agile Methods**

<table>
<thead>
<tr>
<th>Who</th>
<th>Task</th>
<th>Est.</th>
<th>Done</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jack $50</td>
<td>Jan, Feb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hillary $50</td>
<td>Mar, Apr</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Description:**

New Report: Payments in Fiscal Year

<table>
<thead>
<tr>
<th>Jan</th>
<th>Feb</th>
<th>Mar</th>
<th>Apr</th>
</tr>
</thead>
<tbody>
<tr>
<td>$50</td>
<td>$150</td>
<td>$50</td>
<td></td>
</tr>
</tbody>
</table>

**Considerations:**

- Val. date / tran. date (option)

**Acceptance Test:**
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
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

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
Measures for requirements

Measures of size
- Number of clusters
- Number of classes
- Number of deferred features (function points)
- Average number of: precondition/postcondition clauses per feature, invariant clauses per class

Measures of complexity
- Number of intra-cluster cyclic relationships
- Number of inter-cluster cyclic relationships
- Average number of parents/children per class
Measures for requirements

Measures of quality (a posteriori)

- Number of requirements-originating bugs
- Proportion of requirements-originating bugs
- Average time from bug to detection
- Distribution of bugs per requirements module
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 requirements process

Source: Pfleeger & Atlee 05
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
- Domain constraints
- Stakeholder constraints

After: Kotonya & Sommerville 98
Requirements elicitation: who?

Users/customers

Software developers

Other stakeholders

Requirements engineers (analysts)
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

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.

Source: Scharer 81
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
Contract

The contract is not a substitute for a requirements process.
But (unscientific observation): poorly prepared contracts are one of the principal sources of software project failures

Advice:

- Pay attention to the contract
- Involve the technical people
- The contract should delimit the scope of the project
- Strive for win-win clauses
- Define responsibilities precisely, e.g. training, documentation, operation
- Include conflict resolution structures
- Keep the lawyers at bay
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?”
More question types

Open-ended

What:
- What happens next?
- What factors are involved?

How:
- How do you use the product to__________?
- How do people decide which option to select?

"Could":
- Could you conceive of an example when you’d use the product this way?
- Could you see a way to use the product to solve this problem?

Closed

Specific
- Do you have any problems with the login function of the current system?
  (follow-up with specifics, e.g. “Can you recreate the problem?”)

Multiple-choice
(mostly useful to help establish priorities)
- Which would you prefer, A, B, or C?
- If you had to choose one, would you choose, X, Y, or Z?
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?
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.
Workshops

Development VP Avery Kerr of BeWell Medical Corp. assigned our group to define the requirements for a software system to be used by registration agents and customers. “They’re all over the place with what they want,” lamented Mary, RegTrak’s project manager.

I asked her to explain what she meant. Who was all over the place? What did they want? “Product development wants our global agents to use RegTrak to register and order products for their regions or countries,” she explained. “Distribution wants hospital purchasing agents to be able to place orders and check on them. Marketing wants health care workers in the hospital to have access, too.”

It was clear that the RegTrak team needed to nail down the scope and high-level requirements of the proposed project. [Mary’s] comment confirmed my typical experience: Each of the various stakeholders has his or her own understanding of what the scope should be, based on personal experience and knowledge.

“I suggest we bring together all of the stakeholders to define the scope of the project,” I said. Mary agreed. We formed a workshop planning team and got to work.
Benefit of workshops

A workshop is 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
When properly run, help manage user's expectations and attitude toward change
Workshop roles

Selection of stakeholders
Requirements analysts
Facilitator
Recorder

See “JAD” techniques (Joint Application Design)

A workshop is not just a meeting. It must be carefully prepared and have a set of defined deliverables.
Examples of workshop deliverables

Poster
List
Cluster, affinity group
Grids, matrices
Sequences
Drawings
Mind map

Doneness matrix

<table>
<thead>
<tr>
<th></th>
<th>Actor01</th>
<th>Actor02</th>
<th>Actor03</th>
<th>Actor04</th>
</tr>
</thead>
<tbody>
<tr>
<td>UseCase01</td>
<td>x</td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>UseCase02</td>
<td></td>
<td></td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>UseCase03</td>
<td></td>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UseCase04</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Interview/workshop techniques

Brainstorming
Scenarios, use cases
Storyboards
Prototypes
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:

Object-Oriented Requirements Analysis
deferred 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
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

```plaintext
class SCHEDULE feature
    segments: LIST[SEGMENT]
end
```
Schedules

note

description: "24-hour TV schedules"

defered class SCHEDULE feature

segments: LIST [SEGMENT]
  -- Successive segments
  deferred
  end

air_time: DATE is
  -- 24-hour period
  -- for this schedule
  deferred
  end

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
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
### Obligations & benefits in a contract

**deliver**

<table>
<thead>
<tr>
<th></th>
<th>OBLIGATIONS</th>
<th>BENEFITS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Client</strong></td>
<td>(Satisfy precondition:) Bring package before 4 p.m.; pay fee.</td>
<td>(From postcondition:) Get package delivered by 10 a.m. next day.</td>
</tr>
<tr>
<td><strong>Supplier</strong></td>
<td>(Satisfy postcondition:) Deliver package by 10 a.m. next day.</td>
<td>(From precondition:) Not required to do anything if package delivered after 4 p.m., or fee not paid.</td>
</tr>
</tbody>
</table>
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

**note**

`description:`

"Individual fragments of a schedule"

defered 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` deferred end

-- Beginning and end of
-- scheduled air time

`next`: `SEGMENT` 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.)

... Commands such as `change_next`,
`set_sponsor`, `set_rating` omitted ...

**Minimum_duration**: `INTEGER` = 30

-- Minimum length of segments,
-- in seconds

**Maximum_interval**: `INTEGER` = 2

-- Maximum time between two
-- successive segments, in seconds
Segment (continued)

**Invariant**

\[
\text{in\_list: } (1 <= \text{index}) \text{ and } (\text{index} <= \text{schedule\_segments\_count})
\]

\[
\text{in\_schedule: } \text{schedule\_segments\_item(index)} = \text{Current}
\]

\[
\text{next\_in\_list: } (\text{next} /\!\!/= \text{Void}) \implies
\]

\[
(\text{schedule\_segments\_item(index + 1)} = \text{next})
\]

\[
\text{no\_next\_iff\_last: } (\text{next} = \text{Void}) = (\text{index} = \text{schedule\_segments\_count})
\]

\[
\text{non\_negative\_rating: } \text{rating} >= 0
\]

\[
\text{positive\_times: } (\text{starting\_time} > 0) \text{ and } (\text{ending\_time} > 0)
\]

\[
\text{sufficient\_duration:}
\]

\[
\text{ending\_time - starting\_time} >= \text{Minimum\_duration}
\]

\[
\text{decent\_interval:}
\]

\[
(\text{next.starting\_time}) - \text{ending\_time} <= \text{Maximum\_interval}
\]

end
set_primary (p: PROGRAM)
    -- Attach commercial to p.
    require
        program_exists: p /= Void
        same_schedule: p.schedule = schedule
        before:
            p.starting_time <= starting_time
    deferred
    ensure
        index_updated:
            primary_index = p.index
        primary_updated: primary = p
end

note
    description: "Advertizing segment"

deferred class COMMERCIAL inherit SEGMENT
    rename sponsor as advertizer end

feature
    primary: PROGRAM deferred
        -- Program to which this commercial is attached
    primary_index: INTEGER deferred
        -- Index of primary

invariant
    meaningful_primary_index: primary_index = primary.index
    primary_before: primary.starting_time <= starting_time
    acceptable_sponsor: advertizer.compatible (primary.sponsor)
    acceptable_rating: rating <= primary.rating
end
Diagrams: UML, BON

Text-Graphics Equivalence
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!

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
Part 5:
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
Why don’t we like math?

- “Very abstract.”
- “Lots of Greek letters.”
- “Difficult to learn and read.”
- “Can 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.”

Humans = Male ∪ Female

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

Male ∩ Female = ∅
Functions and relations

“Every customer must have a personal attendant.”

attendant : Customers $\rightarrow$ Employees

“Every customer has a set of accounts.”

AccountsOf: Customers $\rightarrow$ $\mathcal{P}$(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.”

EnterBuilding(p) =

\[
\text{PRE} \\
\begin{align*}
& \text{hasAuthorization}(p) \\
& \text{carriesPassport}(p)
\end{align*}
\]

\[
\text{THEN} \\
\begin{align*}
& \text{peopleInBuilding}' = \text{peopleInBuilding} \cup \{ p \} \\
& \text{passportsAtDesk}' = \text{passportsAtDesk} \cup \{ \text{passportOf}(p) \} \\
& \text{not carriesPassport}(p)
\end{align*}
\]
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?

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

or

\[ \text{temperature\_is\_high} \implies (\text{notify\_staff} \text{ xor } \text{ventilation\_on}) \]
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 has is either trusted or untrusted.”

∀ c ∈ customer: trusted(c) xor untrusted(c)

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

InternetPurchase (by) = customers’ = customers ∪ \{by\}

Is the new customer trusted or untrusted ?!
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

- Ideas
- Natural Language Document
- Formal Document

Arrows indicate the flow from Ideas to Natural Language Document to Formal Document, with bidirectional arrows indicating a possible iterative process.
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
Graphical notations (cont.)

- Functions

\[ f : A \rightarrow B \]

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
Tools for formal methods

Categories
- Beautifiers, Editors
- Syntax Checkers
- Type Checks
- Exercisers
- Model Checkers
- Interactive Provers
- Automatic Provers

Complexity
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
Languages for formal methods (cont.)

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
Pro B

Pro B is an exerciser (animator) and (limited) model-checker for the B language

- Accepts B (without refinement)
- Developed by Michael Leusche, Southampton
- http://www.ecs.soton.ac.uk/~mal/systems/prob.html
Alloy is a full model-checker for model based on a relational logic

- Own input language modeled close to object-oriented languages
- Developed by Daniel Jackson, MIT
Atelier B and Click’n’Prove

Prover for the B Method

- Supports the B Method, including refinement, analysis, design and code generation
- Interactive Prover
- Developed by Jean-Raymond Abrial and Dominique Cansell, LORIA, France
- New version currently developed at the ETH as part of the EU Rodin project
- [http://www.loria.fr/~cansell/cnp.html](http://www.loria.fr/~cansell/cnp.html)
Formal methods: an assessment

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

Pros
- Clear and precise notation
- Makes you understand you 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 6:
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


