



---

*Software Engineering  
for  
Outsourcing and Offshoring*

Bertrand Meyer  
Peter Kolb

ETH course, Winter 2006-2007

Part 2: Requirements engineering



- 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.*

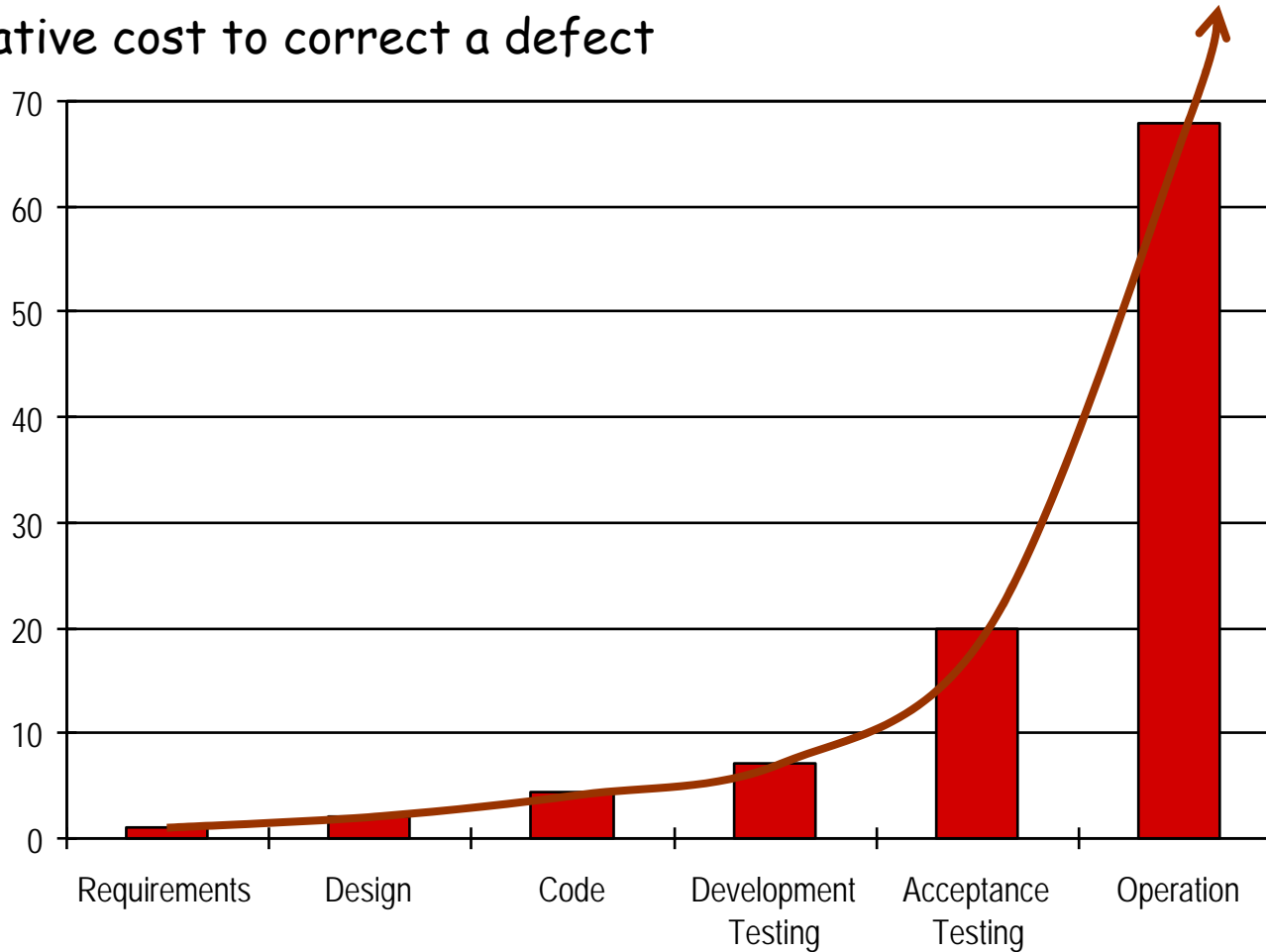
**\*For sources cited, see bibliography**

# Statements about requirements: Boehm



*Source\*: Boehm 81*

Relative cost to correct a defect



# 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



- 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**



"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



*Source: OOSC*

- 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*



- 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:

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.



*Practical advice*

Identify all relevant stakeholders  
early on

# Requirements categories

---



Functional

*Non-functional*

Full system

*Software only*

Procedural

*Object-oriented*

Informal

VS

*Formal*

Textual

*Graphical*

Executable

*Non-executable*



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



*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.



- 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"



- 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



- Committing too early to an implementation

Overspecification!

- Missing parts of the problem

Underspecification!

# A simple problem



*Source: Naur*

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](http://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*).



The program's input is a [redacted] whose end is signaled with [redacted]

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 [redacted] 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. [redacted]).

[redacted], the program's output should have the following properties:

1. A new [redacted] should start only between words and at the beginning of the output text, [redacted].
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 [redacted] ([redacted]).
4. No line may contain more than *MAXPOS* characters (words and *BLS*).

Contradiction Noise Ambiguity  
Overspecification Remorse Forward reference



where

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

$EQUIVALENT(b) = \{s \in seq[CHAR] \mid \begin{aligned} &length(s) = length(b) \text{ and} \\ &(\forall i \in 1..length(b), \\ &s(i) \neq b(i) \Rightarrow \\ &s(i) \in BREAK\_CHAR \text{ and} \\ &b(i) \in BREAK\_CHAR) \end{aligned}\}$

$\max\_line\_length(s) = \max(\{j-i \mid \begin{aligned} &0 \leq i \leq j \leq length(s) \text{ and} \\ &(\forall k \in i+1..j, \\ &s(k) \neq new\_line) \end{aligned}\})$

A few explanations may help in understanding these definitions. If  $s$  is a sequence of characters,  $\max\_line\_length(s)$  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  $i+1..j$ . (We will have more to say about this definition below.)  $EQUIVALENT(b)$  is the set of sequences that are "equivalent" to sequence  $b$  in the sense of being identical to  $b$ , except that *new\_line* characters may be substituted for *blank* characters or vice versa. Finally,  $TRIMMED(b)$  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  $SSC$  be a set of sequences of characters. These se-

quences can be interpreted as consisting of lines separated by *new\_line* characters. We define the set  $FEWEST\_LINES(SSC)$  as the subset of  $SSC$  consisting of those sequences that have as few lines as possible:

$FEWEST\_LINES(SSC) = MIN\_SET(SSC, \text{number\_of\_new\_lines})$

where the function *number\_of\_new\_lines* is defined by:

$\text{number\_of\_new\_lines}(s) = \text{card}(\{i \in 1..length(s) \mid s(i) = new\_line\})$

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 relation of the problem, relation *goal*, precisely. Relation *goal* ( $i, o$ ) holds between input  $i$  and output  $o$ , both of which are sequences of characters, if and only if

$o \in FEWEST\_LINES(TRANSF(i))$   
 $TRANSF(i)$  is the set of sequences related to  $i$  by the composition of the two relations *short\_breaks* and *limited\_length*:

$TRANSF(i) = \{s \in seq[CHAR] \mid tr(i, s)\}$

with

$tr = limited\_length \bullet short\_breaks$

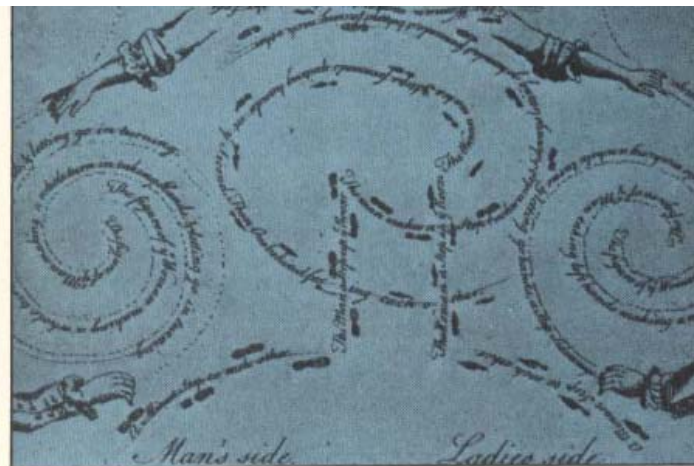
The dot operator denotes the composition of relations (see box). A look at

$\text{dom}(goal) = \{s \in seq[CHAR] \mid \begin{aligned} &\forall i \in 1..length(s) - MAXPOS, \\ &\exists j \in i..i+MAXPOS, \\ &s(j) \in BREAK\_CHAR \end{aligned}\}$

The property expressed by this theorem is that the domain of relation *goal* consists of sequences such that, if a character  $c$  is followed by  $MAXPOS$  other characters, at least one character among  $c$  and the other characters must be a break.

An important problem, not addressed here, is how the specification deals with erroneous cases—that is, with inputs not in the domain of the *goal* relation—like sequences with oversized words. Clearly, a robust and complete specification should include (along with *goal*) another relation, say, *exceptional\_goal*, whose domain is  $INPUT - \text{dom}(goal)$  (set difference); this relation would complement *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 article, 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.<sup>4</sup>

**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

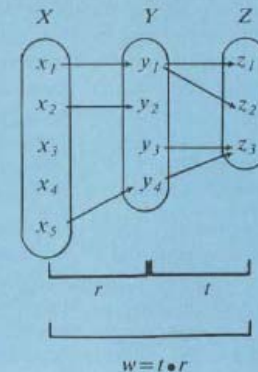


### Composition of relations

Let  $r$  and  $t$  be two relations;  $r$  is from  $X$  to  $Y$  and  $t$  is from  $Y$  to  $Z$  (see figure).

The composition of these two relations, written  $t \bullet r$  (note the order), is the relation  $w$  between sets  $X$  and  $Z$  such that  $w(x, z)$  holds if and only if there is (at least) one element  $y$  in  $Y$  such that both  $r(x, y)$  and  $t(y, z)$  hold.

Thus, in the example illustrated,  $w$  holds for the pairs  $\langle x_1, z_1 \rangle$ ,  $\langle x_1, z_2 \rangle$ , and  $\langle x_5, z_3 \rangle$  (and for these pairs only).



# “My” spec, informal from formal



Given are a non-negative integer *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 *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 *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*

Don't underestimate the potential for help from mathematics



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.



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

- Traceable
- Delimited
- Interfaced
- Readable
- Modifiable
- Testable
- Prioritized
- Endorsed



## 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:

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.

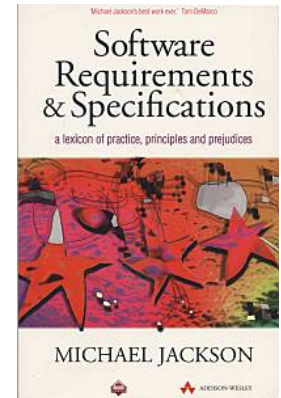
Transactions 1, 2, 4, and 5 are restricted to staff users, except that ordinary borrowers can perform transaction 4 to find out the list of books currently borrowed by themselves. The database must also satisfy the following constraints:

- 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.

# The two parts of requirements



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



Jackson's view: define success as

*machine specification*  $\wedge$  *domain properties*  $\Rightarrow$  *requirement*

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

# Domain requirements



**Domain assumption: trains & cars travel at certain max speeds**

**Requirement: no collision in railroad crossing**



Consider a small library database with the following transactions:

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.



*Practical advice*

Distinguish machine specification  
from domain properties



---

# Part 2:

# Standards and Methods



## Software engineering standards:

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



## "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."



- 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).



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



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



## **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**



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

...



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.



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



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



- 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

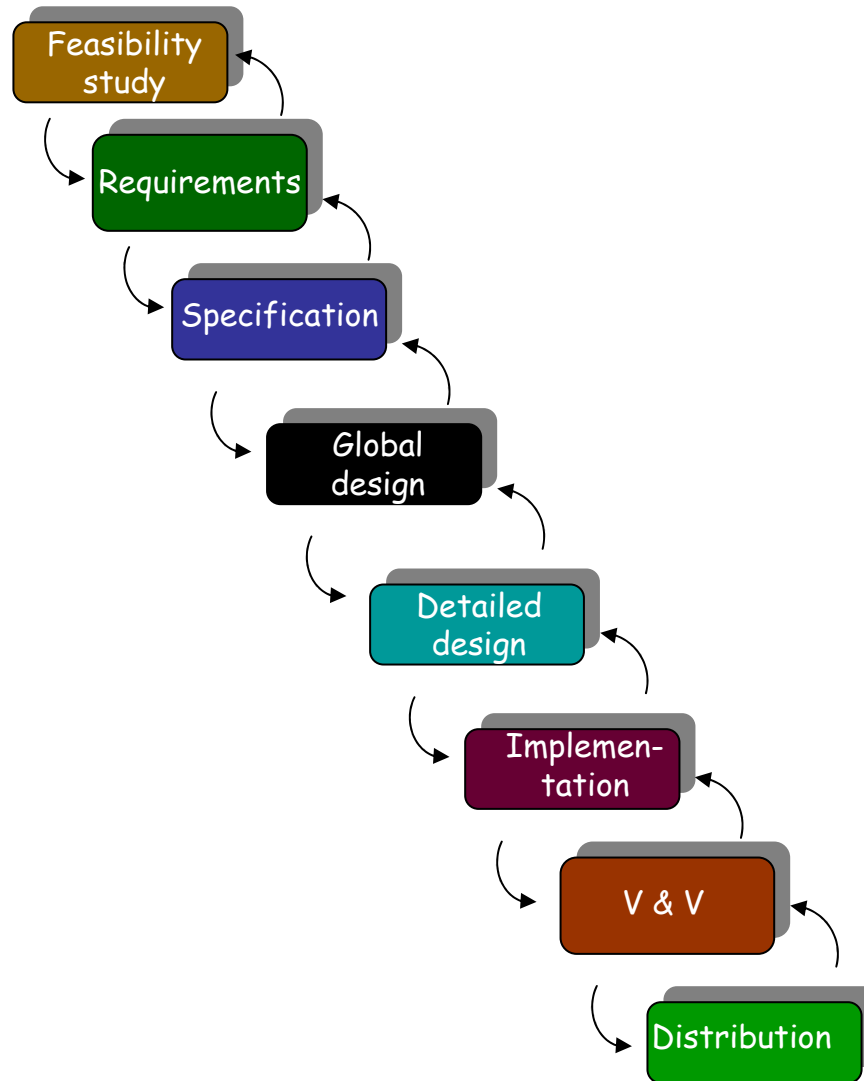
# Natural language elements to be avoided



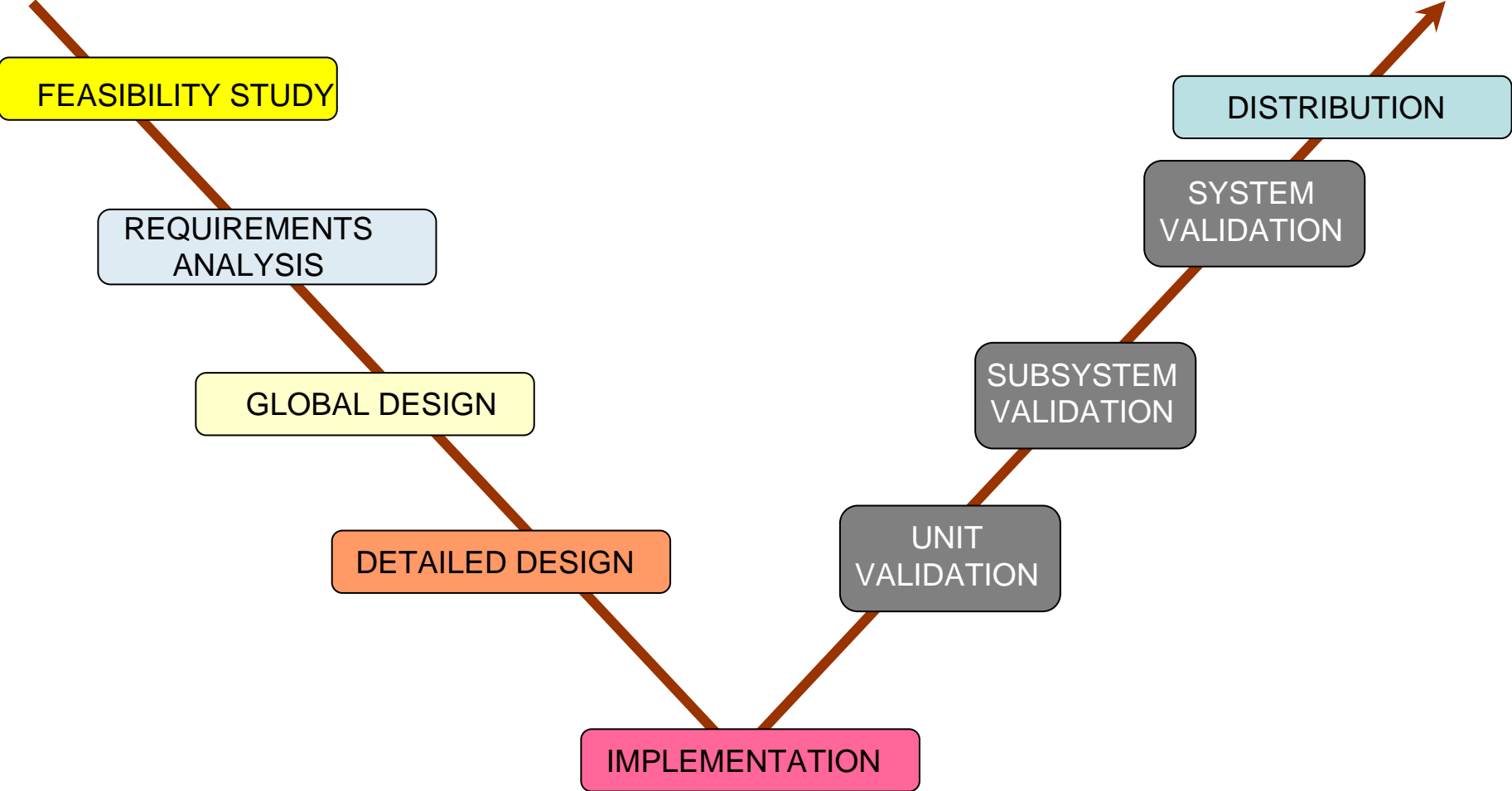
*After Mannion & Keepence 95*

- 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



# Arguments for the waterfall

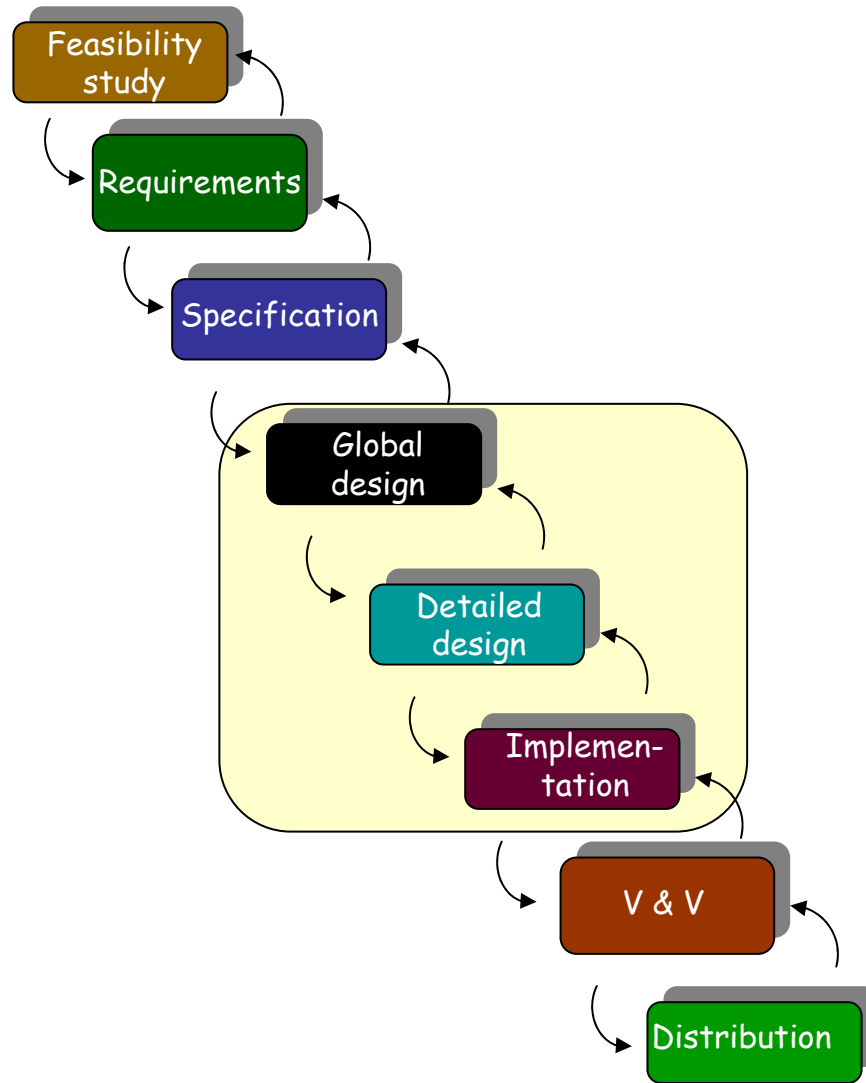


*Source: Boehm 81*

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

- The activities are necessary
  - (But: merging of middle activities)
  
- The order is the right one.

# The waterfall model



# Problems with the waterfall

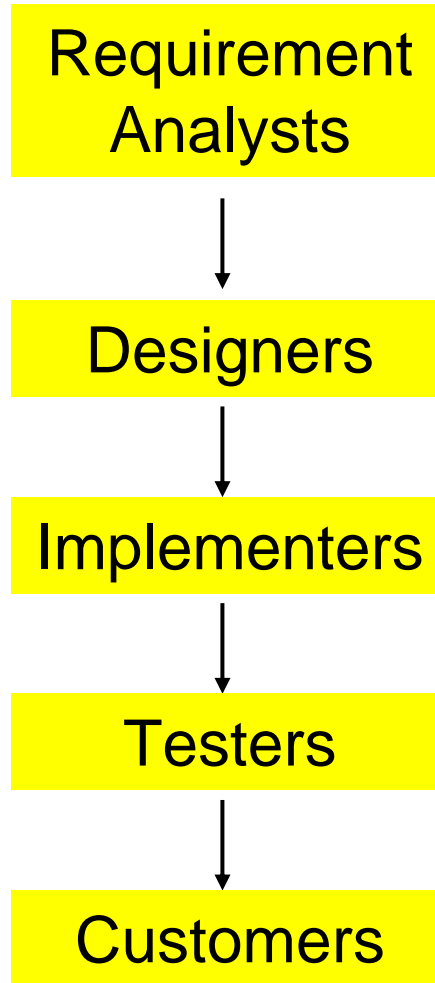
---



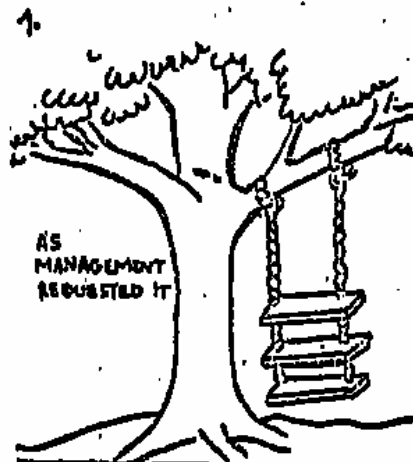
- Late appearance of actual code.
- Lack of support for requirements change — and more generally for extendibility and reusability.
- Lack of support for the maintenance activity (70% of software costs?).
- Division of labor hampering Total Quality Management.
- Impedance mismatches.
- Highly synchronous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

# Quality control?

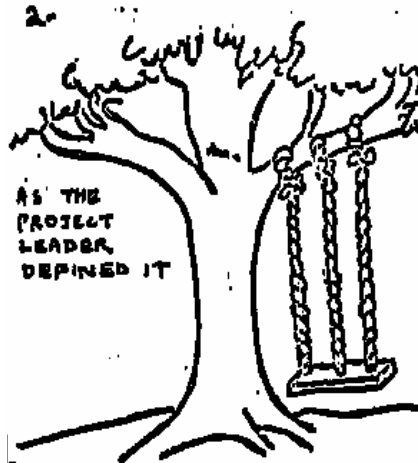
---



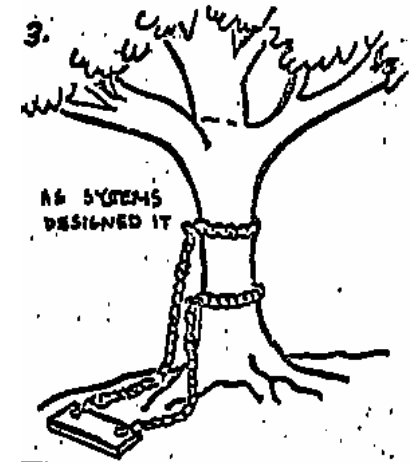
# Impedance mismatches



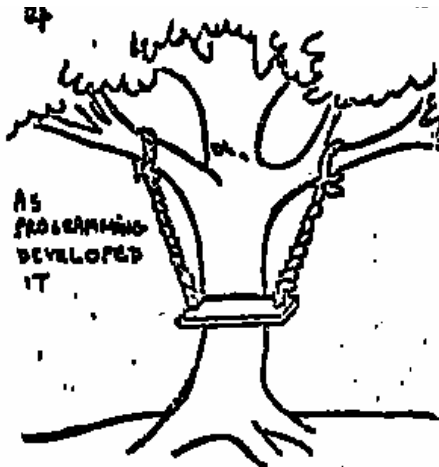
*As Management requested it.*



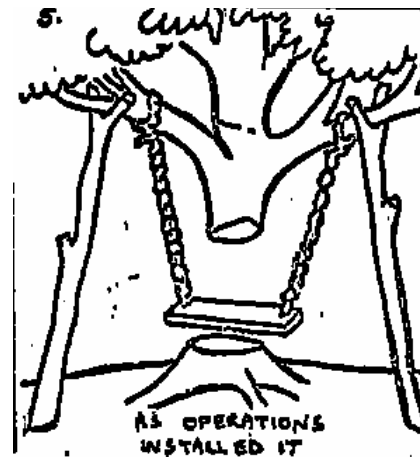
*As the Project Leader defined it.*



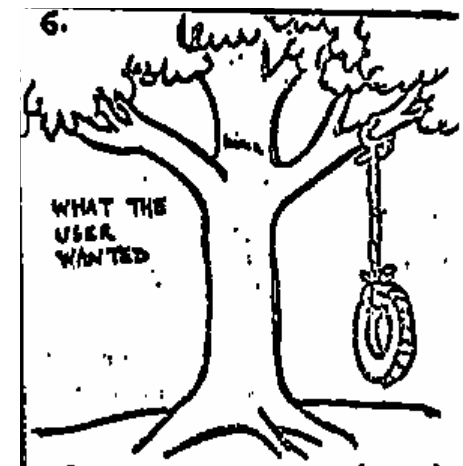
*As Systems designed it.*



*As Programming developed it.*



*As Operations installed it.*



*What the user wanted.*

(Pre-1970 cartoon; origin unknown)

# The Spiral model

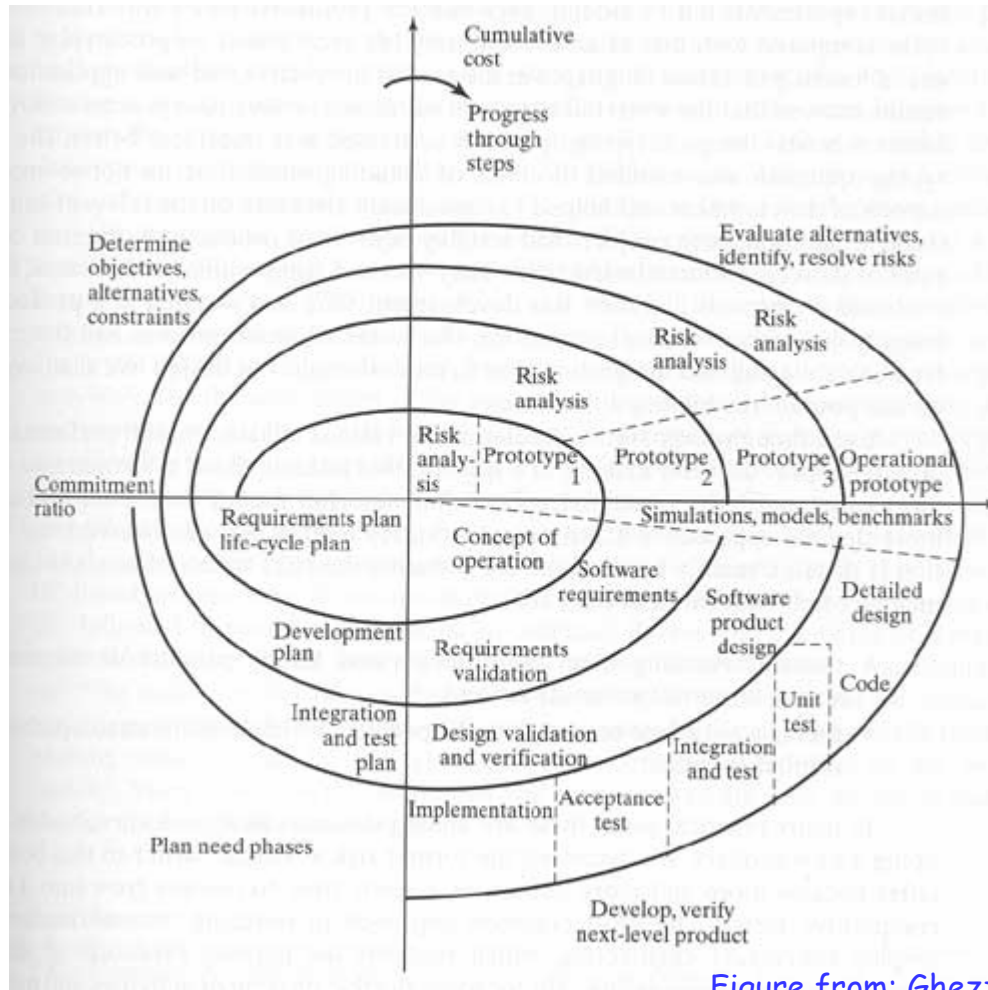


Figure from: Ghezzi, Jazayeri, Mandrioli, *Software Engineering*, 2<sup>nd</sup> edition, Prentice Hall

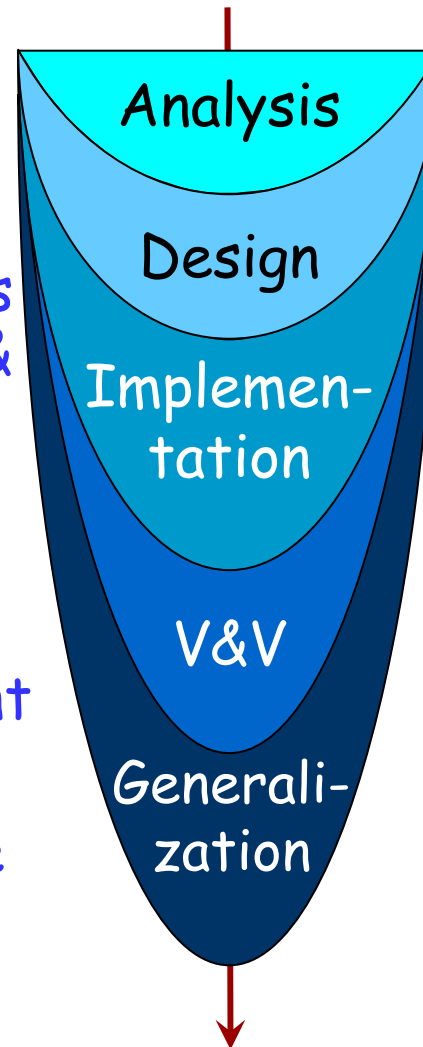


## Seamless development:

- 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...*



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



*Source: SEI*

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

## 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.



## 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



## 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.



## Process area at level 2

*" 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. [...]"*

*Part of the management of requirements is to document requirements changes and rationale and maintain bidirectional traceability between source requirements and all product and product-component requirements."*



## 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



## 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. "*



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



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

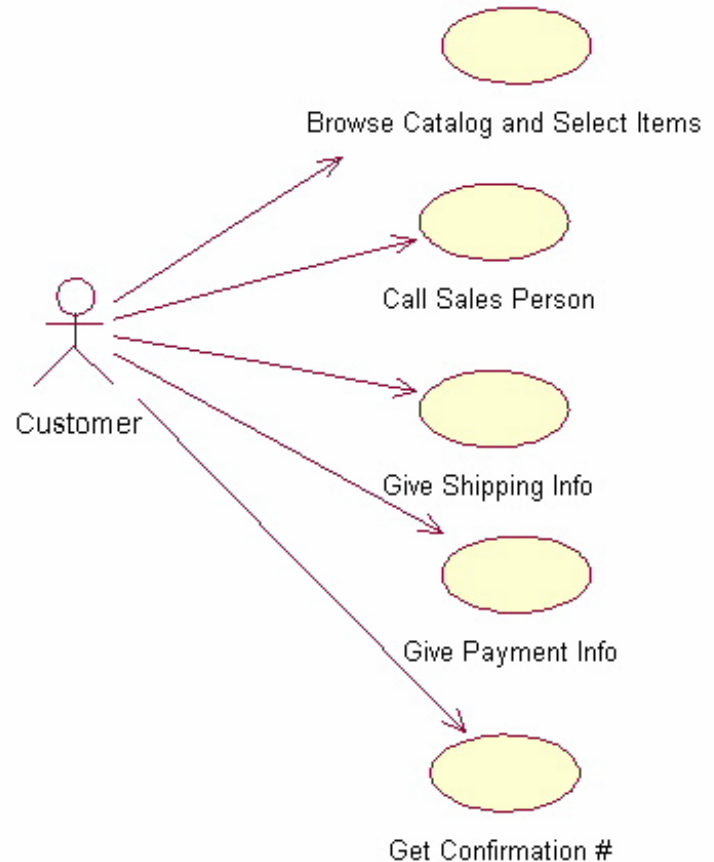
# Use case example



## *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





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



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 ( $\neg row \vee \neg col$ ) 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"



## Example: Extreme Programming

Combination of various ideas:

- Rejects full requirements analysis
- Frequent releases (cf. Microsoft, daily build)
- Recommends rapid prototyping
  - "Code as fast a 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



StoryTag: Payments in Fiscal Year Release: 8/11/01 Priority: 2

Author: Ion on: 7/20/01 Accepted: \_\_\_\_\_

Description: New Report: Payments in Fiscal Year

	Jan	Feb	Mar	Apr
Jack	\$50 12		\$150 12,13	
Hillary			\$50 7	

Val. date / trans. date  
(option)

Considerations:

Estimate: 5

Who	Task	Est.	Done
	Acceptance Test:		



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:

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.



*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.



## 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

---



*After: Kotonya &  
Sommerville 98*

Premature design?

Combined requirements?

Unnecessary requirements?

Conformance with business goals

Ambiguity

Realism

Testability



## 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 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

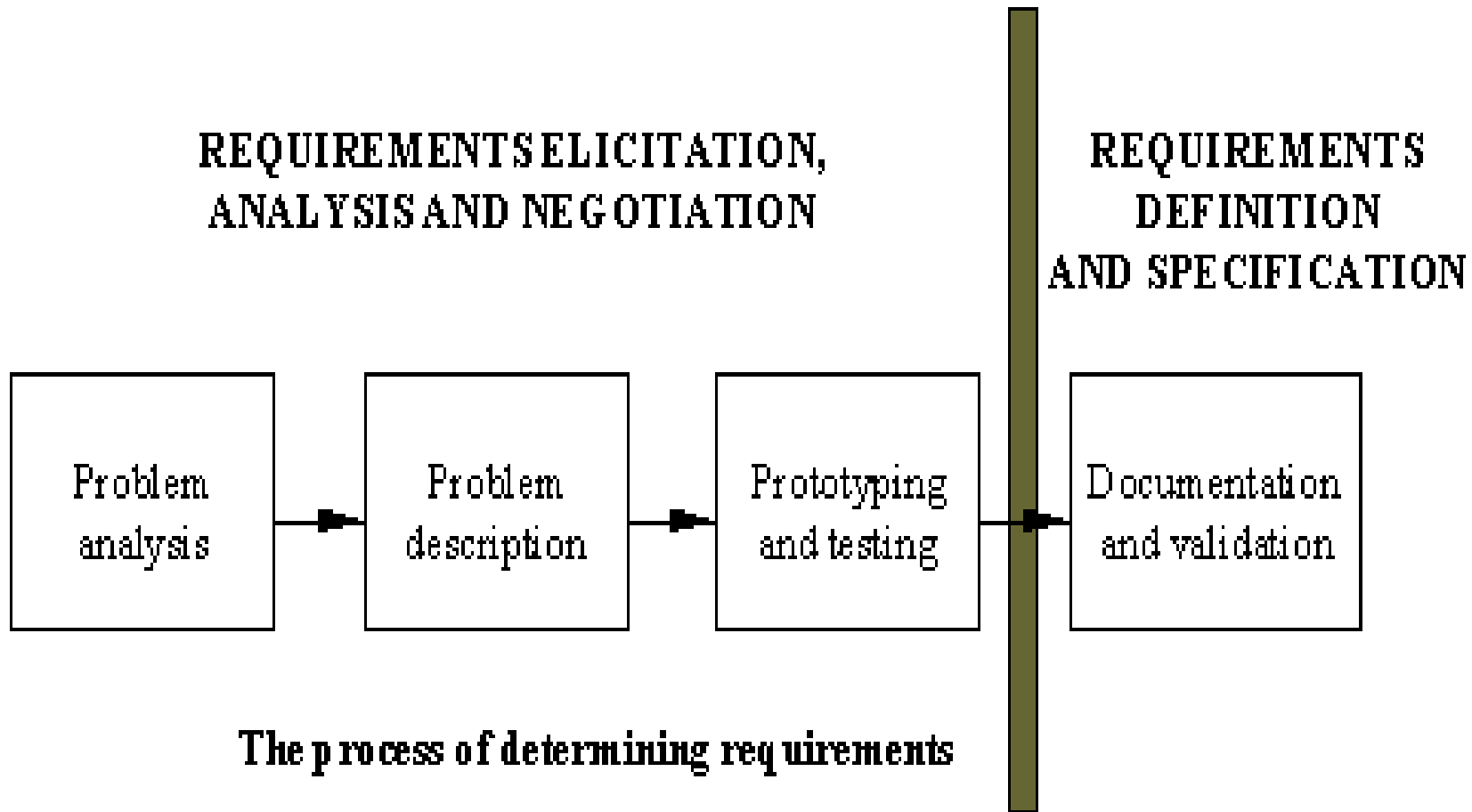


- Define stakeholders
- Discuss quality of statements -- too specific, not specific enough, properly scoped
- Discuss completeness of information: what is missing?
- Any contradictions that need to be resolved between stakeholders?
- Identify domain and machine requirements
- Identify functional and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

# The requirements process



*Source: Pfleeger & Atlee 05*



# The need for an iterative approach



**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.*



At a minimum:

- Overall project description
- Draft glossary



Identify stakeholders

Gather wish list of each category

Document and refine wish lists

Integrate, reconcile and verify wish lists

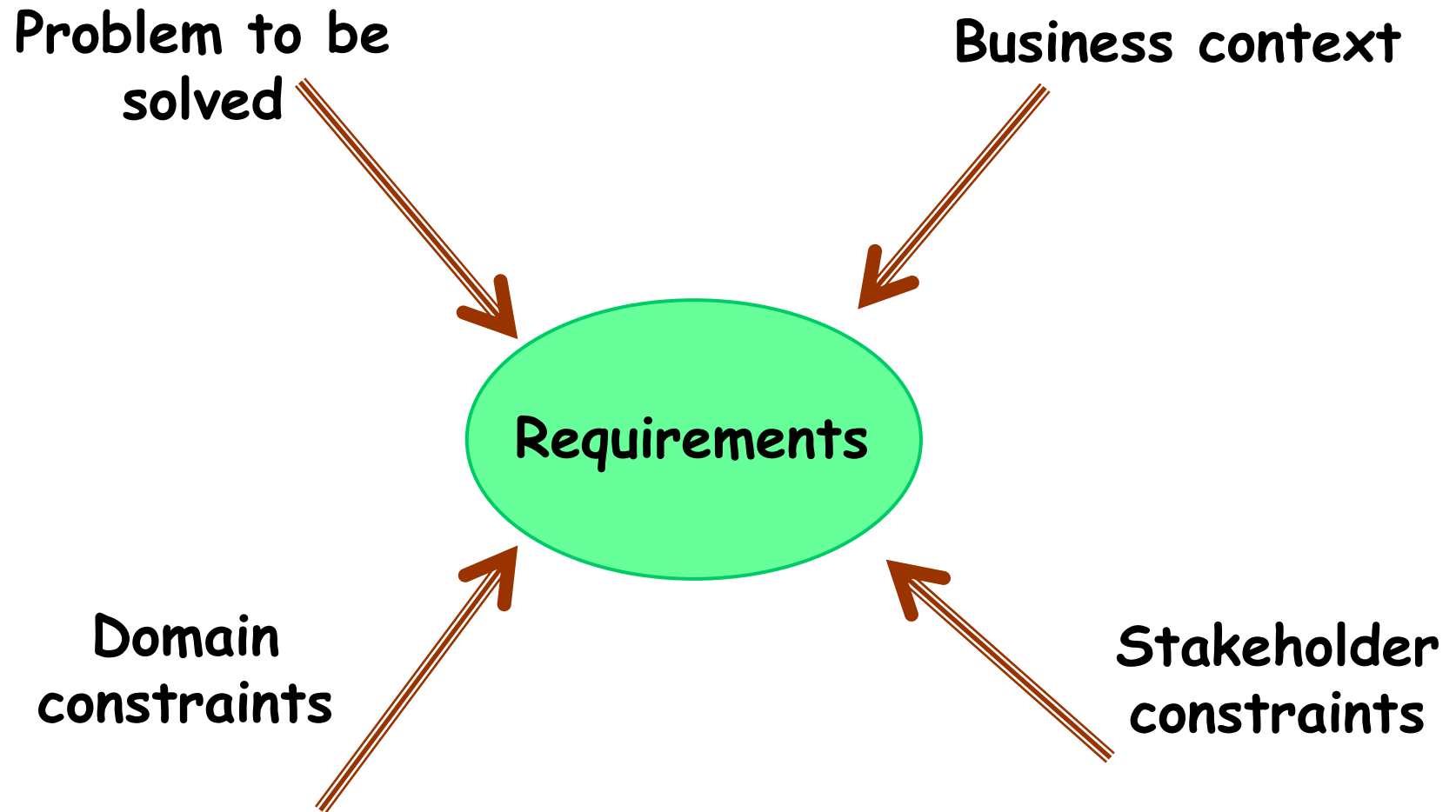
Define priorities

Add any missing elements and nonfunctional requirements

# The four forces at work



*After: Kotonya & Sommerville 98*



# Requirements elicitation: who?

---



Users/customers

Software developers

Other stakeholders

Requirements engineers (analysts)

# The customer perspective

---



***Source: Dubois 88***

*"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"*



## 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.



## 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



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



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



## 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



## 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?"



## 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



*After: Derby 04*

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.



**From: Gottesdiener 02**

*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



*After: Young 01*

- 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



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



*After: Gottesdiener 02*

Poster

List

Cluster, affinity group

Grids, matrices

Sequences

Drawings

Mind map

## Doneness matrix

	Actor01	Actor02	Actor03	Actor04
UseCase01	x		x	
UseCase02				x
UseCase03			x	
UseCase04	x			

# 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



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



- 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**

# Analysis classes



```
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

---



*Source: OOSC*

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



## note

*description:*

*"24-hour TV schedules"*

**deferred 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**



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



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



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"*

**deferred class SEGMENT feature**

*schedule : SCHEDULE deferred end*

-- Schedule to which  
-- segment belongs

*index: INTEGER deferred end*

-- Position of segment in  
-- its schedule

*starting\_time, ending\_time :*

*INTEGER is deferred end*

-- Beginning and end of  
-- scheduled air time

*next: SEGMENT is deferred end*

-- Segment to be played  
-- next, if any

*sponsor: COMPANY deferred end*

-- Segment's principal sponsor

*rating: INTEGER deferred end*

-- Segment's rating (for  
-- children's viewing etc.)

*... 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

*in\_list: (1 <= index) and (index <= schedule.segments.count)*

*in\_schedule: schedule.segments.item(index) = Current*

*next\_in\_list: (next != Void) implies*

*(schedule.segments.item(index + 1) = next)*

*no\_next\_iff\_last: (next = Void) = (index = schedule.segments.count)*

*non\_negative\_rating: rating >= 0*

*positive\_times: (starting\_time > 0) and (ending\_time > 0)*

*sufficient\_duration:*

*ending\_time - starting\_time >= Minimum\_duration*

*decent\_interval:*

*(next.starting\_time) - ending\_time <= Maximum\_interval*

**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
```

```
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
```

## invariant

```
  meaningful_primary_index: primary_index = primary.index
```

```
  primary_before: primary.starting_time <= starting_time
```

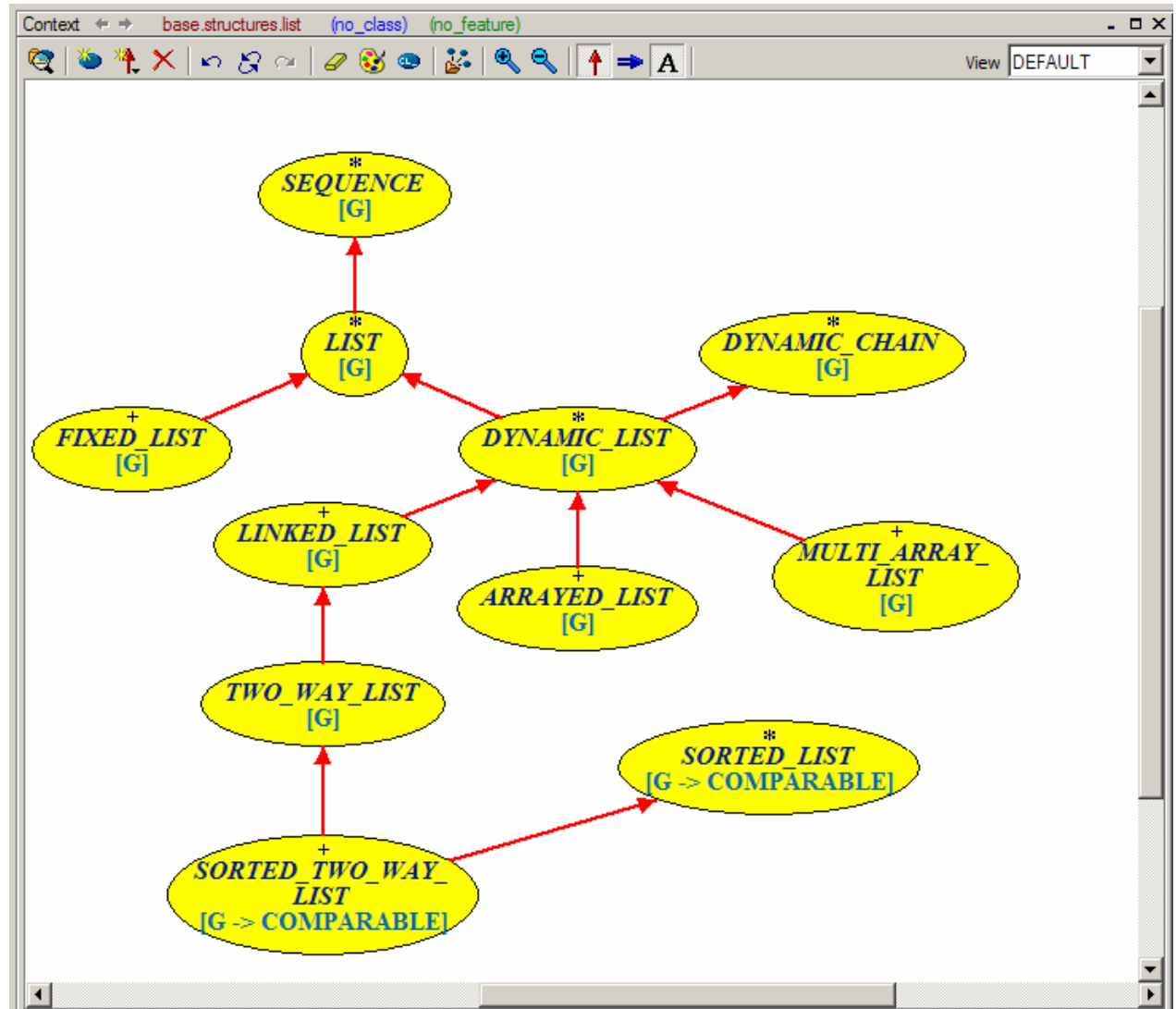
```
  acceptable_sponsor: advertizer.compatible (primary.sponsor)
```

```
  acceptable_rating: rating <= primary.rating
```

```
end
```



## Text-Graphics Equivalence





## 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



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.



*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**



- 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.



- 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."



The type of math required consists of

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

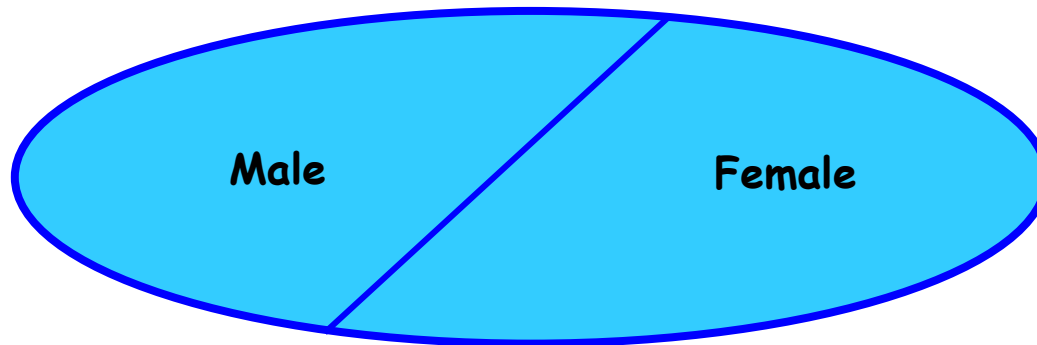


“All humans are male or female.”

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

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

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





“Every customer must have a personal attendant.”

attendant : Customers  $\rightarrow$  Employees

“Every customer has a set of accounts.”

AccountsOf: Customers  $\rightarrow$   $\mathbb{P}$ (Accounts)



“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.”

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



“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) =

*PRE*

*hasAuthorization(p)*

*carriesPassport(p)*

*THEN*

*peopleInBuilding' = peopleInBuilding  $\cup$  { p }*

*passportsAtDesk' = passportsAtDesk  $\cup$  {passportOf(p)}*

*not carriesPassport(p)*



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



## 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

TicketOwner: IssuedTickets → Person



“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**



“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?

$\text{temperature\_is\_high} \Rightarrow (\text{notify\_staff} \text{ or } \text{ventilation\_on})$

or

$\text{temperature\_is\_high} \Rightarrow (\text{notify\_staff} \text{ xor } \text{ventilation\_on})$



- 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.”

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

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

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

Is the new customer trusted or untrusted ?!

# This is indeed requirements

---



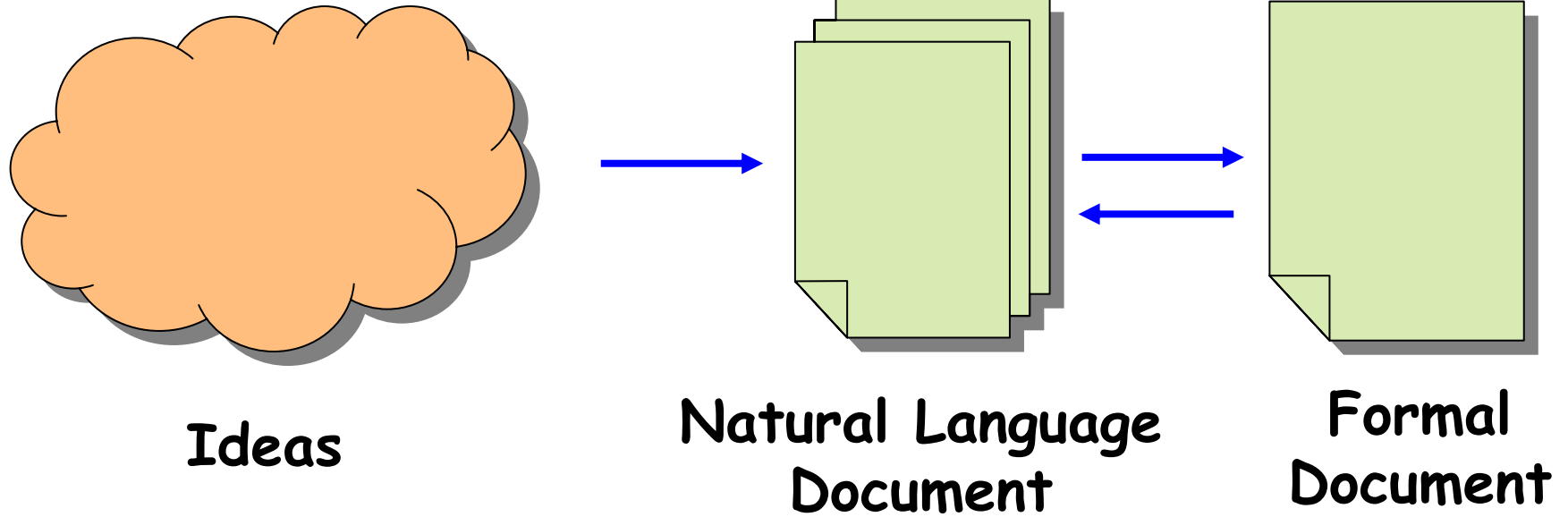
It's not programming:

- Programming describes a **solution** and not a problem
- Programming is **constructive**

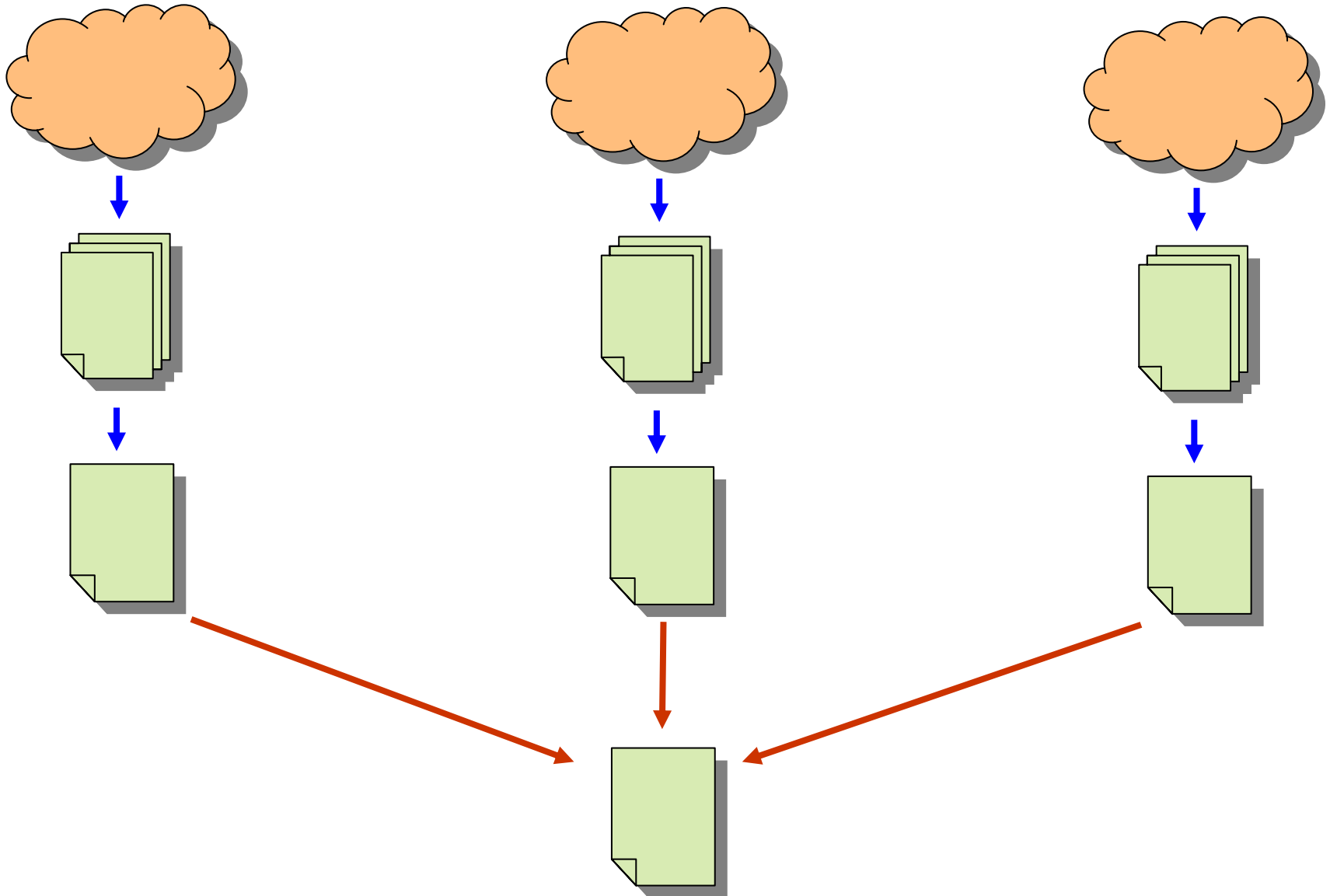
It's not design:

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

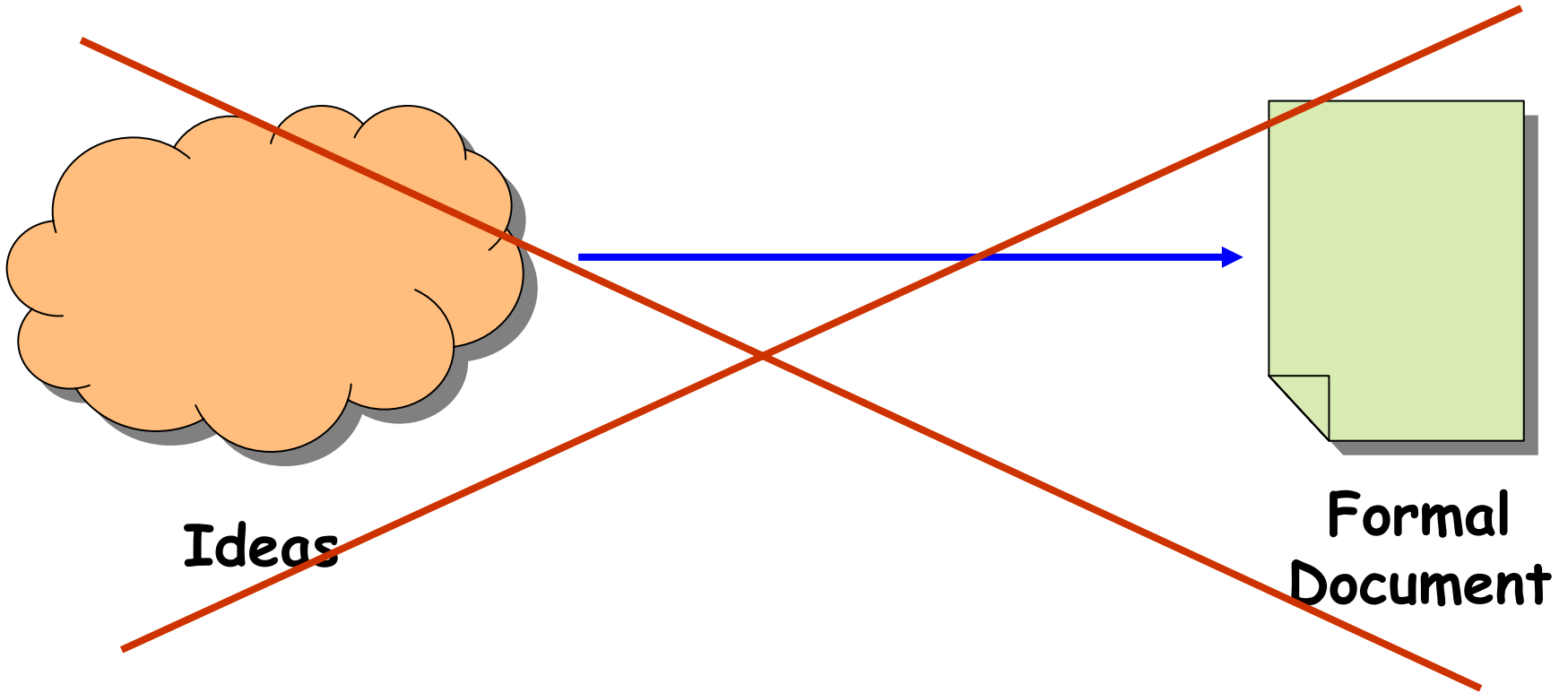
# General approach



# Merging formal requirements



# No natural language?



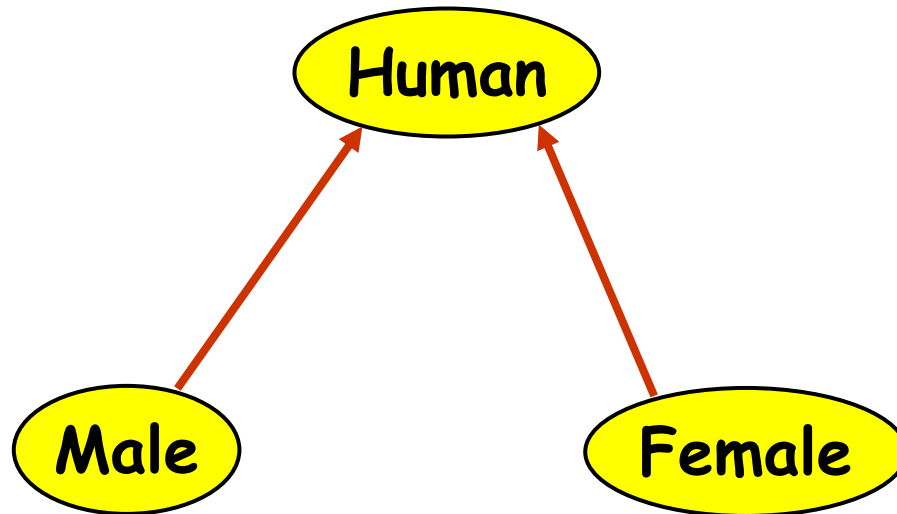


- Once we have a formal document
  - we can transform it back into a **natural language document**.
  - we can also transform it into a **graphical document**.
- There are many graphical notations out there.
- Be careful when choosing a graphical notation:
  - Does it have a **well defined semantics**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

# Graphical notations (cont.)



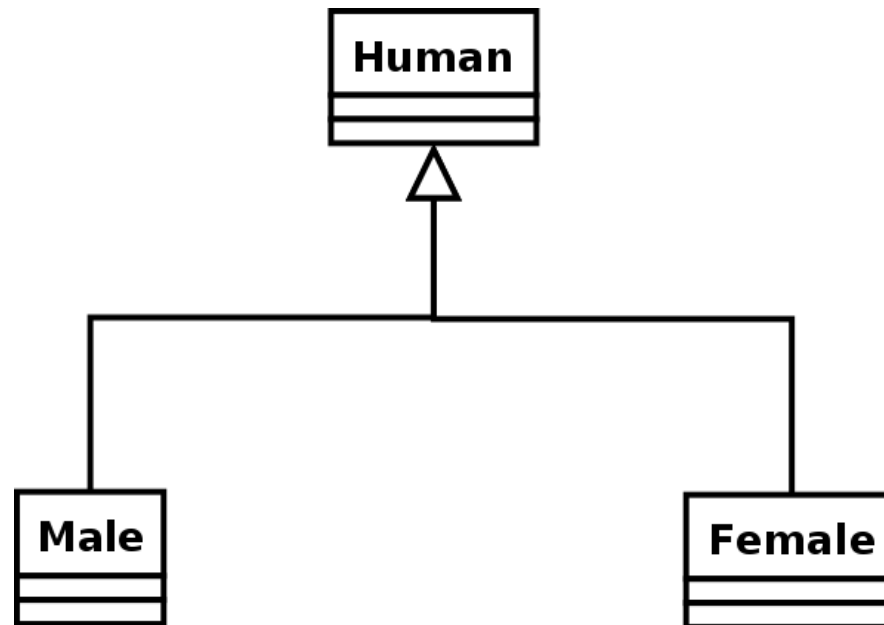
- Sets as Classes
- Subsets as Subclasses



# Graphical notations (cont.)



- Sets as Classes
- Subsets as Subclasses





- Functions



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 }

$(\text{cooling\_system} = \text{on}) \Rightarrow (\text{heating\_system} = \text{off})$

$(\text{heating\_system} = \text{on}) \Rightarrow (\text{cooling\_system} = \text{off})$



## Switch on event

```
switch_on_cooling_system =  
SELECT  
  cooling_system = off &  
  current_temperature > upper_limit  
THEN  
  cooling_system := on  
END
```



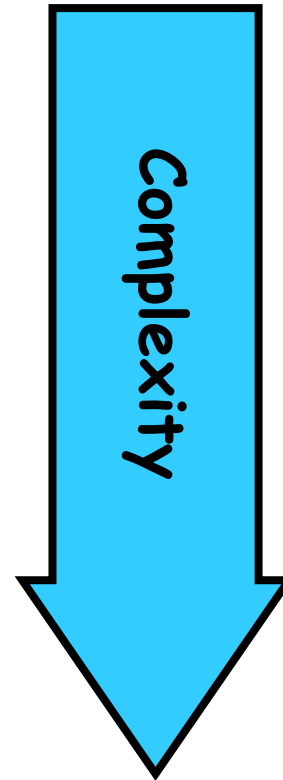
## Switch on event

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



## Categories

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





How should we formalize the requirements?

## The Z notation

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



## The B Method

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



## Other Candidates

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

**In the end, it is all just math**



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

- Accepts B (without refinement)
- Developed by Michael Leuschke, 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
- <http://alloy.mit.edu/>



## 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>



- 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



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!



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



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)

---



Barry W. Boehm: *Software Engineering Economics*, Prentice Hall, 1981.

Fred Brooks: *No Silver Bullet - Essence and Accident in Software Engineering*, in *Computer* (IEEE), vol. 20, no. 4, pages 10-19, April 1987.

John B. Goodenough and Susan Gerhart: *Towards a Theory of Test: Data Selection Criteria*, in *Current Trends in Programming Methodology*, ed. Raymond T. Yeh, pages 44-79, Prentice Hall, 1977.

Esther Derby: *Building a Requirements Foundation through Customer Interviews*, [www.estherderby.com/articles/buildingarequirementsfoundation.htm](http://www.estherderby.com/articles/buildingarequirementsfoundation.htm).

Éric Dubois, J. Hagelstein and A. Rifaut: *Formal Requirements Engineering with ERAE*, in *Philips Journal of Research*, vol. 43, no.  $\frac{3}{4}$ , pages 393-414, 1988.

Ellen Gottesdiener: *Requirements Workshops: Collaborating to Explore User Requirements*, in *Software Management 2002*, available at [www.ebgconsulting.com/pubs/Articles/ReqtsWorkshopsCollabToExplore-Gottesdiener.pdf](http://www.ebgconsulting.com/pubs/Articles/ReqtsWorkshopsCollabToExplore-Gottesdiener.pdf)

# Bibliography (2/4)

---



Gerald Kotonya & Ian Sommerville: *Requirements Engineering: Processes and Techniques*, Wiley, 1998.

IEEE: *IEEE Recommended Practice for Software Requirements Specifications*, IEEE Std 830-1998 (revision of IEEE Std 830-1988), available at [ieeexplore.ieee.org/iel4/5841/15571/00720574.pdf?arnumber=720574](http://ieeexplore.ieee.org/iel4/5841/15571/00720574.pdf?arnumber=720574).

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

Mike Mannion and Barry Keepence: *SMART Requirements*, in *ACM SIGSOFT Software Engineering Notes*, vol. 20, no. 2, pages 42-47, April 1995.

Bertrand Meyer: *On Formalism in Specifications*, in *Software (IEEE)*, pages 6-26, January 1985; available at [se.ethz.ch/~meyer/publications/ieee/formalism.pdf](http://se.ethz.ch/~meyer/publications/ieee/formalism.pdf).

[OOSC] Bertrand Meyer: *Object-Oriented Software Construction*, 2<sup>nd</sup> edition, Prentice Hall, 1997.

Peter Naur: *Programming with Action Clusters*, in *BIT*, vol. 3, no. 9, pages 250-258, 1969.

# Bibliography (3/4)

---



Shari Lawrence Pfleeger and Joanne M Atlee: *Software Engineering*, 3<sup>rd</sup> edition, Prentice Hall, 2005.

Laura Scharer: *Pinpointing Requirements*, in *Datamation*, April 1981. Also available at [media.wiley.com/product\\_data/excerpt/84/08186773/0818677384-2.pdf](http://media.wiley.com/product_data/excerpt/84/08186773/0818677384-2.pdf).

SEI (Software Engineering Institute): *CMMISM for Software Engineering, Version 1.1, Staged Representation (CMMI-SW, V1.1, Staged)*, 2005, available at [www.sei.cmu.edu/publications/documents/02.reports/02tr029.html](http://www.sei.cmu.edu/publications/documents/02.reports/02tr029.html).

Southwell et al., cited in Michael G. Christel and Kyo C. Kang, *Issues in Requirements Elicitation*, Software Engineering Institute, CMU/SEI-92-TR-012 and ESC-TR-92-012, September 1992, available at [www.sei.cmu.edu/pub/documents/92.reports/pdf/tr12.92.pdf](http://www.sei.cmu.edu/pub/documents/92.reports/pdf/tr12.92.pdf).

Becky Winant: *Requirement #1: Ask Honest Questions*, [www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=3264](http://www.stickyminds.com/sitewide.asp?Function=edetail&ObjectType=COL&ObjectId=3264).

# Bibliography (4/4)

---



Jeannette M. Wing: *A Study of 12 Specifications of the Library Problem*, in *Software (IEEE)*, vol. 5, no. 4, pages 66-76, July 1988.

Ralph Young: *Recommended Requirements Gathering Practices*, in *CrossTalk, the Journal of Defense Software Engineering*, April 2002, available at [www.stsc.hill.af.mil/crosstalk/2002/04/young.html](http://www.stsc.hill.af.mil/crosstalk/2002/04/young.html).