ETH Zürich
Chair of Software Engineering

Distributed Software Engineering Laboratory

Peter Kolb, Bertrand Meyer, Martin Nordio

Requirements Analysis

Statements about requirements: Brooks

Source*: Brooks 87

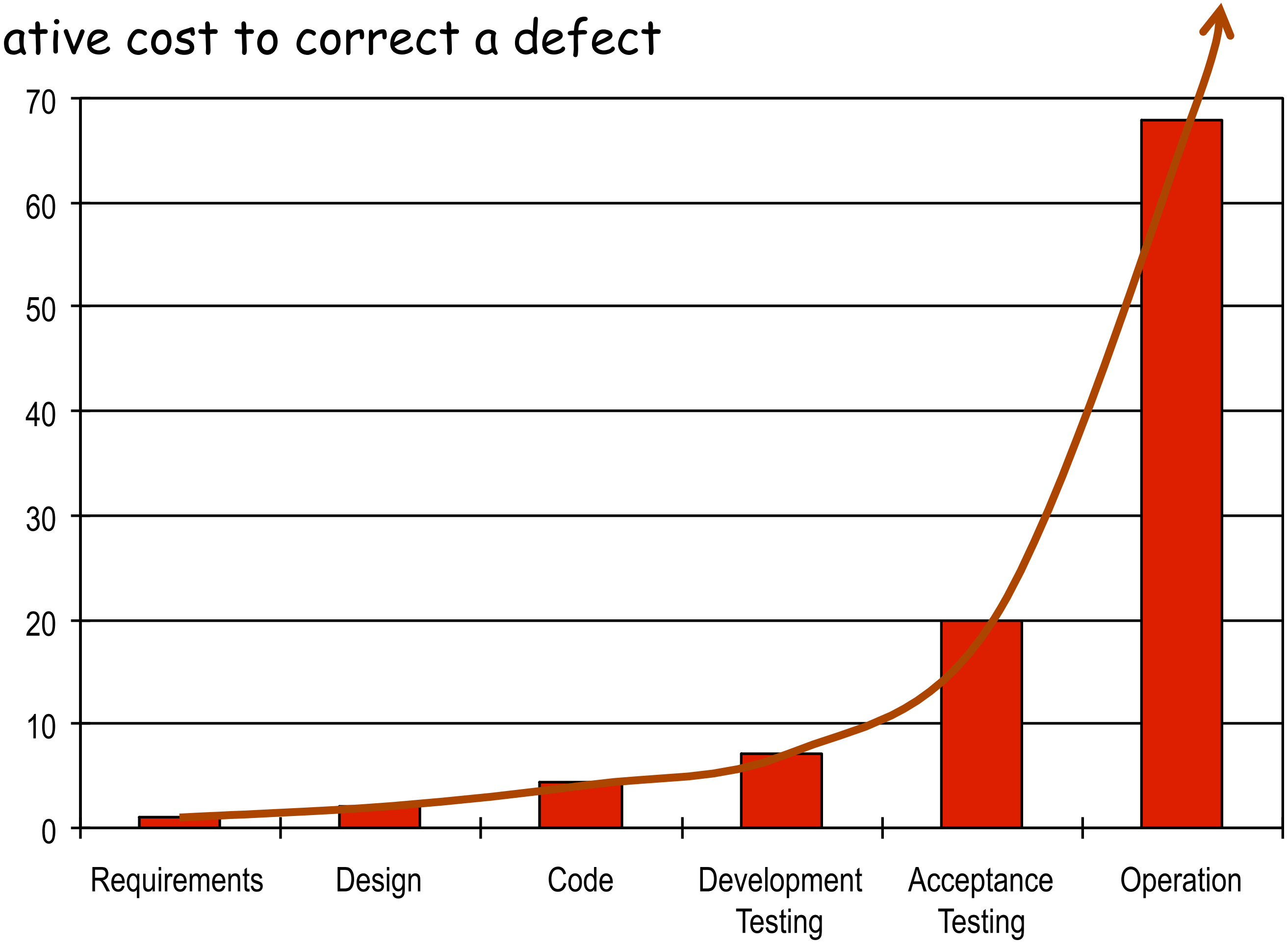
The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the 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

Source*: Wing 88

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.

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

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

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:

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*

Components of requirements

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

15 quality goals for requirements

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

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

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

Difficulties of requirements

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

Bad requirements

Source: Wiegers

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

Better:

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

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

Bad requirements

Source: Wiegers

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

Better:

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

Bad requirements

Source: Wiegers

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

Better:

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

The two constant pitfalls

- Committing too early to an implementation

  Overspecification!

- Missing parts of the problem

  Underspecification!

A simple problem

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

"Improved"

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. 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 *BL*s).

"Improved"

Source: Meyer 85

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 *BL*s).

Contradiction Noise Ambiguity

Overspecification Remorse Forward reference

The formal specification

where

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

$EQUIVALENT\ (b) \equiv$
$\{s \in \mathbf{seq}[CHAR] \mid$
$length\ (s) = length\ (b)$ **and**
$(\forall i \in 1..length\ (b),$
$s(i) \neq b(i) \Rightarrow$
$s(i) \in BREAK\_CHAR$ **and**
$b(i) \in BREAK\_CHAR)\}$

$max\_line\_length\ (s) \equiv$
$max\ (\{j-i \mid$
$0 \leq i \leq j \leq length\ (s)$ **and**
$(\forall k \in i+1..j,$
$s(k) \neq new\_line)\})$

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 sequences 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:

$FEWEST\_LINES\ (SSC) \equiv$
$MIN\_SET\ (SSC,$
$number\_of\_new\_lines)$

where the function *number_of_new_lines* is defined by:

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

and **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*:

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

with

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

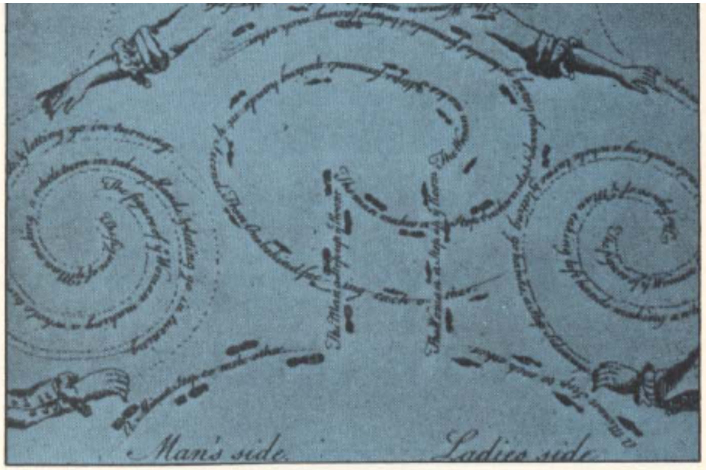
The dot operator denotes the composition of relations (see box). A look at

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

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 oversize words. Clearly, a robust and complete specification should include (along with *goal*) another relation, say, *exceptional_goal*, whose domain is *INPUT* − **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.[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

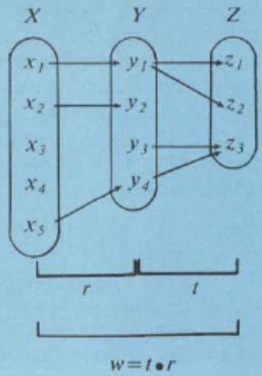


Composition of relations

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 $t\ (x,y)$ hold.

Thus, in the example illustrated, w holds for the pairs $<x_1, z_1>$, $<x_1, z_2>$, and $<x_5, z_3>$ (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

15 quality goals for requirements

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

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

Verifiable requirements

Adapted from: IEEE

Non-verifiable :

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

Verifiable:

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

Practical advice

Favor precise, falsifiable language over pleasant generalities

Complete requirements

Complete with respect to what?

Definition from IEEE standard (see next) :

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

- *All significant requirements, whether relating to functionality, performance, design constraints, attributes, or external interfaces. In particular any external requirements imposed by a system specification should be acknowledged and treated.*
- *Definition of the responses of the software to all realizable classes of input data in all realizable classes of situations. Note that it is important to specify the responses to both valid and invalid input values.*
- *Full labels and references to all figures, tables, and diagrams in the SRS and definition of all terms and units of measure.*

Completeness

Completeness cannot be "completely" defined

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

- Commands x Queries

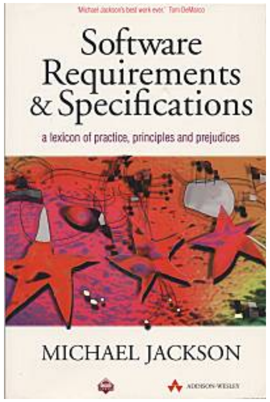
to verify that every effect is defined

Practical advice

Think negatively

The two parts of requirements

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

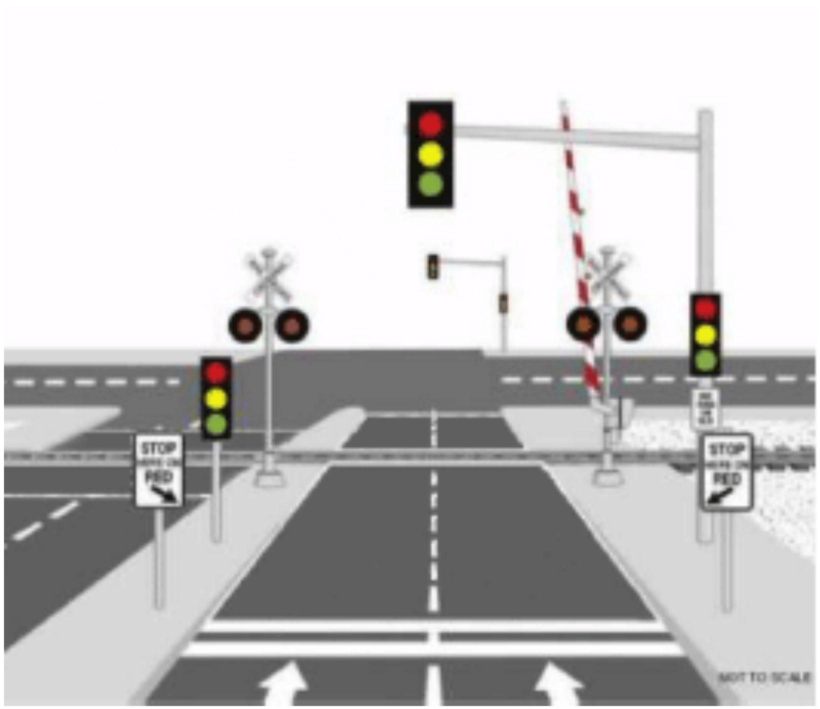


Jackson's view: define success as

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

- *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

Standards and Methods

The purpose of standards

Software engineering standards:

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

IEEE 830-1998

"IEEE Recommended Practice for Software Requirements Specifications"

Approved 25 June 1998 (revision of earlier standard)

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

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

15 quality goals for requirements

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

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

IEEE Standard: definitions

Contract:

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

Customer:

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

Supplier:

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

User:

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

IEEE Standard

Basic issues to be addressed by an SRS:

- Functionality
- External interfaces
- Performance
- Attributes
- Design constraints imposed on an implementation

IEEE Standard

Recommended document structure:

1. Introduction

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

2. Overall description

- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 Constraints
- 2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index

Practical advice

Use the recommended IEEE structure

Practical advice

Write a glossary

Recommended document structure

1. Introduction

- 1.1 Purpose
- 1.2 Scope
- 1.3 Definitions, acronyms, and abbreviations
- 1.4 References
- 1.5 Overview

2. Overall description

- 2.1 Product perspective
- 2.2 Product functions
- 2.3 User characteristics
- 2.4 Constraints
- 2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index

Example section: scope

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

Example section: product perspective

Describe relation with other products if any.

Examples:

- System interfaces
- User interfaces
- Hardware interfaces
- Software interfaces
- Communications interfaces
- Memory
- Operations
- Site adaptation requirements

Example section: constraints

Describe any properties that will limit the developers' options

Examples:

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

Recommended document structure

1. Introduction

1.1 Purpose

1.2 Scope

1.3 Definitions, acronyms, and abbreviations

1.4 References

1.5 Overview

2. Overall description

2.1 Product perspective

2.2 Product functions

2.3 User characteristics

2.4 Constraints

2.5 Assumptions and dependencies

3. Specific requirements

Appendixes

Index

Specific requirements (section 3)

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

Examples:

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

Possible section 3 structure

3. Specific requirements

- 3.1 External interfaces
  - 3.1.1 User interfaces
  - 3.1.2 Hardware interfaces
  - 3.1.3 Software interfaces
  - 3.1.4 Communication interfaces
- 3.2 Functional requirements
  - …
- 3.3 Performance requirements
  - …
- 3.4 Design constraints
  - …
- 3.5 Quality requirements
  - …
- 3.6 Other requirements
  - …

Requirements under agile methods

Under XP: requirements are taken into account as defined at the particular time considered

Requirements are largely embedded in test cases

Benefits:

- Test plan will be directly available
- Customer involvement

Risks:

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

Practical advice

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

Disregard those that contradict proven software engineering principles.

Requirements elicitation

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

After: Kotonya & Sommerville 98

Premature design?

Combined requirements?

Unnecessary requirements?

Conformance with business goals

Ambiguity

Realism

Testability

Using natural language for requirements

Keys are:

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

Advice on natural language

Apply the general rules of "good writing" (e.g. Strunk & White)

Use active form

(Counter-example: "*the message will be transmitted…*")

This forces you to state who does what

Use prescriptive language ("*shall…*")

Separate domain properties and machine requirements

Take advantage of text processing capabilities, within reason

Identify every element of the requirement, down to paragraph or sentence

For delicate or complex issues, use complementary formalisms:

- Illustrations (with precise semantics)
- Formal descriptions, with explanations in English

Even for natural language specs, a mathematical detour may be useful

Advice on natural language

After Mannion & Keepence, 95

- 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

Case study questions

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

The need for an iterative approach

Source: Southwell 87

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

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

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

Before elicitation

At a minimum:

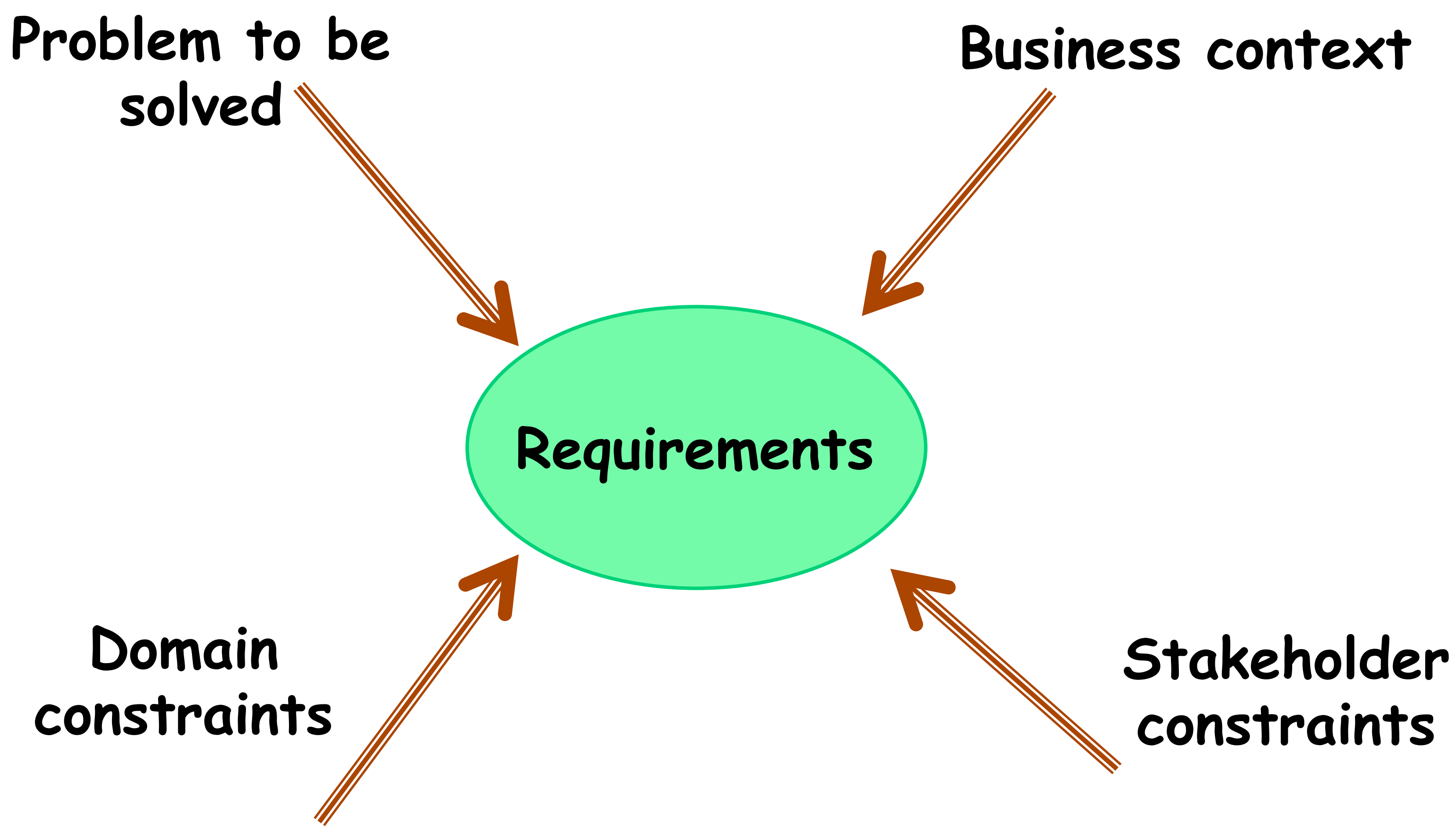
- Overall project description
- Draft glossary

Requirements elicitation: overall scheme

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

The four forces at work

After: Kotonya & Sommerville 98



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"

Stereotypes

Source: Scharer 81

How developers see users

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

How users see developers

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

Requirements elicitation: who?

Users/customers

Software developers

Other stakeholders

Requirements engineers (analysts)

Requirements elicitation: what?

Example questions:

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

Requirements elicitation: how?

- Contract
- Study of existing non-computer processes
- Study of existing computer systems
- Study of comparable systems elsewhere
- Stakeholder interviews
- Stakeholder workshops

Building stakeholders' trust

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

Need to build trust progressively:

- Provide feedback, don't just listen
- Justify restrictions
- Reinforce trust through evidence, e.g. earlier systems, partial prototypes
- Emphasize the feasible over the ideal

Study of existing systems

Non-computerized processes

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

Existing IT systems

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

Stakeholder interviews

After: Winant 02

Good questions:

- Are egoless
- Seek useful answers
- Make no assumptions

"Context-free" questions:

- "Where do you expect this to be used?"
- "What is it worth to you to solve this problem?"
- "When do you do this?"
- "Whom should I talk to?" "Who doesn't need to be involved?"
- "How does this work?" "How might it be different?"

Also: meta-questions: "Are my questions relevant?"

Probe further

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?

Uncovering the implicit

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

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

Requirements workshops

After: Young 01

Often less costly than multiple interviews

Help structure requirements capture and analysis process

Dynamic, interactive, cooperative

Involve users, cut across organizational boundaries

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

Help manage users' expectations and attitude toward change

Knowing when to stop elicitation

Keep the focus on scope

Keep a list of open issues

Define criteria for completeness

After elicitation

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

Make decisions on contentious issues

Finalize scope of project

Go back to stakeholders and negotiate

15 quality goals for requirements

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

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

Practical advice

Treat requirement elicitation as a mini-project of its own

Object-Oriented Requirements Analysis & Abstract Data Types

Use Cases (scenarios)

One of the UML diagram types

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

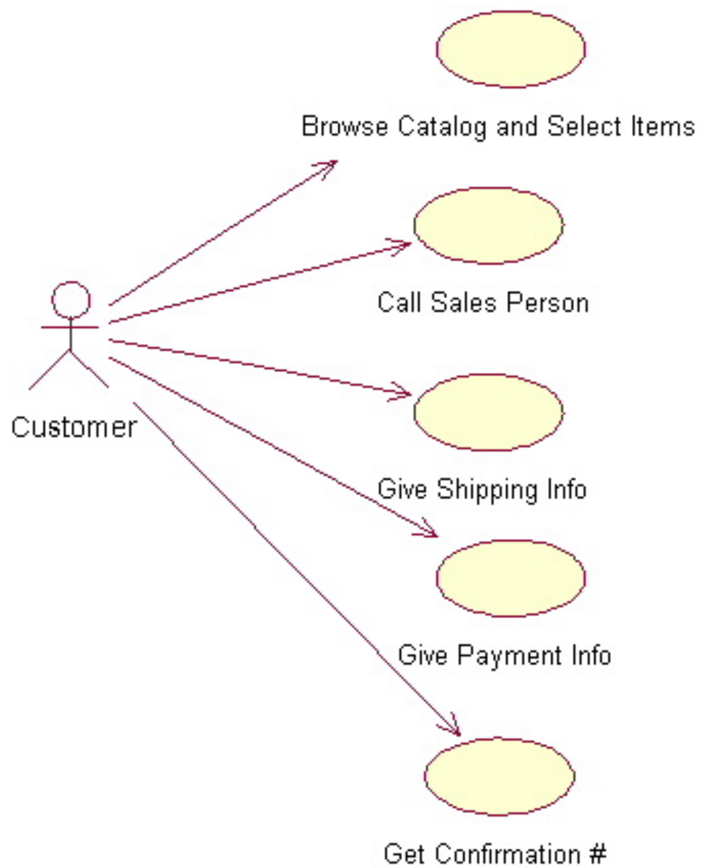
A good use case must:

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

Use case example

Place an order:

- Browse catalog & select items
- Call sales representative
- Supply shipping information
- Supply payment information
- Receive conformation number from salesperson



May have precondition, postcondition, invariant

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.

My view

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

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

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

Major application: for testing

Practical advice

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

Analysis classes

```
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
```

Schedules

```
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
```

Contracts

Feature precondition: condition imposed on the rest of the world

Feature postcondition: condition guaranteed to the rest of the world

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

Why contracts

Specify semantics, but abstractly!

(Remember basic dilemma of requirements:

- Committing too early to an implementation

  Overspecification!

- Missing parts of the problem

  Underspecification!

)

Segment

```
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 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

    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
```

Commercial

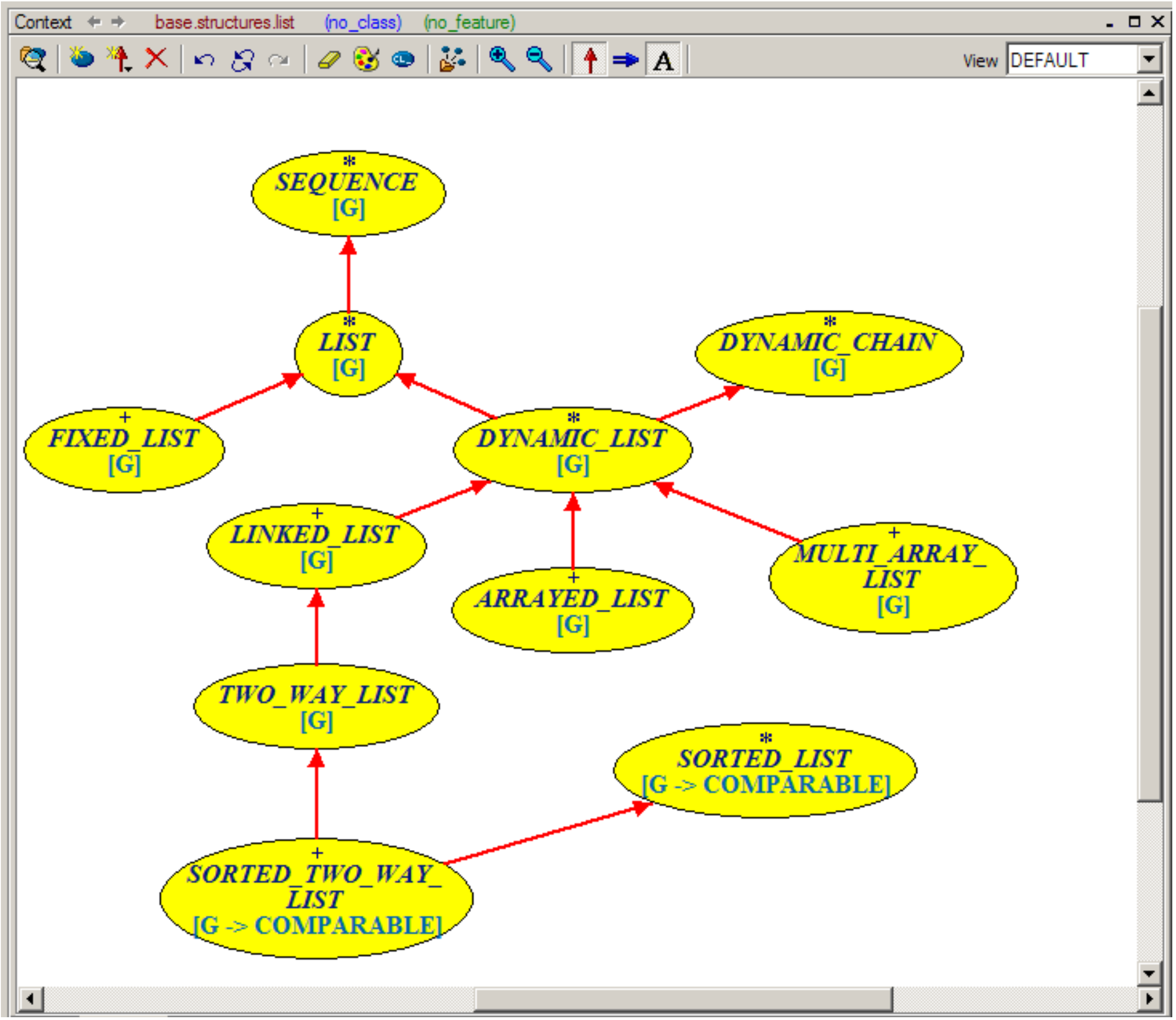
```
note
    description: "Advertizing segment"
deferred class COMMERCIAL inherit
    SEGMENT
        rename sponsor as advertizer end
feature
    primary: PROGRAM deferred
        -- Program to which this
        -- commercial is attached
    primary_index: INTEGER deferred
        -- Index of primary
```

```
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
```

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

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

Practical advice

Write ADT specifications for delicate parts of the system requirements

Conclusion

Key lessons

Requirements are software

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

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

Since requirements will change, seamless approach is desirable

Distinguish domain properties from machine properties

- Domain requirements should never refer to machine requirements!

Key lessons

Identify & involve all stakeholders

Requirements determine not just development but tests

Use cases are good for test planning

Requirements should be abstract

Requirements should be traceable

Requirements should be verifiable (otherwise they are wishful thinking)

Object technology helps

- Modularization
- Classifications
- Contracts
- Seamless transition to rest of lifecycle

Key lessons

Formal methods have an important contribution to make:

- Culture to be mastered by requirements engineers
- Necessary for critical parts of application
- Lead to ask the right questions
- Proofs & model checking uncover errors
- Lead to better informal requirements
- Study abstract data types
- Nothing to be scared of

Bibliography (1/4)

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.

Éric Dubois, J. Hagelstein and A. Rifaut: *Formal Requirements Engineering with ERAE*, in *Philips Journal of Research*, vol. 43, no. ¾, 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

Bibliography (2/4)

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

IEEE: *IEEE Recommended Practice for Software Requirements Specifiations*, IEEE Std 830-1998 (revision of IEEE Std 830-1988), available at 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, also at se.ethz.ch/~meyer/publications/ieee/formalism.pdf.

[OOSC] Bertrand Meyer: *Object-Oriented Software Construction*, 2nd 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*, 3rd 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.

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.

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.

Becky Winant: *Requirement #1: Ask Honest Questions*, 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.