Guidelines for Requirements Analysis in Students' Projects

Information Systems Group
Faculty of Electrical Engineering, Mathematics and Computer Science
University of Twente

Introduction

This document provides guidelines about how to do a requirements analysis and how to write a requirements specification. It gives hints about what you could do and warns you about things that you should not do. It is not a method that you can follow step by step. Problems are different, and what works well in one case would not be the best approach in another case. The available time can take from a few weeks to several months. And your skills and experience also determine what are good techniques to use. If a particular specification technique is treated in a course you haven't taken, then it might not be a good idea to try it out on an important job. So you have to decide for yourself what is the best to do in your project.

These guidelines are written primarily for master and bachelor students of Business Information Technology, but could be used by others. The document is self-contained, but refers to other sources for detailed descriptions of techniques. Many references are given to the book used in the bachelor course Requirements Engineering (232081), S. Lauesen: Software Requirements [Lau02]. The UT library has a copy that is permanently available (it may not leave the library).

Outline of the Guidelines

After an introductory chapter

0. **What you should know before you start**

the remainder or these guidelines is structured as a series of steps that comprise an idealized life cycle of a requirements specification:

1. **Analysing the problem and the problem context**
   After this step, you have an understanding of the problem context and you have learnt what should be improved and why.
2. **Defining the ideal solution**
   After this step, you know what, in principle, the best solution to the identified problem(s) would be.
3. **Defining a realistic solution**
   After this step, it has been defined what the system, for which you are going to do a requirements analysis, should achieve. Moreover, relevant stakeholders agree about its mission.
4. **Gathering requirements**
   After this step, you know what people would like the system to do and which requirements and constraints there are.
5. **Writing a requirements specification**
   After this step, you have a readable first version of the requirements specification that can be discussed with involved persons. We distinguish four separate concerns
   5.1. The contents of a requirements specification
   5.2. Specification techniques
   5.3. Readability and linguistic issues
   5.4. Quality check
6. **Validating the requirements specification**
   After this step, you have made sure that the requirements reflect what the relevant stakeholders want from this project. This is the requirements specification that you deliver.
7. **Maintaining the requirements specification**
   The world goes on, and new requirements may come up. This is outside the scope of most students' projects, but for the sake of completeness we discuss it briefly.

The ideal requirements process would follow these steps in consecutive order. As you may have guessed, the ideal requirements process does not occur in practice. But for the purpose of organising the material, it makes sense to discuss the steps one by one.

Each chapter treats a single step in the requirements specification life cycle. An outline gives essential questions that you should ask yourself (and others) and what to do about these. The remainder of the chapters treat specific topics in more detail. Appendices at the end of the document give yet more detail and references to further literature.

Not every topic is applicable in every context. Read all the outlines and study other topics as appropriate.

About this document

These Guidelines have been compiled and are maintained by the Information Systems group at the University of Twente.

Feedback is welcome! It helps us to improve future versions of the Guidelines.
Please contact Klaas Sikkel, room ZI 3102, email: k.sikkel@utwente.nl.

Contents

0. What you should know before you start

The purpose of this chapter is to give you some general words of advice. You should read this before you start your requirements analysis.

Way of thinking – What are the essential questions?

- ❑ What is a requirements specification?
- ❑ How do you obtain a requirements specification?

0.1 The requirements process

Requirements analysis is for a large part a social activity. The requirements analyst's job is to find what relevant stakeholders want and lay that down in a suitable specification (and not to invent the requirements himself). Gause and Weinberg [GW89] define a *requirements process* as

> the part of [system] development in which *people attempt to discover what is desired*.

In the early days of computing, it was thought that the requirements analyst's job is to find out what is *needed*. This presupposes that there is some objective need, and analysis will reveal what that need is. In many projects, this is not the case. There are various things that could be desired for various reasons. Moreover, many relevant persons do not have a clear picture of their own desires – the process of requirements discovery helps them to find out what they really want.

To make things more complicated, any project has a number of different stakeholders with different interests, and it is usually not feasible to incorporate all desires of all stakeholders. Choices have to be made and somebody has to put some effort into making the stakeholders accept the resulting requirements specification.

0.2 The requirements specification life cycle

In this section we elaborate a requirements specification life cycle of seven steps. In the next section we will argue that it doesn't work that way, and in practice you won't be able to strictly separate these steps.

What, then, is the point of introducing this model? It's a *reference model*, describing the ideal case. Even though you will never meet the ideal case, it helps to keep structure and put things in the right place. For example, if you return from a chaotic focus group meeting which has done bits of steps 1, 2, 4, and 6 in random order, you can get some structure in your equally chaotic notes by ordering them according to these steps.

It's like the waterfall model in Software Engineering – the first thing you learn in an SE course, despite the fact that nobody ever could make it work that way. It's the lucid enumeration of steps that makes it worth knowing it.

In the generic requirements process described here we distinguish different phases

- Finding out what the problem is, and what kind of solution is desired (steps 1–3)
- Drawing up a requirements specification for the desired solution (steps 4–6)
- Maintaining the requirements specification when requirements change later on in the project (step 7)

In each phase we can distinguish four different kinds of activities:

- *Preparation*: getting organized before you start, finding out what you are going to do and whom you may want to talk to, etc.
- *Elicitation*: going out and finding requirements, by asking people, observing, reading documents, etc.
- *Engineering*: putting things together: specifying what elicited and observed, organizing and combining things. There is always an element of *design* involved.
- Negotiation and decision making. This is *politics*, rather than engineering, but is an inevitable part of getting a requirements specification accepted.

The complete life cycle model is shown in Figure 1. The phases cycle through the different activities, yielding our seven steps:

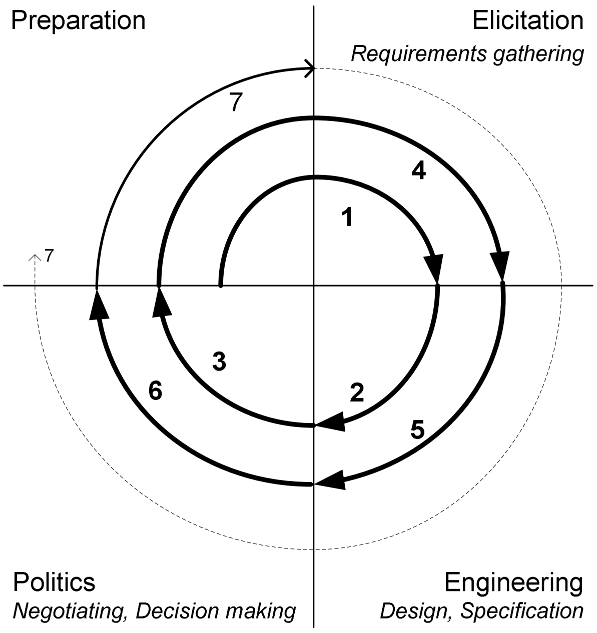


Figure 1: The requirements life cycle

1. ***Analysing the problem and the problem context***
2. ***Defining the ideal solution***
3. ***Defining a realistic solution***
4. ***Gathering requirements***
5. ***Writing a requirements specification***
6. ***Validating the requirements specification***
7. ***Maintaining the requirements specification***

The maintenance phase is never finished and can cycle on forever. (But we can anticipate this).

0.3 From business problem to system specification

Another way to look at the relation between problem and solution is shown in Figure 2.

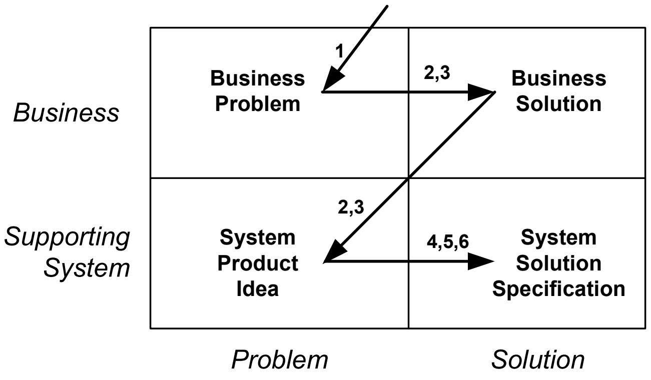


Figure 2: The Z model

We distinguish between problem and solution, and between business and supporting (software) system. In a perfectly rational world, a requirements analysis process would follow the arrows in the diagram.

In a narrow sense, requirements analysis is only concerned with the last arrow. Somebody has suggested that a system for a particular purpose can be developed (or bought) and your task as a requirements analyst is to find the requirements for that system. However, in order to find these requirements, it is important to know *why* this system is needed, what problem it will solve – otherwise it's not possible to determine the requirements.

A problem *always* arises in the real world. Even when it's clear that the system is to blame. E.g. "our system is too slow." It would not be a problem if people would not depend on that system for doing the particular job they do. In other circumstances (e.g. the same company 5 years ago) the same system might not be experienced as being to slow. The idea to design, replace, or upgrade a system doesn't arise because having the system is a goal in itself, the system is needed for some purpose.

It is called "business problem" because most requirements engineering is done for systems that have some business purpose, but it doesn't have to be related to commercial business.

The solution to a business problem is always a business solution. It is possible that this solution involves a computer system. It is tempting to think that acquiring a new system may solve a business problem (this is a mistake that is often made). *Using* a new system can be the solution to a problem. Acquiring the system isn't suffficient, the system has to fit into the way the work is done – or perhaps the work has to be reorganised, so as to exploit the capabilities of the new system.

In perfectly rational top-down design process, one would first define a business solution to address the business problem, then consider what kind of system is needed to support the business solution and finally draw up a requirements specification. After the arrows in Figure 2, this is called the Z model.

To make sure that we do requirements analysis for a system that helps addressing the *right* problem, we start with step 1 – identifying the problem. Steps 2 and 3 yield an idea of the solution and the system needed to realize that solution. After that we can do a more detailed requirements analysis in steps 4–6.

At least, that's the theory...

0.4 Why isn't there a proper method?

Life would be a lot easier with a method that you could follow step by step. Unfortunately, our life cycle model doesn't pretend to be that kind of method. In fact no such method exists for requirements analysis.

There is no method that addresses all cases

For each project you have to decide which issues are important and need a lot of care, and which issues are trivial or do not apply. These guidelines are no substitute for thinking for yourself, and *you* have to judge what is needed in your project.

Requirements analysis projects differ a lot in scope and nature. Some examples from projects carried out by M.Sc. students:

1. A commercial bank has a problem with customer loyalty. Obtaining new customers by means of marketing actions seems to work, but the bank isn't able to retain these customers for a long time. Can appropriate CRM software help them to increase the loyalty of their customer base?

The focus in this project is more on organizational practices than on the technical support system. In this project something was implemented in the end, but initially it was not at all clear what the solution should look like. But it was evident that a system won't help if the bank's employees are unable or unwilling to use it properly. Steps 1–6 were carried out, but the emphasis was on steps 1, 4, and 6.

2. A telecom company wants to find out how it could rent telephone services to corporate clients, making use of VoIP (Voice over IP) technology.

This is primarily a technical project. Not much study has to be done about how people would use a VoIP telephone, because it should work as a regular telephone, and possibly clients shouldn't even be aware of the difference. Steps 3–6 were carried out in this case (the result of step 2, the ideal solution, was given as a starting point for the project) but the emphasis was on steps 4 and 5.

3. The Police department in a region in the north of the Netherlands has difficulties in providing statistical material to the Ministry of Justice. Sometimes when the Ministry asks for statistics about a particular type of crime, they have to go through all the database records to find the requested numbers by hand.

The stated problem is clear, but it is a symptom of an underlying problem that was hard to find and harder to solve. In this project only steps 1–3 were carried out.

The steps in a requirements analysis process do not take place in consecutive order

Only in the ideal situation, you do step 1 first, then step 2, and so on, without retracing your steps. In practice you will find it hard to separate analysing the problem (step 1) from eliciting the requirements (step 4). Also, it makes sense to combine requirements elicitation (step 4) with writing down the elicited requirements (step 5).

Many projects, and some excellent requirements analysis methods, start with step 3. If the project goals are straightforward and you are asked to draw up a requirements specification for a system with a clear purpose, step 3 is a natural starting point. This implies that *somebody else* has already performed steps 1 and 2, found out what the problem and the ideal solution was, decided to set up a project and engage you as a requirements engineer. If this is the case, you can – and should – find the results of the problem analysis. If these don't exist, e.g. if the project is driven by a solution, rather than a problem, you should consider doing some problem analysis after all.

However, in many cases, including most cases in which our students do a requirements analysis, there is some idea about the problem, but it is not immediately obvious what the best solution is – otherwise they wouldn't have asked the university.

Many systems fail, despite the fact that they fulfil the requirements, because the problem is poorly understood and a solution is built that doesn't address the real problem. For this reason we insist that step 1 is part of the requirements analysis.

Problem-solution co-refinement

It's a very good idea to define the problem first, and then the solution. If it's a difficult problem with no easy solution, there is a complex relationshop between problem and solution. The nature of a possible solution determines what problems you can solve, and if we don't know the solution yet we might not know exactly which problem we *can* solve. Empirical studies have shown that refining the solution and refining the problem go hand in hand [Cro89]. That's why you always have to do some rework on previous steps, no matter which method you follow.

The method does not work

You do the work. The method is just a set of guidelines. The method is not responsible for your work products, nor are the authors of the method. You are responsible yourself.

Step 1. Analysing the problem and the problem context

The purpose of this step is to find out what the problem is and, equally important, to understand the situation in which the problem occurs. It is not the purpose of this step to think about possible solutions. That comes later, after we have learnt enough about the problem.

Way of thinking – What are the essential questions?

- ❑ What are the problems (goals, desires) and what are the causes for these problems?
- ❑ Is the stated problem the real problem or it is a symptom of an underlying problem?
- ❑ Who are the stakeholders?
- ❑ What will be the impact if the problems are resolved / the goals are accomplished?

Approach – How to find answers to these questions?

It makes sense to learn something about what is going on, what are the causes for the problems and which parties have an interest in (not) solving the problem. To that end you have to do two things:

- identify (groups of) stakeholders
- interview relevant persons

Your supervisor or the client can help you drawing up an initial list of persons you might want to speak to (and talking to these you may become aware of other stakeholders to be considered). If there are relevant documents about the current system, it could be worthwhile to read those first. If you know what you’re talking about, you’ll get better results.

The list of “context-free questions” in Appendix A could be a good starting point. Some other points are elaborated below.

Product – What do you write down?

Lay down your problem analysis in a short paper. Target audience for this paper are the stakeholders. They should be able to find out, as easily as possible, whether you have captured their problem appropriately. Hence it is important that the analysis is easily readable and to the point. Making it short and readable is a *lot* more work than just summing up what you’ve found. But it’s well worth the effort if you want to get feedback and gain credibility with the client and other stakeholders.

Follow-up – What do you do with this document?

- Make sure that you have a good enough version (if possible, consult your supervisors)
- Circulate it to relevant persons and ask for their feedback
- If needed: adapt it, based on the feedback
- Include the adapted version as a chapter or an appendix to your final report.

1.1 What is the problem?

How much time, effort and skill it takes to identify the problem varies from case to case.

There are (few) projects in which the problem is clear. Consider a project to develop a prototype for some technologically innovative gadget. You may find it interesting to know what people eventually will do with it, but the prime challenge in *this* project is in getting the technology working.

In some projects, finding the problem is very hard. For example in a situation where key persons have hidden agendas, it needs skill and tact to find out what is going on.

In some projects, the problem *appears* to be clear. But the problem that people experience is a symptom of a deeper, underlying problem, and it makes a lot more sense to solve the real problem than to address the symptom.

Problems at which level?

If you ask people which problems they experience, they often will tell you that properties of the current system (or their absence) are a problem. This is experienced as a problem, it directly bothers people. The *real* problem, however, is that they cannot perform some task effectively or efficiently. Adapting the system functions they complain about can be, but need not be the best solution. Perhaps is it better to reorganize the work, or to replace the whole system rather than to repair some functions.

Do not just ask what the problems are, always ask *why* this is experienced as a problem. Sometimes you have to ask "why" several times to find the real reason behind the reason behind the reason behind the problem.

Problem vs. solution

When you ask for problems, many people (including most students not trained in requirements engineering) will come up with solutions.

- A problem is a difference between what is experienced and what is desired.
- A solution is a way to reduce a problem

These two are related, but different. It is possible that there are different solutions for the same problem.

If you inquire about problems you may be told, e.g. "we need an ERP system." What is stated here is the absence of a solution. Again, we need to go up one level, and ask "why". There could be various reasons. Perhaps implementing an ERP system is indeed the best solution, perhaps there are also other solutions worth considering.

How important is a problem?

Not all problems are equally important. One way to get an indication is to ask the following questions (costs and benefits are not only financial).

- What are the costs when this problem is solved?
- What are the benefits if this problem is solved?
- What are the costs if the problem is not solved?
- What are the benefits if the problem is not solved?

If you want to get an idea about the *urgency* of a problem, you could add

- What are the costs if the problem is solved after one year?
- What are the benefits if the problem is solved after one year?

More about problem analysis

A course Problem Analysis and Software Requirements (232080) is part of the BIT master programme.

1.2 Organisational context

How is the project positioned in the organisation?

- How does the project fit in the organisation's strategy?
- What does management think about this project?
- Who is responsible for the project's funding (the client) and who is responsible for managing the project?

Goals

A problem is a problem because it prevents some goal from being realized. In perfectly logical world, you would first write down the goals and then look for problems obstructing these goals. Eliciting goals is a lot more difficult than making a list of problems. Many people are not willing or able to state their goals. Try to get some idea about the following issues:

- What are the goals of the organisation?
- Which personal goals (which are usually hidden) also play a role?
- What are the goals of the organisational unit? Are these different from the goals of the organisation as a whole?

The official goals of the organisation (typically: running the primary process effectively and efficiently) give some hold, and can be used in your problem analysis to motivate why a solution is needed. But keep an open mind for what is going on around you.

1.3 Stakeholders

A stakeholder to a project is someone who gains or loses something (could be functionality, revenue, status, compliance with rules, and so on) as a result of that project [AR04].

Stakeholders include

- the client (who pays for the system development),
- customers,
- system developers,
- direct users (who will work with the system),
- indirect users (e.g. who will get information from the system),
- system operators.

And there could be others, e.g.

- government bodies, having an interest that the law is not violated.

Alexander [Ale03] gives a simple but powerful model of stakeholder roles that can help you discover the stakeholders for your project.

In some cases you may consider an organisation or company to be a stakeholder. It is always better to think of concrete persons, rather than abstract bodies. ("Mr. Smith in the procurement department", rather than "company A&B"). A stakeholder group is *homogeneous* if all persons in that group want the same thing. This is not always the case.

If you want to involve stakeholders in the requirements process, you have to determine who represents a stakeholder group. There are several forms of representation:

- exhaustive (everybody in the group)
- representation by sample (choose the sample carefully of the group is not homogeneous)
- representation by surrogate (somebody who knows a group of stakeholders quite well).

Representation by surrogate ("our marketing department knows what our customers want") is always risky. If you don't have access to real users, you must read *The Inmates are Running the Asylum* [Coo99] before you attempt to write down other people's estimate of what the users would desire.

Stakeholders have different problems. Even in the unlikely event that there is only a single problem, stakeholders will experience this problem differently.

If you want to get clear which stakeholder has which problem, you could make a schema as follows:

Stakeholder ***Problem***	A	B	...
Problem 1			
Problem 2			
...			

1.4 Interviewing[1]

Interviewing is the most often used technique to learn about problems. It works fine, if you are aware of its limitations.

When you ask people about their daily tasks, they have difficulties explaining what they do and why they do things the way they do. Some people have hidden agendas and will not give honest answers.

Make sure you have the right interview partner, and not a surrogate. If you want to know the problems on the shop floor, you should talk to the people who do the work there, not to their managers.

Prepare yourself for the interview. If you know what you're talking about you will get a better response. Make a list of questions. The context-free questions in Appendix A can serve as inspiration. If you can make these questions more specific for the situation, that's better.

Despite this, an interview is not a question-and-answer session. Start with one issue, and most likely the interviewees will cover a number of questions when you let them talk. If they bring up issues that are relevant, but not on your list, even better. Use your list to check whether the issues are covered. If something hasn't been touched upon, you may bring it up.

When you discuss day-to-day problems with an unsatisfactory system, ask about *critical tasks*. When does the user work under stress? When is it important that nothing goes wrong?

As a general rule you should be polite and sensitive to the interview partner. Some people don't like to admit that they have problems. There is whole range of euphemism that roughly mean the same thing: challenges, things you find hard to deal with, concerns, issues, things that could be improved, ...

Some managers get offended if you ask "why", as they are not used to be questioned about their motives. If asking "why do you do this" doesn't work, you may ask "when do you do this" as a substitute.

[1] largely based on [Lau02], section 8.8.2.

Step 2. Defining the ideal solution

Armed with sufficient knowledge of what the problems are, we can start to think about a solution. Usually it makes sense to do that in two steps. A realistic goal – the subject of step 3 – is constrained by practical limitations. The purpose of this step is to find out what the client would *like* to achieve.

Way of thinking – What are the essential questions?

❑ What is the essential problem?

❑ What would be an ideal solution to this problem?

Approach – How to find answers to these questions?

If there is a single problem and everybody agrees that this is the problem that needs to be solved, step 2 is easy. If, however, there are various issues and different stakeholders experience different problems, this is not trivial. It has to be decided, somehow, what the essential problem is. In that case you have to discuss it with the client or perhaps organise a focus group with different stakeholders (see 4.5).

Product – What do you write down?

A brief text (maximum one page, preferably half a page) describing

- the essential problem,
- the proposed solution,
- a brief explanation about the motivation of the essential problem and the choices you made.

If there was a group session, you probably have a list of other problems and possible solutions. The explanation should make clear why *this* problem was chosen as the essential problem.

Follow-up – What do you do with this document?

This is not an official document (achieving the ideal solution is not an objective of the project), but it could be the most important page in the whole project. Check informally whether relevant stakeholders can agree with it. If they can, there is agreement about the focus of the project.
If, on the other hand, it turns out that some stakeholders have serious troubles with the choice of the essential problem or solution, you have achieved your first success! You have shown that the matter is more complicated and delicate than the client thought, and identified a potentially fatal risk for the project.

2.1 One essential problem

The goal of the project is to solve, in the best possible way, the essential problem. The solution may partially solve other problems as well, but the priorities must be clear. If you have multiple goals, all equally important, then sooner or later you will face design decisions that cannot fully satisfy these goals simultaneously and you'll have to favour one goal at the expense of another.

2.2 The client's goal vs. the project goal

There is a difference between *the external goal* or *client's goal* (what the client wants to achieve, e.g. increased sales) and the *project goal* (what the project intends to deliver, e.g. a system to support the sales process). The external goal provides a motivation for the project goal.

2.3 Business solution vs. software solution

The external goal is always to find a solution to a business problem (see the Z model in 0.3). The project goal could be on the software level (otherwise you weren't asked for a requirements analysis).

If the project goal is to come up with a software solution specfication, you should spend some words on the business solution to which your software solution will contribute.

Step 3. Defining a realistic solution

The purpose of this step is to define a realistic solution and to gain acceptance for it.

Way of thinking – What are the essential questions?

❑ What is a realistic solution?

❑ What needs to be done to get support for this solution?

❑ How can the migration to an improved situation be accomplished?

Approach – How to find answers to these questions?

There could be all kinds of reasons why the ideal solution is not achievable. Budget limitations are a mundane but common example.

It is not always clear whether a solution is acceptable for various parties. If an important stakeholder strongly objects to the solution, it is not a good solution (even though you may find his reasons irrelevant). Acceptance can be increased by involving the right persons in the right way.

If difficult choices have to be made, they are for the client, not for you to make. But you can support the client in making the right choice by providing clear alternatives with their consequences.

Issues to think about:

- Which factors determine the success of the project?
- Which resources are available for the project?
- What is the attitude (motivation, acceptance) of the intended users?
- Which resources (funds, courses, etc.,) are available for migration?

Product – What do you write down?

Write a realistic mission statement. Desired properties that will not be realized are to be listed as exclusions.

If you think there could be problems with the migration to a new solution, it makes sense to make an outline of a migration plan.

Follow-up – What do you do with this document?

The mission statement is a formal document, to be incorporated in the requirements specification. Show it to all stakeholders (which can lead to minor changes) and make sure that it is approved by the client.

3.1 Mission statement

There are various definitions of a mission statement. Wieringa [Wie03, Ch. 5] describes the mission statement according to Yourdon. We use a slightly different format; the suggested solution need not be limited to a computer system. The system can contain people and procedures, and need not even involve a computer system.

A mission statement describes the following points

- A short motivation
- System boundary (is it a computer system, or a system that includes people around the hardware/software)
- The goal of the system (which problem will be solved)
- Exclusions (which problems will not be solved)
- How the problem will be solved

An explanation can be added as to why certain issues are (not) treated. This explanation is not part of the mission statement proper.

If different stakeholders have different interests, you could formulate alternative mission statements, and ask the client to make a choice. As stated in 2.1 a project should pursue one prime goal. Having a mission statement that is a compromise between different goals is asking for trouble later in the project.

The final version of the mission statement should be known, understood, and accepted by all important stakeholders. That doesn't mean that stakeholders agree about what they desire and what would be ideal. It means that they agree that this is the mission for this project.

Example of a mission statement

The following mission statement is taken from a recent M.Sc. project. It has five paragraphs which could be labelled: introduction / type of system / goal / exclusions / solution. The external goal is given in the first paragraph as a motivation for the project goal in the third paragraph. The system boundary is not stated explicitly, evidently(?) it is a software system.

The purpose of each paragraph is clear, so there is no need to include headers.

> *A problem to be solved in electronic commerce is the specification of terms of delivery in such a way that can it can be established beyond doubt – if necessary, in court – what these terms were at the time the contract was made. The E-Terms consortium wishes to address this problem by establishing an E-Terms repository. When a business party submits terms to the repository, the consortium guarantees that the applicable terms can be retrieved unaltered by any interested party at any future moment.*
>
> *In this project [student] will develop a prototype of an E-Terms repository.*
>
> *The purpose of the prototype is to serve as a proof of concept, aimed at showing the possibility of creating a repository and functioning as a guide for the development towards a final version. Furthermore, the prototype will be used in the external promotion of the concept to potential users, submitters and developers. It should serve both to increase the interest in the E-Terms service and to gather relevant feedback from interested parties.*
>
> *Efficiency and reliability requirements envisaged for the final product need not be met by the prototype repository.*
>
> *[Some words about the different functions to be supported by the E-terms die door de repository.]*

Step 4. Gathering requirements

The purpose of this step is to find out what people would desire the system to do, which demands they have, and which constraints there are.

Way of thinking – What are the essential questions?

- ❑ Which kind of requirements are needed?
- ❑ How and where can I find these requirements?
- ❑ Which questions do I ask?
- ❑ Could I have missed any important requirements?

Approach – How to find answers to these questions?

A common way to find requirements is to interview people. If you did that in step 1, you may already have collected some requirements. With a clear project goal and mission, it could happen that you want more specific requirements from persons you talked to earlier.

A number of other techniques are listed below. Obviously, it depends on the context and the kind of system which technique is most suitable, and which stakeholders to involve.

We make a distinction between *business-level requirements* and *system-level requirements* (elaborated below in Section 4.1) System-level requirements describe what the system should do. Business-level requirements describe which tasks should be supported by the system. Traditional software engineering has a focus on system-level requirements. However, if the main challenge is to find out how the efficiency of a task or an organisation can be improved, it could be worthwhile to focus on the business-level requirements.

Product – What do you write down?

You have written notes of all the requirements you gathered and other relevant information that people gave you.

Follow-up – What do you do with this document?

Writing an easily readable requirements specification, based on your notes, is still a lot of work. That will be the subject of step 5.

4.1 Requirements at different levels

Consider an information system for the reception desk at a hotel. It could have the following requirements:

R1. *The system shall allow the hotel to increase its bookings with 15 % without adding reception staff.*

R2. *The system will support the receptionist to prepare for the arrival of a tourist bus.*

R3. *The system shall be able to record that a room is occupied for repair in a specified period.*

R4. *The system shall record the data specified in the Class diagram in appendix X.*

We can make a distinction between business and system and between problem and solution, as illustrated in Figure 3. The requirements R1–R4 describe a business goal, business process, system requirement and system design, respectively.

	Problem	*Solution*
Business	**Goal-level requirements** *business goal*	**Business-level requirements** *business process*
Supporting System	**System-level requirements** *system requirements*	**Design-level requirements** *system design*

Figure 3 – requirements levels[2]

[2] Astute readers will have noticed a difference between figures 2 and 3. In the Z model in Figure 2, it was suggested that the requirements specification, produced in steps 4, 5, 6, provides a *solution* (bottom left corner). In Figure 3, system requirements are stated as a *problem* (bottom left corner). This paradox is caused by a

Most relevant are the business process and system requirements – assuming that the focus of your requirements specification is to make clear how a proposed system can support an envisaged business process. But we discuss each of them and give them a name for easy reference.

- **Goal-level requirements** describe a business problem, i.e., a goal that the client intends to achieve. This is an external goal (see 2.2); the supplier of the system can never guarantee that goal will be achieved, hence it is not a project goal. It could be useful to know the business goals of the client (you want the client to be happy with the delivered system), but goal-level requirements are not usually part of a requirement specification.
- **Business-level requirements**[3] describe business process: they deal with tasks to be supported by the system – without being specific about which system functions are needed to do so. The normal check-in procedure in a hotel has been designed for guests who come alone or in small groups. If a bus with several dozens of guests arrives, the reception will follow a different procedure in which the administration is done in advance, perhaps printing a list of guest names and room numbers. Which particular solution is to be chosen isn't important at this stage. The requirement in this example is that the system allows the staff to handle the exceptional situation in an appropriate manner.
- **System-level requirements**[4] specify a software problem, i.e. the desired behaviour of the system: individual functions of the system (functional requirements) and overall quality properties of the system (quality requirements).
- **Design-level requirements** specify a software solution, i.e., details about how a particular function of the system is to be implemented. These should be used sparingly in a requirements specification, it is not meant to give a detailed design of the system. But sometimes problem and solution are hard to separate. A class diagram is a good example: by specifying the object classes and their relations, it becomes clearer which information can be stored in and retrieved from the system.

In Software Engineering, the focus is on technologically challenging projects, rather than embedding the technology in an organizational context. In that tradition, software requirements are system-level requirements. In Software Engineering handbooks, finding business-level requirements is done in a separate, first phase of the software life cycle, which they call system analysis or information analysis.

In Information Systems, the biggest challenge in a project is often to make sure that a system fits the context in which it is to be deployed, rather than the technical development of the system itself. Therefore we have a broader view of requirements analysis and explicitly include the business level.

System-level requirements tell us *what* the desired properties of a system are. Business-level requirements tell us *why* a system must have certain properties.

4.2 Modeling the system vs. modeling the system's environment

Typically business-level requirements are about the system's environment, and system-level requirements about the system itself. But the system environment is not limited to the business level. Systems usually have to exchange data with other systems, which may cause requirements at the system level and even at the design level.

A requirements specification should contain a model of the environment, including other systems it has to interface with. A context diagram (see, e.g., Lauesen [Lau02, section 3.2], Wieringa [Wie03]) is a good high-level description of a system's environment.

4.3 Types of requirements

Requirements come in different types. In a requirements specification you may find the following categories:

- **Constraints.** These are global requirements that restrict the way you produce the product. Budget and delivery deadline are constraints. There can also be technical constraints, e.g. that the system should run on particular hardware or interface with an existing legacy system. Usually you are not at liberty to negotiate changes to constraints.
- **Data requirements.** A requirements specification could have a data model, specifying the kind of data that have to be stored in the system, e.g. in the form of a UML class diagram.
- **Functional requirements.** These describe the functions of the system. This can be on the system level or on the business level. In the latter case, functional requirements describe the tasks to be supported by the system.

difference in level of abstraction. Figure 2 takes the perspective of the first iteration in the requirements life cycle, steps 1, 2, 3. Defining how the system will behave is, at that stage, a solution to the real-world problem that needs to be solved. Figure 3 is takes the perspective of later iterations of the life cycle: the requirements are regarded as a problem statement, the solution is realizing a system that meets these requirements. Problem and solution are not absolute categories: some person's solution is another person's problem. A solution at a higher level is a problem at a lower level.

[3] Lauesen [Lau02] calls these "domain-level requirements," another term often found in the literature is "user requirements."

[4] Lauesen [Lau02] calls these "product-level requirements."

- **Quality requirements,** also called **non-functional requirements.** These describe quality properties of the system as a whole, see 4.4 below. Not all properties are relevant for each system.

Many examples of these types of requirements are given by Lauesen [Lau02].

4.4 Quality factors

Different sources give different classications for quality factors, but they usually overlap. ISO 9126 distinguishes

- Functionality (accuracy, security, interoperability, suitability, compliance)
- Reliability (maturity, fault tolerance, recoverability)
- Usability
- Efficiency
- Maintainability (testability, changeability, analyzability, stability)
- Portability (adaptability, installability, conformance, replaceability)

For large, safety-critical systems there could be requirments for all the second-level quality factors mentioned in parentheses. Probably you need to address only the main categories.

Usually there are trade-offs between quality factors. Increasing the security may decrease the usability of the system, and reversed.

In the initial stages of requirements elicitation, it is very difficult to get measurable quality requirements. What you really want to know, initially, is the relative importance of various quality factors for the project you're working for. Is security a really big issue, or is it only marginally relevant? If the system would be down for half a day, what would be the consequences for the customer?

For quality factors that really matter, you should try, later on, to get measurable requirements – see 4.7: fit criteria – otherwise there is no way of knowing whether the system, when it is delivered, meets the requirements.

4.5 Priorities

In the process of requirements gathering, you want to get an idea how important the various requirements are. It is possible that not all the demands and desires can be fulfilled, so it useful to know what could eventually be dropped. At a later stage (step 6), when there is a complete list of requirements, priorities can be ranked and negotiated, if necessary. At this stage, you want a first indication.

MoSCoW

For a rough indication you can use the so-called MoSCoW classification:

- **Must**: essential requirements, the system must meet these
- **Should**: requirements that the system should meet, if possible
- **Could**: nice features, that could be included if it doesn't take too much time and effort
- **Won't**: exclusions, i.e., features that some stakeholders would consider reasonable requirements, but, for some reason or other, will not be included in the system

Customer satisfaction and dissatisfaction

The Volere method [RR99] suggests estimating, on a scale of 1 to 5, customer satisfaction and dissatisfaction.

- **Customer satisfaction** is a measure of how happy the client will be if you successfully deliver an implementation of the requirement.
- **Customer dissatisfaction** is a measure of how unhappy the client will be if you do not successfully deliver this requirement.

Customer satisfaction and dissatisfaction need not be each other's inverse. For example: a very nice feature in the "could" category could make the client really happy (satisfaction = 5), but, since it's not necessary for solving the essential problem, he is not going to be deeply disappointed if it doesn't materialize (dissatisfaction = 3). Another example: If a system is supposed to be online 24/7, availability is taken for granted (satisfaction = 3), but poor availability is problematic (dissatisfaction 5).

It is generally a good idea to ask customers for (dis)satisfaction rates.

4.6 The Requirements Shell

In the Volere method [RR99], Suzanne and James Robertson give a template to be filled in for each requirement. They call it the Requirements Shell. It is suggested that you carry cardboard copies of the template with you when go around gathering requirements. See Appendix C.

4.7 Fit criteria

The Volere Requirements Shell template makes a distinction between the *description* of a requirement (what you want) and the *fit criterion* (how to determine whether what you want has been achieved). A requirement with a fit criterion is measurable: there is a way to determine objectively whether the requirement is satisfied by a given product.

For data and functional requirements this is not too difficult; if the requirement is complete and unambiguous there is no room for discussion whether a particular solution does or does not satisfy the requirement.

Quality requirements are usually harder in this respect. You may have gathered some requirements that have a description but as yet no fit criterion. E.g.

The system must respond [...] fast.

This is a clear desire, but not measurable. A fit criterion should tell precisely how fast.

The system must respond [...] within 2 seconds

is clear enough. However, is it necessary to guarantee that *all* responses are within 2 seconds and, say, 2.2 seconds during peak load is not acceptable, even if this would greatly increase the cost of the system?

A typical form for such a requirement is

The system must respond [...] within 2 seconds in 90 % of the cases and always within 5 seconds.

This is a usual form for such requirements and the fit criterion is okay. Yet, you could ask yourself wether these values are arbitrary (in which case other values can be negotiated if these would cause problems) or derived from some specific purpose. *Rationale* is another slot in the requirements shell. If you get to know why response time is an issue, but the proper values cannot be estimated right now, you should at least capture the rationale, e.g.

The system must respond [...] not slower than comparable systems.

This has no proper fit criterion yet, because it isn't defined what comparable systems are, but for the time being it expresses appropriately what is desired.

Another possibility is to give a template for a fit criterion and leave it to the system provider/designer to suggest a reasonable value:

The system must respond [...] within __ seconds in __% of the cases and always within __ seconds.

For a response time requirement we know at least that *time* is the dimension in which a fit criterion has to be specified. For some other quality requirements there is not even an obvious choice for the dimension in which quality can be measured.

Usability

Usability is one of the hardest things to quantify. Lauesen [Lau02, Chapter 6.7] gives 9 different ways to specify measurable usability requirements. Some examples:

U1*. Novice users shall perform tasks Q and R in 15 minutes. Experienced users complete tasks Q, R, and S in 2 minutes.*

U2. *80 % of the users shall find the system easy to learn. 60 % shall recomment it to others.*

U3. *Three prototype versions shall be made and usability tested during design.*

4.8 Requirements elicitation vs. requirements creation

Finding requirements is traditionally called "elicitation", which means "uncovering". Implicitly it is assumed that there are some objective needs, and it is the task of the requirements engineer to find out what those needs are. Gause and Weinberg [GW89] made clear that in most cases requirements are not elicited but *created*. The customer usually hasn't thought about the details, and the requirements analysis process may help him to explore possibilities and/or force him to decide what he wants.

In requirements *elicitation,* you are like a scientist studying the behaviour of planets: you observe what happens but you do not influence it. Requirements elicitation is simply writing down the requirements as they are told to you by stakeholders. In requirements *creation,* on the other hand, you work with the customer to identify the requirements. You join the customer in the search for goals to achieve and problems to solve. In the first case, the customer knows what the requirements are and you help him or her to write these down. In the second case, the customer does not know what the requirements are and you work with him or her to determine what they are. Requirements elicitation in its pure form does not exist.

4.9 Techniques for requirements gathering

Common techniques include (but are not limited to) the following. Lauesen [Lau02] gives some more details. Appendix B gives a longer list with further references.

- **Interviewing.** (See step 1)
- **Documents.** If the purpose of a project is to replace an existing system, the documentation of that system can give useful information, e.g. data models. Also, if you studied documents in step 1, for finding the goals and background of the project, these may hint to requirements. It is always useful to cross-check what you read in documents with what you hear in interviews.
  In an IT-intensive organisation there could be architecture documents with guidelines and constraints for individual applications.
- **Observation.** The way people work is not necessarily the same as the way people *think* they work or the way they *describe* how they work. To be a good observer, you need some skills (not taught in our courses). See Beyer & Holtzblatt [BH98].
- **Brainstorming.** You should have experience with brainstorms if you want to moderate one.
- **Focus groups.** In a focus group, representatives of different stakeholder groups come together to identify problems, needs and possible solutions. Lauesen [Lau02, section 8.4] describes how to organize focus groups.
  If you get people to attend a focus group, they are motivated to discuss problems, requirements, and solutions, and you should allow for that. You cannot limit a focus group to a single step of our life cycle, but you can emphasise one step of our

life cycle. There is always some overlap with other steps.

- **Prototyping.** Prototypes can help to imagine what the system could be like and thus to be more concrete about what they (don't) want. A prototype is typically a mock-up, in which the functionality is faked or absent. For a very first impression, a sketch on paper will do as well.
- **Study similar companies.**

4.10 Requirements elicitation for custom-tailored or COTS systems

Most requirements analysis methods deal with the case that a new system has to be developed, for which requirements need to be drawn up. In many cases, however, there is no need to develop a new system – you can buy one. Software that you can readily buy is called common off-the-shelf (COTS) software.

When a COTS solution is sought, some steps in the requirements process differ from our general outline.

Another possibility is that a commercial system is bought, but more work (fine-tuning, interfacing with other systems) needs to be done in the operation environment. This is typically the case with ERP systems. If there is a choice of different suppliers, this would call for a tender process.

After the project goals and mission are clear, some alternatives to are

- **Tender process**. You draw up a requirements specification of what is needed, and ask different vendors whether they can supply this, and at what price.
- **COTS selection.** If different companies sell software packages with the same kind of service, you have to select which one is the most suitable. Chances are that the functionality of these packages is rather similar (if they wouldn't satisfy the *market requirements*, i.e. the functions that such a package ought to have, they wouldn't be in business). There is usually more difference in quality issues (e.g. how good is their service?). Hence the selection should pay due attention to these.

If your client is a vendor of COTS software, some of the items in these guidelines have to be reinterpreted accordingly. It is important as ever that the product satisfies the customer. The client will be satisfied if the customer wants to buy it, but he is not the most authoritative source for the customer's desires. See Cooper [Coo99] for learning the user's desires in COTS software production.

Step 5. Writing a requirements specification

The purpose of this step is to write a draft version of the requirements specification. Some requirements may change, as a result of discussing the draft with relevant persons – but in order to engage in such discussions, you need a good document.

There are a number of different things to consider when you write the first full version of your requirements specification. This section is split into four subsections, treating separate concerns:

5.1 What should be the contents of a requirements specification,

5.2 Specification techniques,

5.3 Readability and linguistic issues,

5.4 Quality check.

Product – What do you write down?

A complete, well-structured, readable requirements specification.

Follow-up – What do you do with this document?

Send this document to relevant stakeholders. You may ask them for written comments or discuss the document with them. The latter is more work but yields better results.

5.1. Contents of a requirements specification

Way of thinking – What are the essential questions?

❑ Which subjects should be covered in the requirements specification?

❑ How to structure the requirements specification?

Approach – How to find answers to these questions?

In addition to a list of requirements, a requirements specification gives some information about the reasons for the project, the context of the system, and any other issue for which you find it relevant to provide written details. Examples of real requirements specifications are given by Lauesen's [Lau02], Chapters 11–15. A detailed, generic table of contents for a requirements specification from the Volere method [RR99] is given in Appendix D. You can use it as a checklist of things you'd like to discuss in your requirements specification. You don't want to cover all of these (unless you're doing requirements for a multi-million Euro project), so you should think about what is relevant for your project.

5.1.1 Free form or template?

Some organizations that do a lot of software projects have their own template for requirements specifications, with a fixed table of contents. Using such a standardized format has the advantage that it is easier to find particular pieces of information (if both the writer and the reader are familiar with the standard). The disadvantage is that the prescribed table of contents is probably not the most suitable for the particular project you're working on. Kovitz [Kov98] advocates the principle "form follows content." If you know what you want to say, then choose the structure that is best suited to express what you want to say.

5.2. Specification techniques

Way of thinking – What are the essential questions?

- ❑ Which parts/aspects of the environment and the desired solution need to be specified in some detail?
- ❑ What is the most appropriate specification technique in this context?

Approach – How to find answers to these questions?

Diagrams are more precise and less ambiguous than words. It is not uncommon to include use case diagrams in the functional requirements and to use a class diagram for specifying data requirements for a system. It could make sense to use an entity-relationship diagram to specify the *environment* of the system and a context-diagram to specify the interaction of the system with its environment.

What is useful depends on the project – and to a certain extent on the requirements analyst. Techniques you are familiar with work better (if they are appropriate) than techniques you have never used before. The courses Information Systems (212010) and Requirements Engineering (232081) provide enough technical background for bachelor students. Master students Business Information Technology could also apply techniques from Specification of Information Systems (233030).

5.3. Readability and linguistic issues

Way of thinking – What are the essential questions?

- ❑ Who is my target audience? Can they understand it?
- ❑ Can the presentation be improved?
- ❑ Can the text be shortened?

Approach – How to find answers to these questions?

The purpose of the document you are writing is to communicate its contents to other interested parties. In order to achieve that purpose, it pays off to make an effort to make the document well-written and well-structured. Unfortunately, the form that is easiest accessible for the target audience is not the easiest one to write. Some tips are given below.

5.3.1 Keep it short[5]

Many requirements specifications are longer than necessary. This has several disadvantages. Firstly, the readers may not read the whole document. If it's long, people are inclined to browse through the document, rather than read it. Secondly, it is more difficult to find back some piece of text. This makes it harder to use it as a reference. Thirdly, a longer text is more difficult to comprehend than a short one. Unfortunately, writing a short text is more difficult than writing a long text.

Repetition

A prime way to make a text longer than needed is to repeat information. Occasionally it is useful, to repeat text, e.g. when you give an overview or an example. Most other repetitions are not needed and can be discarded.

Metatext

Metatext is text about the text. Again, in some cases this is useful. It makes sense, for example, to explain the structure of the document in the introduction. A typical example of superfluous metatext: "In this chapter the user interface requirements are given" as introductory statement in a chapter "User interface requirements".

[5] Sections 5.3.1-3 are primarily based on Kovitz [Kov98] and translated from a version in Dutch compiled by Emile de Maat.

Generalities

Generalities are pieces of text that are not specific for the requirements that you are writing, but are more generally applicable. Consider, for example, a requirement

> *Each input screen shall fit entirely within the window and shall use as little scrolling as possible to display and/or retrieve information.*

A good user interface designers knows this and will try to apply it. A requirements specification is not a proper place to teach others about good user interface design.

Useless additions

Sometimes authors add extra text that carries no additional information. They do so, apparently, for fear of short texts – perhaps they are afraid that somebody will judge these texts as insufficient because they are short. For example:

> *The system should be user-friendly and have a simple user interface*

The second part is redundant.

Another useless addition is upgrading a short piece of text to a separate section. E.g.

> *3.3 Performance*
> *Downtime should be limited to one day per year.*

If this is all there is in Section 3.3, it could have been merged with another section.

The use of a template with a standard table of contents leads to sections like 3.3 above or, even worse,

> *3.4 Hardware constraints*
> *There are no hardware constraints*

5.3.2 Keep it simple

Requirements specifications often are hard to understand. Usually this is not because the requirements are inherently complicated, they are just specified in a complicated way. We discuss some causes for this.

Use short sentences

Many authors write too long sentences. This is often caused by the desire to provide complete and precise information. It is good to be aware, however, that all this information does not have to be captured in a single sentence. Long sentences can be made more understandable by dividing them into smaller sentences. For example

> *In this document the requirements are given for a system that Wertor will design for Myriad.*

This not a really complicated sentence. But it could be replaced by

> *Wertor will design a system for Myriad. This document gives the requirements for this system.*

Use clear and consistent terminology

When you elicit requirements, different persons may use different terms to describe the same concept. This can easily be carried over into the requirements specification, but it is confusing for the reader. It pays to make the extra effort to ensure consistent terminology. Make a glossary and make sure that the text is consistent with your glossary.

Also, the author may use a term that is known to his professional colleagues (or even worse, invent a new term) but not understood by the readers of the document. If you *must* use an unfamiliar term, make sure that you define it.

Avoid overspecification

Requirements should be complete and unambiguous. This is generally true, but it can be carried too far. Consider the following requirement for an inventory system

> *Every object in the store that is meant for sale has a unique identification code*

The store contains objects that are not for sale: shelves, fork-lift trucks, etc. These do not need a unique ID in the inventory system, but in the domain of inventory systems that is quite obvious. Hence the following, easier requirement will do

> *Every object in the store has a unique identification code*

5.3.3 Structuring text

The structure of a document can contribute a lot to its readability. Structure tells the reader what to expect where, and helps him understanding the text. In a well-structured document it is easy to find back pieces of information. This makes it suitable as a reference document.

Structuring a document is done in three steps

1. Make a list of all subjects to be treated
2. Group these into coherent groups
3. Decide upon an order in which to present them.

Most difficult is step 2. There are different ways to group subjects, and usually each of them poses some problem for presenting them in a linear order. Choose the grouping that seems most suitable and solve the ordering problems by appropriate cross-references. Make sure that you always treat one subject at the time.

Examples of different structuring principles:

- Group requirements by type of requirement
- Group requirements by stakeholder
- Group requirements by subsystem.
- Group requirements by priority, first state the “must”, then the “should”
- Order the subjects from general to specific
- Order the subjects from important to unimportant

- Order the subjects from easy to difficult, so that the reader can increase his understanding along the way.
- and so on ...

Whatever structure you choose, it is important that you support it in text and lay-out.

5.3.4 Presenting information

Whatever specification techniques you have used, there will be a lot of natural language in the document. If this contains factual information, it is advisable to present this in the form of lists and tables. Lists offer more structure, and people can use them as checklists.

A table is in fact a two-dimensional list. Information suitable for a table is hard to present in flat text. Tables are easier to read, but also easier to write.

5.4. Quality check

Way of thinking – What are the essential questions?

- ❑ Are all requirements unambiguous and complete?
- ❑ Is there a fit criterion for each requirement?
- ❑ Do we know for each requirement why it is in the specification?
- ❑ Are there conflicting requirements?
- ❑ Is the document as a whole properly finished?

Approach – How to find answers to these questions?

Below you find some quality criteria that should be applied to each requirement to determine whether it is a good requirement.

Finally, before you deliver the document, make sure that there are no loose ends, that cross-references are correct and that spelling errors, typos, and word processing errors have been eliminated.

5.4.1 Quality criteria for individual requirements

Robertson and Robertson [RR99] say that any requirement that does not satisfy all the quality criteria is, at best, a *potential* requirement. In the final version of the specification there should not be a single requirement of insufficient quality. But what we are working on here is still a draft version. For a draft version, it could make sense to include potential requirements – with an annotation of the defects yet to be solved – if these requirements were raised and should not be forgotten.

Complete?

In step 4.6 we introduced Volere's Requirements Shell [RR99], a template to be filled in for each requirement, see Appendix C. Are any components for the template not filled in? Perhaps there is nothing to fill in. For example, if there are no supporting materials, then the Shell should say "Supporting materials: None" (rather than leaving it blank). Other things might not have been clear at the time the requirement was elicited. For example, at the moment you don't know about dependencies or conflicts. Or perhaps you would need the customer to assess (dis)satisfaction values but you didn't have a chance to talk to him after the requirement was raised. It is likely that you do not yet have a fit criterion for each requirement.

If some of the questions cannot be answered right now, we have to live with that for the time being. You could indicate in the document specifically to which questions you still need answers. What you should never do is *guessing* the answers in order to complete the specification.

Precise, unambiguous and meaningful to all stakeholders?

Check whether the requirements can be misunderstood and interpreted differently from what you wanted to say.

Could possible ambiguity be reduced by stating more precisely what you mean? For example

> *"Supporting Material: Information plan of company X"*

is unambiguous only if there is a single version of this information plan. Therefore

> *"Supporting Material: Information plan of company X d.d. 22 December 2005"*

is better.

Consistent terminology (see 5.3.2) is a precondition for precise, unambiguous and meaningful requirements.

Fit criterion?

Does each requirement has a fit criterion (see 4.7), i.e. it is possible, when the system will be delivered, to establish objectively whether the requirement has been satisfied?

Relevant to the system's purpose?

Sometimes people get great ideas about what a system could also do. In the mission statement we have clearly laid down the purpose of the system. If a requirement does not contribute to the purpose, it is in the nice-to-have ("could") category. If it is included in the requirements specification, it must be made clear that it is not an essential requirement.

Unnecessary requirements are typically those with high customer satisfaction rating and low customer dissatisfaction rating.

Viable within constraints?

Does the project have the time and budget to satisfy the requirement? If not, it's not a good requirement, and should be discarded. (Or the time and budget should be adapted. If neither is acceptable the project should probably be abandoned!)

5.4.2 Consistency across requirements

In 5.4.1 and 5.4.2 we have scrutinized each requirement individually. Similar questions can be asked about the whole set of requirements. There could be

- redundant requirements;
- incompatible requirements (i.e. it is not possible to satisfy all at the same time);
- missing requirements.

Obviously there is no fail-safe way to discover missing requirements. An important way to get these is to get feedback from relevant stakeholders on the draft requirements specfication (see 6.1, validation). However, there are some consistency checks that you can do before the draft specification is finalized.

All tasks / use cases covered?

If there are task descriptions or use cases for the system, check that all actions have been covered.

System administration and support covered?

Most computer systems offer two kinds of functions: primary functions that serve the purpose of the system (users can do something useful) and secondary functions to allow the system to be operated (e.g. adding new users, maintaining the system's data). Are these secondary functions covered?

CRUD check

If there is a data model, check whether each attribute is Created and Read, and, if applicable, can be Updated and Deleted.

5.4.3 Have you finalized the document?

There are various natural roles that people can have when they work in a team (called *Belbin roles*, after the person who discovered them). Experience shows that the role *completer/finisher* is poorly represented among our students. Before submitting a document, such a person would scrutinize every detail to make sure that

- everything is numbered correctly,
- cross-references are correct,
- figures and tables appear in the right place,
- citations and references are marked appropriately in the text
- literature references in the reference section are complete,
- the lay-out is consistent,
- the names of the author(s) and other contributors are mentioned appropriately, and
- the document carries the right date and version number

If you have no such person on your team, or if you are working alone, you should force yourself to do this. This gives the document a professional appearance.

Step 6. Validating a requirements spec

The purpose of this step it to ensure that a solution that satisfies the requirement specification achieves the goals laid down in the mission statement.

Way of thinking – What are the essential questions?

- ❑ Does the specification reflect the desires and needs of the stakeholders?
- ❑ Do the stakeholders agree on the priorities, when there are conflicting requirements or when not all requirements can be met?
- ❑ Is it technically possible to meet the requirements?
- ❑ Which requirements have not passed the quality test?

Approach – How to find answers to these questions?

Validation means that you make sure that you have specified the right solution, i.e. that a product satisfying these requirements will meet the goal that was laid down in the mission statement. The persons who can decide that are the stakeholders, not the requirements analyst. (And In order to decide that, they have to be able to understand the draft specification – that is why we spent so much effort on step 5).

In situations where a complex and technically challenging system is proposed, it is wise to consult the software architects who will be involved in the design. The can warn you about requirements that are hard or impossible to realize.

If there are conflicting requirements, or if not all the requirements can be met, tough decisions have to be made. There are two things you can do: engage some stakeholders in ranking essential requirements according to importance, or ask the client to decide (or one after the other).

At the same time, when you are going back to the stakeholders with the draft requirements specification, this could be an opportunity to elicit missing elements of the requirements shell, e.g. fit criteria. You can put gentle pressure on them by explaining that, ultimately, an incomplete requirement cannot be included in the final specification.

Product –What do you write down?

The final version of the requirements specification.

Follow-up – what do you do with this document?

Deliver the specification. The requirements analysis has been completed.

6.1 Requirements validation

There are several way in which you can get feetback on the draft requirements specification. You can circulate the specification to the stakeholders and discuss it with each stakeholder individually, or you can organise a validation meeting.

If you want to know what people really think about the requirements specification, you must make sure that they understand it. That is why it is worthwhile to make the draft spec a complete, legible, accessible document, rather than circulating a premature version.

If a prototype was made for requirements gathering, you could show (an updated version of) the prototype in addition to the specification document.

Validation meeting

At a validation meeting, a selection of relevant stakeholders is present. The participants at this meeting must have enough knowledge of the application domain and the context in which the system is going to be used (the organisation for which the system is developed). Also at least one end user must be present.

The purpose of a validation meeting is to draw up a list of problems with the requirements specification, and possibly an agreed list of actions to address these problems. (It is *not* the purpose of the meeting to solve the problems here and now).

See Kotonya and Sommerville [KS89, Chapter 4] for a more elaborate description of validation meetings.

6.2 Requirements prioritization

Sometimes it's impossible to satisfy all the requirements. A finite budget is the most mundane and the most frequent reason to scale down your desires. But it could be the case that requirements are at odds with each other. Higher security may imply lower user-friendliness, and reversed. Also, if you buy an existing system or a COTS product, you have to choose from what is available, which may not be exactly what you want.

Section 4.5 discussed the MoSCoW classification and customer (dis)satisfaction values. These are absolute values, to give a first indication of what is important. When it comes to making tough decisions – what to discard, or to postpone to a future release – absolute values aren't good enough (usually too many things are important).

What is needed, then, is to assign priorities. These are relative values: is a requirement A more important or less important than requirement B?

In order to reach an optimal decision, one should

- establish relative values for all requirements
- estimate the cost of implementing the requirement

A formal method for this, based on the Analytic Hierarchy Process (AHP), is presented by Karlsson and Ryan [KR97].

Such a method yields an optimal decision, if the costs estimates are accurate and if there is no disagreement among the stakeholders (or if only the prime stakeholder, the client, matters).

When different stakeholders with different desires are important to a project, there is a political element in prioritizing requirements. When some get all their priorities granted, and others get none, the project is in for trouble.

Informal ways to assign priorities include

- Ask persons to assign a total of 100 points to different requirements in any way they want. (could be done by different stakeholder representatives as a starting point for a meeting to decide the priorities)
- Get a meeting of stakeholder representatives to agree on the 10 most important requirements. (If politics are really troublesome this could be done without further ranking among the top 10).

Step 7. Maintaining the requirements specification

The purpose of this step is to ensure that in all steps of the system's life cycle there is an accurate requirements specification for the current version of the system.

Way of thinking – what are the essential questions?

- ❑ How do you manage new requirements that arise during system development?
- ❑ How do you maintain requirements traceability and keep the requirements specification consistent when requirements change?

Approach – How to find answers to these questions?

In nearly all cases where students do a requirements analysis, the students are no longer involved in the later stages of project development. Chances are that you will not be asked to maintain the requirements specification that you delivered. Nevertheless we briefly mention some issues, completing the requirements specification life cycle.

7.1 Requirements evolution

In the ideal case, all stakeholders agree that your final requirements specification accurately descibes their requirements for the new system – at this moment. There are many reasons why requirements may change in the future:

- Testing and operation of the system may reveal defects. That is, some essential requirements were missed after all.
- Stakeholders may come up with new desires for additional features.
- The world changes, which my lead to new business requirements, or may require the system to interact with new (versions of) systems in its environment

While the system is still under development, some care should be taken in allowing new requirements to come up. *Goldplating* is a well-known software engineering risk: additional requirements continue to be added, where each requirement in itself may seem harmless, but the overall result is that it becomes impossible to build the system on time and within budget. A related risk is *feature creep*: at little extra effort (so it seems) a function can be added that would be nice to have. This may lead to a system with more capabilities than required – but at a later date and with a higher cost.

On the other hand, errors will be found and unforeseen circumstances may demand new requirements. In order to balance these concerns, any large project will have an explicit procedure for handling change requests.

7.2 Traceability

Traceability supports the maintenance of a system. The (evolving) requirements specification should on the one hand reflect the business needs and stakeholders' demands, and on the other hand specify the system's behaviour. This leads to 4 traceability relations:

- *From business/stakeholders to requirements*: It should be verified that the business goals of the system are covered. Essentially, the requirements should enable the mission statement (see 3.1) to be fulfilled.
- *From requirements to business/stakeholders*: For each requirement, there should be a business reason why the requirement is included in the specification (otherwise the requirement should be deleted).
- *From requirements to system:* For each requirement it should be known which pieces of code / parts of the system make sure that the requirement is satisfied
- *From system to requirements:* For each piece of code / part of the system it should be clear which requirements depend on it. (otherwise, it serves no purpose).

Hence, if a change is proposed, it can be easily determined which parts of the system are affected and what the effort will be to implement the change.

For any sizeable project, a specialzed tool, e.g. DOORS[6], is needed for implementing traceability. Currently, traceability is not used a lot in practice, because it brings additional cost in the development phase, whereas most of the savings take place in the maintenance phase. (Note that on average maintenance accounts for 70 % of the total software life cycle costs). However, in future it may become a standard practice in software engineering, due to new legislation. Quality standards like the higher CMM levels enforce traceability.

[6] http://www.telelogic.com/corp/products/doors/

Glossary[7]

Client. The person who pays for the development of the system. (see also *customer*)

Constraint. A global requirement that restricts the way the system can be produced. The project budget is an example of a constraint. Usually constraint are not subject to negotiation.

Customer. The person who buys the system. This could be the same as the client. If the product is to be sold, the customer and client are different.

Data requirement. A specification of the kind of data and the relation between data elements to be stored in the system.

Business-level requirement. A description of a task to be supported by the system, without specifying what exactly the system will do.

External goal or **Client's goal.** Something the client hopes to achieve as a result of the project. The project carries no responsibility for an external goal. Nevertheless, if the external goal will not be achieved, the client may consider the project a failure. (see also *project goal*)

Fit criterion. A quantification or measurement of a requirement such that it is possible to determine whether a system satisfies this requirement.

Functional requirement. Something that the system must do, a description of the behaviour of a system

Goal. See *external goal* and *project goal*.

Migration. The path of change leading from the current situation to a new situation, in which a new system is deployed and effectively used.

Problem. A difference between what is experienced and what is desired.

Project. Throughout the text it is assumed that there is a project to deliver some system, and you are doing the requirements analysis for this project.

Project goal. Something that should be realized by the project (and for which the project manager can be held responsible). (see also *external goal*)

System-level requirement. A desired property of the system. In previous times, *requirements* was considered to be equivalent with *system-level requirements*.

Quality requirement. An overall property of the system, describing how well the system performs its functions.

Requirement. See *contraint*, *data requirement*, *functional requirement*, *quality requirement*.

Requirements process. The part of system development in which people attempt to discover what is desired.

Solution. A way to reduce a problem.

Stakeholder. Someone who gains or loses something (could be functionality,revenue, status, compliance with rules, and so on) as a result of that project.

[7] Some definitions are taken directly from other sources ([AR04], [GW89], [Lau02], [RR99]). References are given where a term is introduced in the text.

References

[BCN92] C. Batini. S. Ceri and S.B. Navathe (1992). *Database Design: An Entity-Relationship Approach.* Benjamin/Cummings.

[Ale03] I. Alexander. Stakeholders – Who is Your System For? http://easyweb.easynet.co.uk/%7Eiany/consultancy/stakeholders/stakeholders.htm

[AR04] I. Alexander, S. Robertson (2004). Understanding Project Sociology by Modeling Stakeholders. *IEEE Software*, January/February 2004.

[Cro89] N. Cross. (1989). *Engineering Design Methods.* Wiley, Chichester, UK.

[Che81] P.B. Checkland (1981). *Systems Thinking, Systems Practice.* Wiley.

[BH98] H. Beyer and K. Holtzblatt (1998)*. Contextual design: Defining Customer-Centered Systems.* Morgan Kaufmann.

[Coo99] A. Cooper (1999). *The inmates are running the asylum*. Macmillan Computer Publishing, Indianapolis, IN.

[GW89] D.C. Gause, G.M. Weinberg (1989). *Exploring Requirements: Quality Before Design*. Dorset House, New York, NY.

[HC88] J.R. Hause, D. Clausing (1988). The house of quality. *Harvard Business Review,* 66(3), 63–73.

[KK92] K.E. Kendall and J.E. Kendall (1992). *Systems Analysis and Design*. Second edition.Prentice-Hall.

[KR97] J. Karlsson, K. Ryan. A Cost-Value Approach for Prioritizing Requirements. *IEEE Software* 14(5), 67–74.

[KS98] G .Kotonya and I. Sommerville (1998). *Requirements Engineering.* Wiley, Chichester, UK.

[Kov98] B.L. Kovitz (1999). *Practical Software Requirements: A Manual of Content and Style*. Manning Publications, Greenwich, CT.

[Lau98] S. Lauesen (1988).*Software Requirements: Styles and Techniques*. Samfundslitteratur, Frederiksberg, Denemarken.

[Lau02] S. Lauesen (2002). *Software Requirements: Styles and Techniques*. Addison-Wesley, Harlow, UK.

[Lun81] M. Lundeberg, G. Goldkuhl and A. Nilsson (1981)*. Information Systems Development: A Systematic Approach*. Prentice-Hall, Englewood Cliffs, NJ.

[Mac96] L. Macaulay (1996). *Requirements Engineering*. Springer Verlag, New York, NY.

[RE95] N.F.M. Roozenburg and J. Eekels (1995) *Product design: Fundamentals and Methods.* Wiley, Chichester, UK.

[Ret94] M. Rettig (1994). Prototyping for tiny fingers. *Communications of the ACM*, 37(4), 21–27.

[RR99] S. Robertson, J. Robertson (1999). Mastering the Requirements Process. Addison-Wesley, Harlow, UK.

[Wie03] R.J. Wieringa (2003). *Design Methods for Reactive Systems: Youdon, Statemate, and the UML*. Morgan Kaufmann Publishers, San Francisco, CA.

Appendix A. Context-free questions

When you first enter an organization for which you are to do requirements work you may be overwhelmed by the number of potentially relevant people, departments, systems, goals and problems. This appendix lists some simple questions that you can always start with. They are called "context-free" because they apply to all kinds of problems, independent of the particular problem context. The following list is largely from Gause and Weinberg [GW89]. The problem identification and analysis questions are from ISAC [Lun81].

The business

- What kind of business is this?
- What is the structure of the business?
- Which departments of the business are involved in the system?
- What are the mission and goals of the business and its relevant departments?
- Are there any related projects?

Problems

- What are the problems?
- For each problem:
  - What is the real reason for wanting to solve this problem?
  - Can a solution to this problem be obtained elsewhere?
  - Which organizational goal is served by solving this problem?
  - How bad is the problem? (Quantify if possible)
  - How urgent is it?

Stakeholders

- Who are the stakeholders?
- For each stakeholder:
  - What is his/her relation to the system?
  - What are the responsibility relations between the stakeholders?
  - Who is responsible for improving the system?
  - Is management committed to improving the system?

Problem analysis

- Which stakeholders have which problems?
- For each stakeholder/problem combination:
  - How much is it worth to this stakeholder to solve the problem?
  - How bad is it for the stakeholder if the problem is not solved?
  - How urgently should this problem be solved?
  - How bad is it if this problem is solved one year later?
  - What is the trade-off between time and value?

The current system

- Who is using the current system and in support of which business activity?
- What problems are solved by the current system? For whom?
- What problems are introduced by the current system? For whom?
- Does the system fit into the business strategy?
- Is the system mission-critical?
  - How bad is it if the system breaks down?
- Does the system interface with legacy systems?

Appendix B. Requirements elicitation techniques

During requirements work, you must find the goals, desires and wishes of the stakeholders. This appendix lists some techniques that you can use for this.

It is important to distinguish requirements elicitation from requirements creation.

Finding out about current environment and its goals, and about the current system.

The following techniques are useful for fact-finding. They are closer to elicitation than to creation.

- **Interviews.** Asking stakeholders what they currently do and how they would like to change this. Kendall and Kendall [KK92] give a useful introduction to interview techniques for information analysis.
- **Observation** of current work. Observing what stakeholders actually do, as opposed to what they *say* they do. Beyer and Holtzblatt [BH98] give an excellent survey of models to make when observing stakeholders at work (models of flow, sequence, artifacts, culture and the physical situation), how to make them and how to create requirements from them.
- **Participation** in current work to actually experience what the current environment does. There is no literature on this: Just join the stakeholders in doing their work. Take your time doing this.
- **Questionnaires.** Sending out forms with questions to stakeholders about the current environment. Kendall and Kendall [KK92] give a useful introduction to the construction of questionnaires for information analysis.
- Study **current system documentation.** There is no literature on this. Brace yourself to digest a mountain of information.
- Study **current forms** (paper forms, screen forms). Analyzing forms in use by the current system to discover data structures and work procedures hidden in them. Batini, Ceri and Navathe [BCN92] give a useful introduction to uncovering data structures from forms.

Problem Analysis

The following techniques help you to analyze problems identified during fact-finding.

- **Soft Systems Methodology (SSM).** A method defined by Checkland [Che81] to analyze exceptionally vague problems (problems where the problem is that the problem is not known). Macaulay [Mac96] gives a handy introduction.
- **Stakeholder analysis.** Set off stakeholders against problems and analyze each problem on severity (quantify!) and urgency. Gause and Weinberg [GW89] give useful hints.

Creating requirements for new system

The following techniques can be used to create new ideas about possible solutions to problems.

- **Brainstorm.** Generating wild ideas in a group without criticizing any idea, followed by a rationalization of the ideas. Roozenburg and J. Eekels [RE95] give a very useful introduction to brainstorming for product design, including its variations, such as brainwriting (in which participants anonymously submit their ideas in writing).
- **Focus groups.** Let a group of users discuss requirements with each other. Macaulay [Mac96] gives a short introduction to the use of focus groups for requirements engineering.
- **JAD workshops.** Bring stakeholders from the customer and developer sides together and let them jointly do the design. Macaulay [Mac96] gives a short introduction to the use of JAD workshops for requirements engineering.
- **Visiting similar companies.** Visit companies with similar problems to get an idea about the desirable properties of solutions to these problems.
- **Quality Function Deployment (QFD).** Maintain traceability tables that match user requirements with system requirements. Attach weights to indicate priorities, and indicate conflicts between requirements that. Discuss with all stakeholders and agree on choices based on this traceability information. Hausaer and Clausing [HC88] give a good introduction and Macaulay [Mac96] provides a very short summary.
- **Goal-means analysis.** Make a goal tree. Indicate for each requirement the goals that it serves, and indicate for each goal the desired system properties that would help reaching that goal. Lauesen [Lau98] gives an example.

Techniques for refining system requirements and corresponding environment models

The following techniques all assume that you alrerady have some idea about system requirements and allow you to improve them.

- Collecting **supplier information**. Collect documentation from suppliers, let them give demos in order to get an idea of which system requirements can actually be realized with current commercially available technology.
- **Throw-away prototypes.** Constructing a software system that implements a few of the system requirements, and letting users experiment with it to give them the occasion to form more concrete ideas about what they really want. After experimenting, the improved

requirements are written down and the prototype is thrown away. Any software engineering book contains a section about throw-away prototyping. Ince [Inc92] is one of the many overviews. Less well-known is a description of low-tech prototyping, involving pencil, paper, glue, and role playing, described by Rettig [Ret94], that in many cases is more efficient and at least as effective as high-tech prototyping.

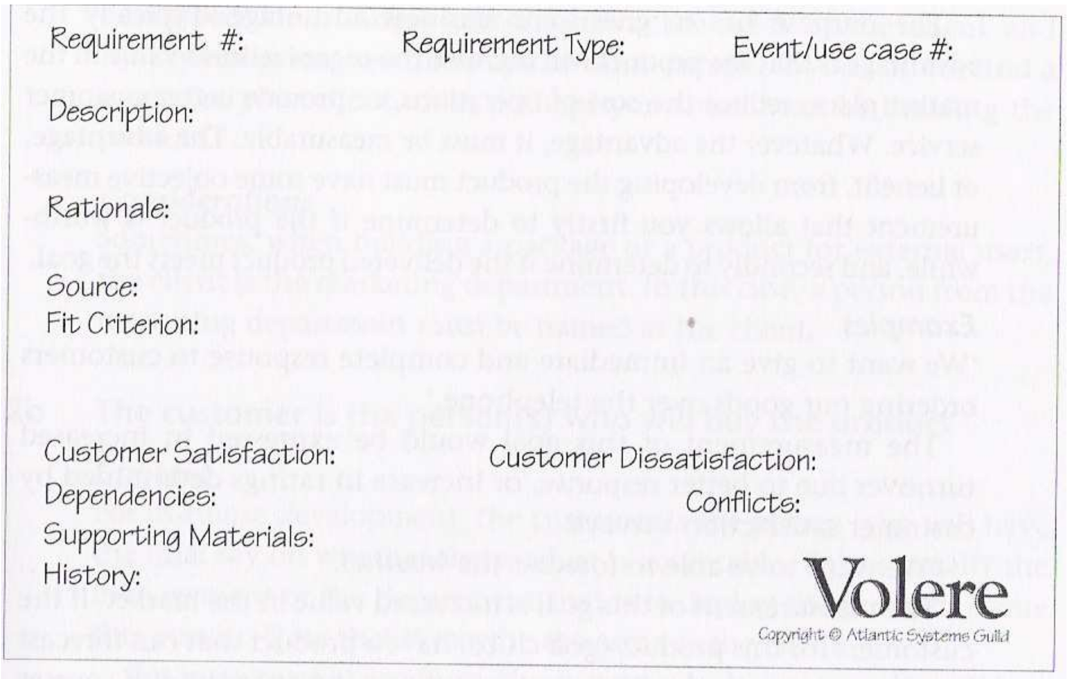
- **Pilot project.** Implement the system in a part of the organization where it is not critical, in order to get experience with real use of the system. This should lead to improved requirements.

Appendix C. Volere Requirements Shell

In the Volere method [RR99], Suzanne and James Robertson give a template to be filled in for each requirement – see Figure C1. They call it the Requirements Shell. It is suggested that you carry copies of the template with you when go around gathering requirements.

Filling in the template for each requirement reminds you of what you want to ask the person(s) you're talking with. The slots have the following purpose

- *Requirement #*: unique ID for each requirement
- *Requirement type:* constraint / data / functional / quality
  (or refer to section in requirements specification template in Appendix C)
- *Event/use case # :* If use cases or an event list has been specified, refer to its number
- *Description:* A one-sentence statement of the intention of this requirement
- *Rationale:* Why is this requirement considered important or necessary?
- *Source:* Who raised this requirement?
- *Fit criterion:* A quantification of the requirement used to determine whether the solution meets the requirement (not always easy to determine up front. If no sensible criterion can be found when the requirement is raised, we suggest to leave it open for the time being.)
- *Customer (dis)satisfaction:* Measures for the (un)happiness of the customer if this requirement is (not) implemented. See section 4.5
- *Dependencies:* Dependencies between this requirement and others.
- *Conflicts:* Requirements that contradict this one
- *Supporting Materials:* Pointer to supporting information
- *History:* Changes to this requirement (and reasons why)



Requirement #: Requirement Type: Event/use case #:

Description:

Rationale:

Source:

Fit Criterion:

Customer Satisfaction: Customer Dissatisfaction:

Dependencies: Conflicts:

Supporting Materials:

History:

Volere

Copyright © Atlantic Systems Guild

Figure C1: Volere Requirements Shell

Appendix D. Volere Requirements Specification Template

The Volere method [RR99] provides a template for the contents of a requirements specification. Here we only give the contents with some bits of explanation. An extensive description of the template can be downloaded from www.volere.co.uk. It is very thorough and complete, and for a small project there is probably no need write a requirements specification with 27 chapters. But you may use this as a checklist.

Project Drivers

1. **The purpose of the project**
2. **Client, customer and other stakeholders**. The client is the person paying for the development, and owner of the delivered product. The customer is the person buying the software. Client and customer are the same for in-house developments but different when the system to be developed will be sold to others.
3. **Users of the product**

Project Constraints

4. **Mandated constraints**. Constraints that the project must satisfy. Includes development time and budget.
5. **Naming conventions and definitions**
6. **Relevant facts and assumptions**

Functional requirements

7. **The scope of the work**. Describes the domain. Could include a context diagram.
8. **The scope of the product**. Could include use case diagram.
9. **Functional and data requirements**

Non-functional requirements

10. **Look and feel requirements**
11. **Usability and humanity requirements**
12. **Performance requirements**
13. **Operational requirements**. Expected physical environment, hardware, and software applications with which the system should interface.
14. **Maintainability and support requirements**
15. **Security requirements**
16. **Cultural and political requirements**
17. **Legal issues**

Project issues

18. **Open issues**. Issues that have been raised and do not yet have a conclusion.
19. **Off-the-shelf solutions**. Ready-made software products or components that can be used
20. **New problems**. Problems that may result from introducing the system.
21. **Tasks**. A stepwise description of system development, delivery, and implementation
22. **Cutover**. Issues related to the migration to the new system.
23. **Risks**
24. **Costs**
25. **User documentation and training**
26. **Waiting room**. Requirements that will not be part of the agreed system, but could be included in future versions.
27. Ideas for solutions

Software Engineering
Spring 2008

Michel Chaudron

Ariadi Nugroho

Outline

- **Introduction**
- **Course logistics**
- **Introductory lecture Software Engineering**
  - What is SE?
  - What does a SE do?
  - What does a SE process look like?

Introduction

Michel Chaudron

- Associate Professor in Leiden (1d) & Eindhoven (4d)
- Ph.D. students: Ariadi Nugroho (assistant) & Werner Heijstek

- M.Sc. & Ph.D. from Leiden, some time abroad
- some years with IT company
- research in software engineering:
  - software architecture and component-based sw engineering
  - quality, measurement in SE – esp. UML
- Collaborations with companies: Philips, Oce, CapGemini, LogicaCMG, KLM, Nokia, …

What you will learn?

Engineering = skill + knowledge

This course 80% knowledge and 20% skills

Basic concepts, vocabulary of Software Engineering

Main activities in SE projects

Main methods and techniques (excluding: programming)

Guest Lectures by professionals

SE as an academic research area

Book: Object-Oriented Software Engineering, Timothy C. Lethbridge, Robert Laganière (2nd Ed.)

Ch 1: introduction to the subject

Ch 2: OO-basics

Ch 4: Requirements

Ch 5 & Ch 8: Modeling using UML

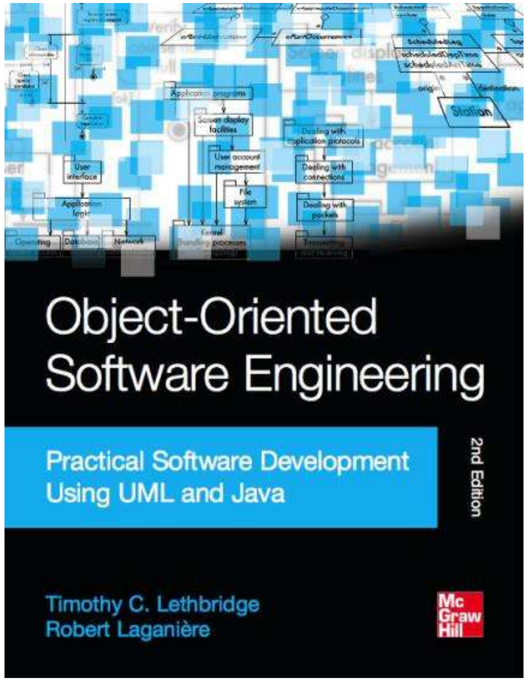
Ch 6: Design patterns

Ch 9: Architecture & Designing

Ch 10: Testing / Quality Assurance

Ch 11: Management (Estimation, Risk)

Websites: www.mhhe.com/lethbridge en www.llsoeng.com



Object-Oriented Software Engineering

Practical Software Development Using UML and Java

2nd Edition

Timothy C. Lethbridge
Robert Laganière

Mc Graw Hill

Assignment

Car Navigation System

- Requirements
- Architecture & Design
- Analysis
- Implementation (mock-up)
- Test

© Lethbridge/Laganière 2001

Lectures Schedule

Wk-nr		Datum	lecturer	onderwerp	Huiswerk/leeswerk
6	1	7 feb	Chaudron	Introduction Software Engineering	LL Ch 1, 2
7	2	14 feb	Chaudron	Requirements Engineering	LL Ch.4.
8	3	21 feb	Chaudron	Software Architecting	LL Ch. 9
9	4	28 feb	Chaudron	Modeling with UML	LL Ch 5
10	5	6 maart	Peter Bink	Cost Estimation, Planning & Control	LL Ch 11
11	6	13 maart	Chaudron	Software Metrics	LL Ch 10 & 11
12	7	20 maart	Bart Kienhuis	Design Patterns / Refactoring	LL Ch 6
13	8	27 maart	Chaudron	onderzoeksmethoden empirisch onderzoek in software engineering	
14	9	3 april	Bart Knaack	Testing & Quality Assurance (Requirements, Design, Code)	LL Ch. 10
15	10	10 april	Rijn Buve?	Gastspreker (KLM ? / TomTom?)	
16	11	17 april	Rijn Buve?		
17	12	24 april			
18		1 mei	--	Hemelvaart	
19	13	8 mei	Chaudron	Vragen-uur	

Object-Oriented Software Engineering

Practical Software Development using UML and Java

Chapter 1: Software and Software Engineering

What is Software Engineering?

What is SW quality?

What is a software development process?

1.1 The Nature of Software...

Software is intangible

- Hard to understand development effort

Software is easy to reproduce

- Cost is in its *development*
  - —in other engineering products, manufacturing is the costly stage

The industry is labor-intensive

- Hard to automate

The Nature of Software ...

Untrained people can hack something together

- Quality problems are hard to notice

Software is easy to modify

- People make changes without fully understanding it

Software does not ‘wear out’

- It deteriorates by having its design changed:
  - —erroneously, or
  - —in ways that were not anticipated, thus making it complex

The Nature of Software

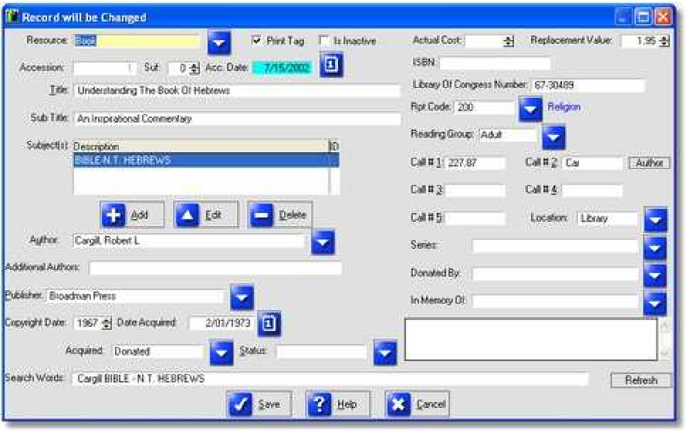
Conclusions

- Much software has poor design and is getting worse
- Demand for software is high and rising
- We are in a perpetual ‘software crisis’
- We have to learn to ‘engineer’ software

Types of Software...

Custom

- For a specific customer



Record will be Changed

Resource: Book ☐ ☑ Print Tag ☐ Is Inactive

Accession: 1 Suf: 0 Acc. Date: 7/15/2002

Title: Understanding The Book Of Hebrews

Sub Title: An Inspirational Commentary

Subject(s): Description ID
BIBLE-N.T. HEBREWS

Author: Cargill, Robert L

Additional Authors:

Publisher: Broadman Press

Copyright Date: 1967 Date Acquired: 2/01/1973

Acquired: Donated Status:

Search Words: Cargill BIBLE - N.T. HEBREWS

Actual Cost: Replacement Value: 1.95

ISBN:

Library Of Congress Number: 67-30489

Rpt Code: 200 Religion

Reading Group: Adult

Call # 1: 227.87 Call # 2: Car

Call # 3: Call # 4:

Call # 5: Location: Library

Series:

Donated By:

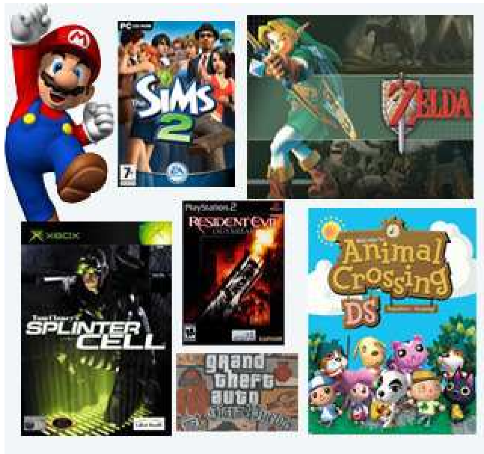
In Memory Of:

Generic

- Sold on open market
- Often called
  - —COTS (Commercial Off The Shelf)
  - —Shrink-wrapped

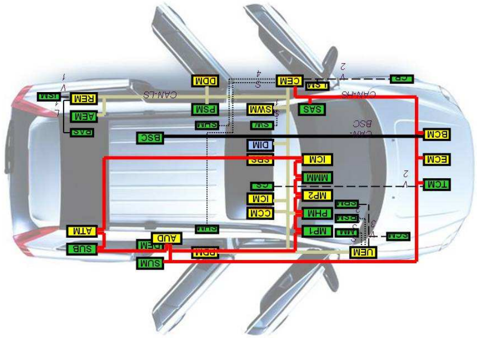
Exact software ™

ORACLE



Embedded

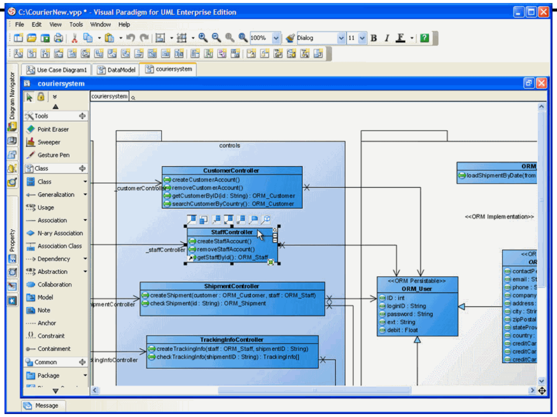
- Built into hardware
- Hard to change

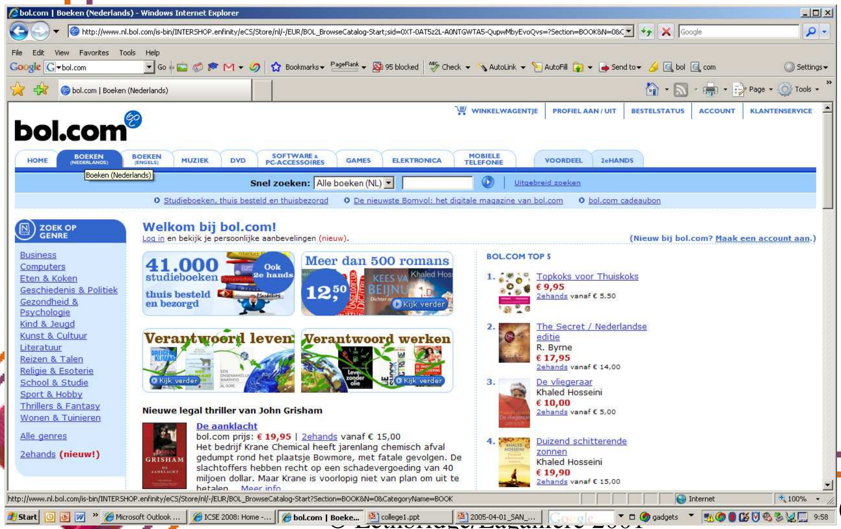


© Lethbridge/Laganière 2001

Types of Software

	Custom	Generic	Embedded
Number of *copies* in use	low	medium	high
Total *processing power* devoted to running this type of software	low	high	medium
Worldwide annual *development effort*	high	low	high





Snel zoeken: Alle boeken (NL)

Uitgebreid zoeken

Studieboeken, thuis besteld en thuisbezorgd

De nieuwste Bomvol: het digitale magazine van bol.com

bol.com cadeaubon

Welkom bij bol.com!

Log in en bekijk je persoonlijke aanbevelingen (nieuw).

(Nieuw bij bol.com? Maak een account aan.)

Business
Computers
Eten & Koken
Geschiedenis & Politiek
Gezondheid & Psychologie
Kind & Jeugd
Kunst & Cultuur
Literatuur
Reizen & Talen
Religie & Esoterie
School & Studie
Sport & Hobby
Thrillers & Fantasy
Wonen & Tuinieren

Alle genres

2ehands (nieuw!)

Nieuwe legal thriller van John Grisham

De aanklacht
bol.com prijs: € 19,95 | 2ehands vanaf € 15,00
Het bedrijf Krane Chemical heeft jarenlang chemisch afval gedumpt rond het plaatsje Bowmore, met fatale gevolgen. De slachtoffers hebben recht op een schadevergoeding van 40 miljoen dollar. Maar Krane is voorlopig niet van plan om uit te betalen. Meer info

BOL.COM TOP 5

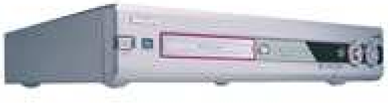
1. Topkoks voor Thuiskoks
€ 9,95
2ehands vanaf € 5,50

2. The Secret / Nederlandse editie
R. Byrne
€ 17,95
2ehands vanaf € 14,00

3. De vliegeraar
Khaled Hosseini
€ 10,00
2ehands vanaf € 5,00

4. Duizend schitterende zonnen
Khaled Hosseini
€ 19,90
2ehands vanaf € 15,00

http://www.nl.bol.com/is-bin/INTERSHOP.enfinity/eCS/Store/nl/-/EUR/BOL_BrowseCatalog-Start?Section=BOOK&N=0&CategoryName=BOOK



Types of Software

Real time software

- E.g. control and monitoring systems
- Must react immediately
- Safety often a concern

Business Information Systems (Data processing)

- Used to run businesses
- Accuracy and security of data are key

Some software has both aspects

1.2 What is Software Engineering?...

The process of solving customers' problems by the systematic development and evolution of large, high-quality software systems within cost, time and other constraints

Solving customers' problems

- This is the goal of software engineering
- Sometimes the solution is to buy, not build
- Adding unnecessary features does not help solve the problem
- Software engineers must communicate effectively to identify and understand the problem

© Lethbridge/Laganière 2001

What is Software Engineering?…

Systematic development and evolution

- An engineering process involves applying well understood techniques in a organized and disciplined way
- Many well-accepted practices have been formally standardized
  —e.g. by the IEEE or ISO

Large, high quality software systems

- Software engineering techniques are needed because large systems cannot be completely understood by one person
- Teamwork and co-ordination are required
- Key challenge: Dividing up the work and ensuring that the parts of the system work properly together
- The end-product that is produced must be of sufficient quality

What is Software Engineering?...

Other definitions:

- IEEE: (1) the application of a systematic, disciplined, quantifiable approach to the development, operation, maintenance of software; that is, the application of engineering to software. (2) The study of approaches as in (1)
- The Canadian Standards Association: The systematic activities involved in the design, implementation and testing of software to optimize its production and support.

What is Software Engineering?

Cost, time and other constraints

- Finite resources
- The benefit must outweigh the cost
- Others are competing to do the job cheaper and faster
- Inaccurate estimates of cost and time have caused many project failures

What is the Science of Software Engineering?

The scientific study of
methods, techniques, processes
for creating software

Effect of techniques on quality, productivity

Object Oriented programming languages are better.

Agile development processes lead to faster development.

Often studied empirically

1.4 Stakeholders in Software Engineering

1. Users

- Those who use the software

2. Customers

- Those who pay for the software

3. Software developers

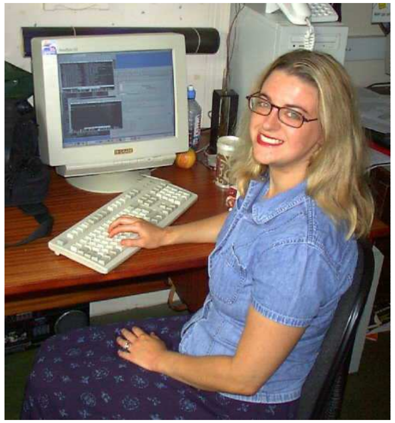
- Those who make the software

4. Development Managers

All four roles can be fulfilled by the same person

What does a Software Engineer do?

individually



programming
documenting
planning
presenting
reviewing
reporting

interacting
with clients

listening
explaining
feedback
selling

in team

Microsoft 1978

Specializing in different roles

- designing, programming, testing …

brainstorming
discussing
planning

1.5 Software Quality...

Usability

- Users can learn it and fast and get their job done easily

Efficiency

- It doesn’t waste resources such as CPU time and memory

Reliability

- It does what it is required to do without failing

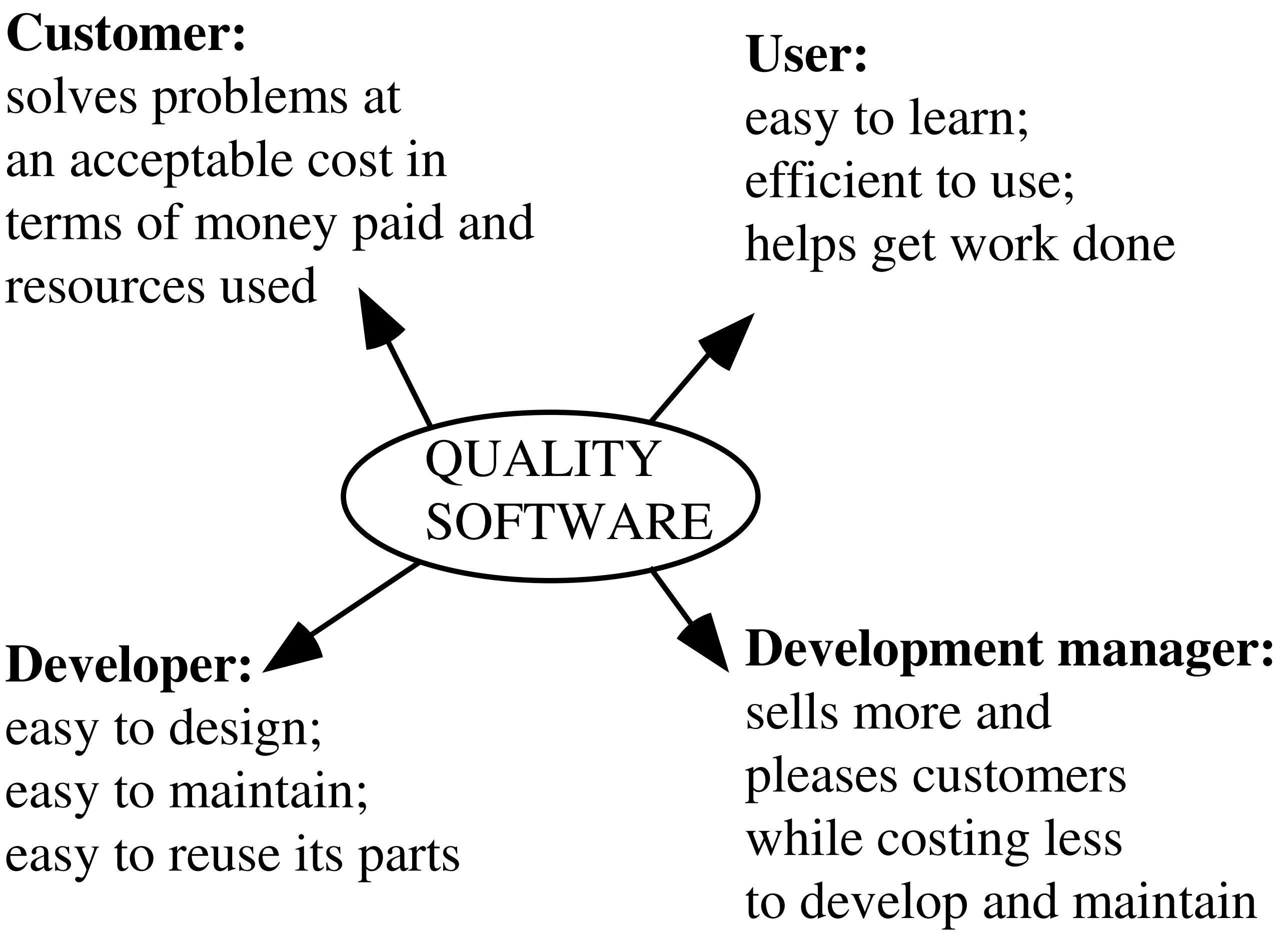
Maintainability

- It can be easily changed

Reusability

- Its parts can be used in other projects, so reprogramming is not needed

Software Quality...



QUALITY SOFTWARE

Customer:
solves problems at
an acceptable cost in
terms of money paid and
resources used

User:
easy to learn;
efficient to use;
helps get work done

Developer:
easy to design;
easy to maintain;
easy to reuse its parts

Development manager:
sells more and
pleases customers
while costing less
to develop and maintain

Software Quality

The different qualities can conflict

- Increasing efficiency can reduce maintainability or reusability
- Increasing usability can reduce efficiency

Setting objectives for quality is a key engineering activity

- You then design to meet the objectives
- Avoids 'over-engineering' which wastes money

Optimizing is also sometimes necessary

- E.g. obtain the highest possible reliability using a fixed budget

Internal Quality Criteria

These:

- Characterize aspects of the design of the software
- Have an effect on the external quality attributes
- E.g.
    - —The amount of commenting of the code
    - —The complexity of the code

Short Term Vs. Long Term Quality

Short term:

- Does the software meet the customer's immediate needs?
- Is it sufficiently efficient for the volume of data we have today?

Long term:

- Maintainability
- Customer's future needs

© Lethbridge/Laganière 2001

1.6 Software Engineering Projects

Most projects are *evolutionary* or *maintenance* projects, involving work on *legacy* systems

- <u>Corrective</u> projects: fixing defects
- <u>Adaptive</u> projects: changing the system in response to changes in
  - —Operating system
  - —Database
  - —Rules and regulations
- <u>Enhancement</u> projects: adding new features for users
- <u>Reengineering</u> or <u>perfective</u> projects: changing the system internally so it is more maintainable

Software Engineering Projects

‘Green field’ projects

- New development
- The minority of projects

Software Engineering Projects

Projects that involve building on a *framework* or a set of existing components.

- The framework is an application that is missing some important details.
    - —E.g. Specific rules of this organization.
- Such projects:
    - —Involve plugging together *components* that are:
        - Already developed.
        - Provide significant functionality.
    - —Benefit from reusing reliable software.
    - —Provide much of the same freedom to innovate found in green field development.

1.7 Activities Common to Software Projects...

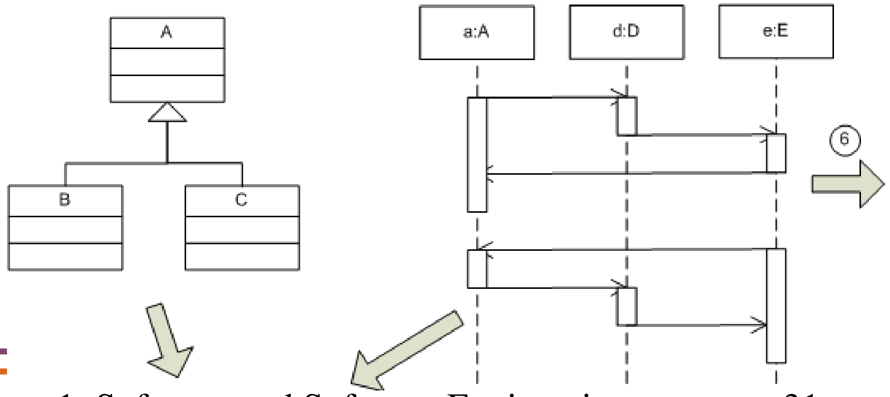
Requirements and specification

- Includes
  - —Domain analysis
  - —Defining the problem
  - —Requirements gathering
    - Obtaining input from as many sources as possible
  - —Requirements analysis
    - Organizing the information
  - —Requirements specification
    - Writing detailed instructions about how the software should behave

Activities Common to Software Projects...

Design

- Deciding how the requirements should be implemented, using the available technology
- Includes:
  - —Systems engineering: Deciding what should be in hardware and what in software
  - —Software architecture: Dividing the system into subsystems and deciding how the subsystems will interact
  - —Detailed design of the internals of a subsystem
  - —User interface design
  - —Design of databases



Activities Common to Software Projects

Modeling

- Creating representations of the domain or the software
  - —Use case modeling
  - —Structural modeling
  - —Dynamic and behavioural modeling

Programming

Quality assurance

- Reviews and inspections
- Testing

Deployment

Managing the process

1.8 The Eight Themes of the Book

1. Understanding the customer and the user

2. Basing development on solid principles and reusable technology

3. Object orientation

4. Visual modeling using UML

5. Evaluation of alternatives

6. Iterative development

7. Communicating effectively using documentation

8. Risk management in all SE activities

Difficulties and Risks in Software Engineering

- **Complexity and large numbers of details**
- **Uncertainty about technology**
- **Uncertainty about requirements**
- **Uncertainty about software engineering skills**
- **Constant change**
- **Deterioration of software design**
- **Political risks**

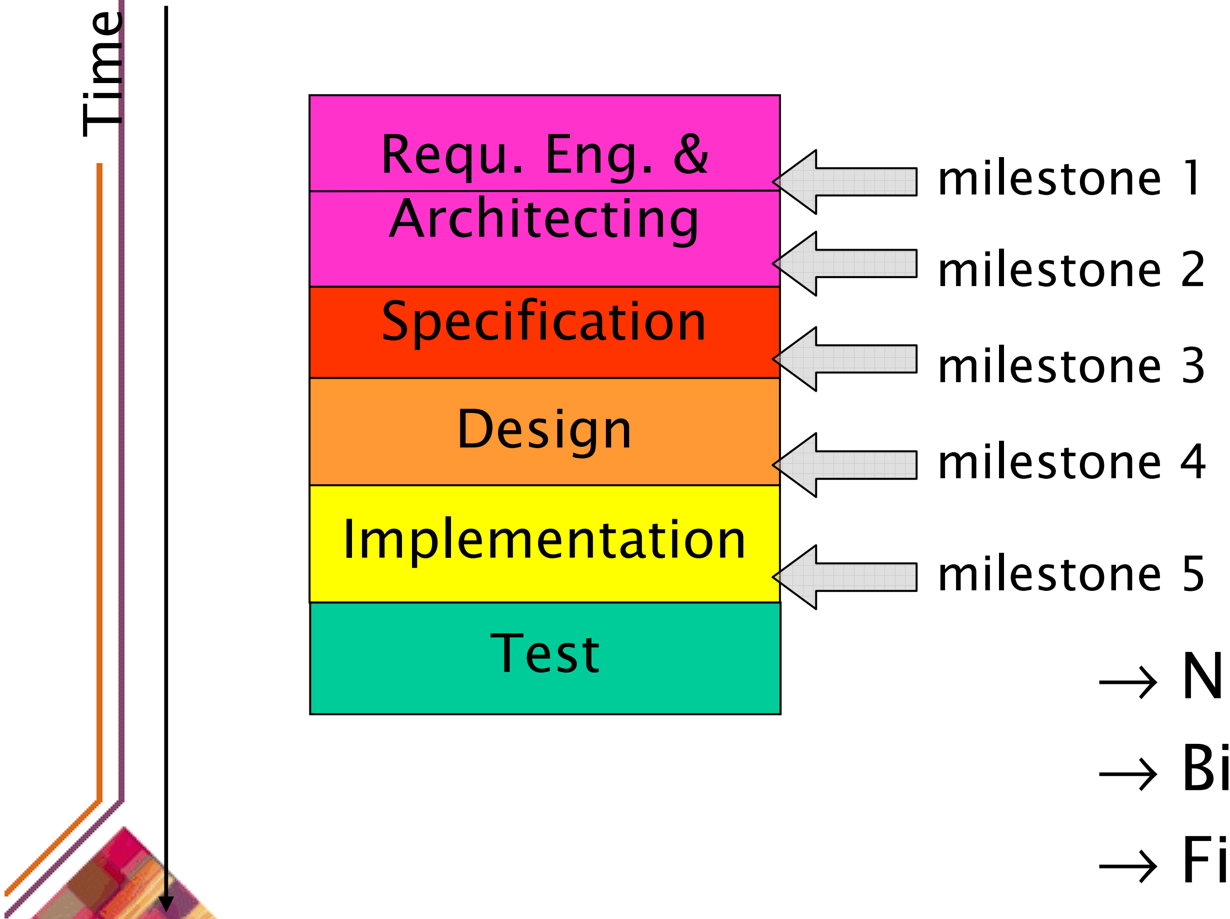
Software Development Process Models

- **Waterfall**
- **Iterative**

© Lethbridge/Laganière 2001

SDP Models (1)

Waterfall Model (Mid 70ies)

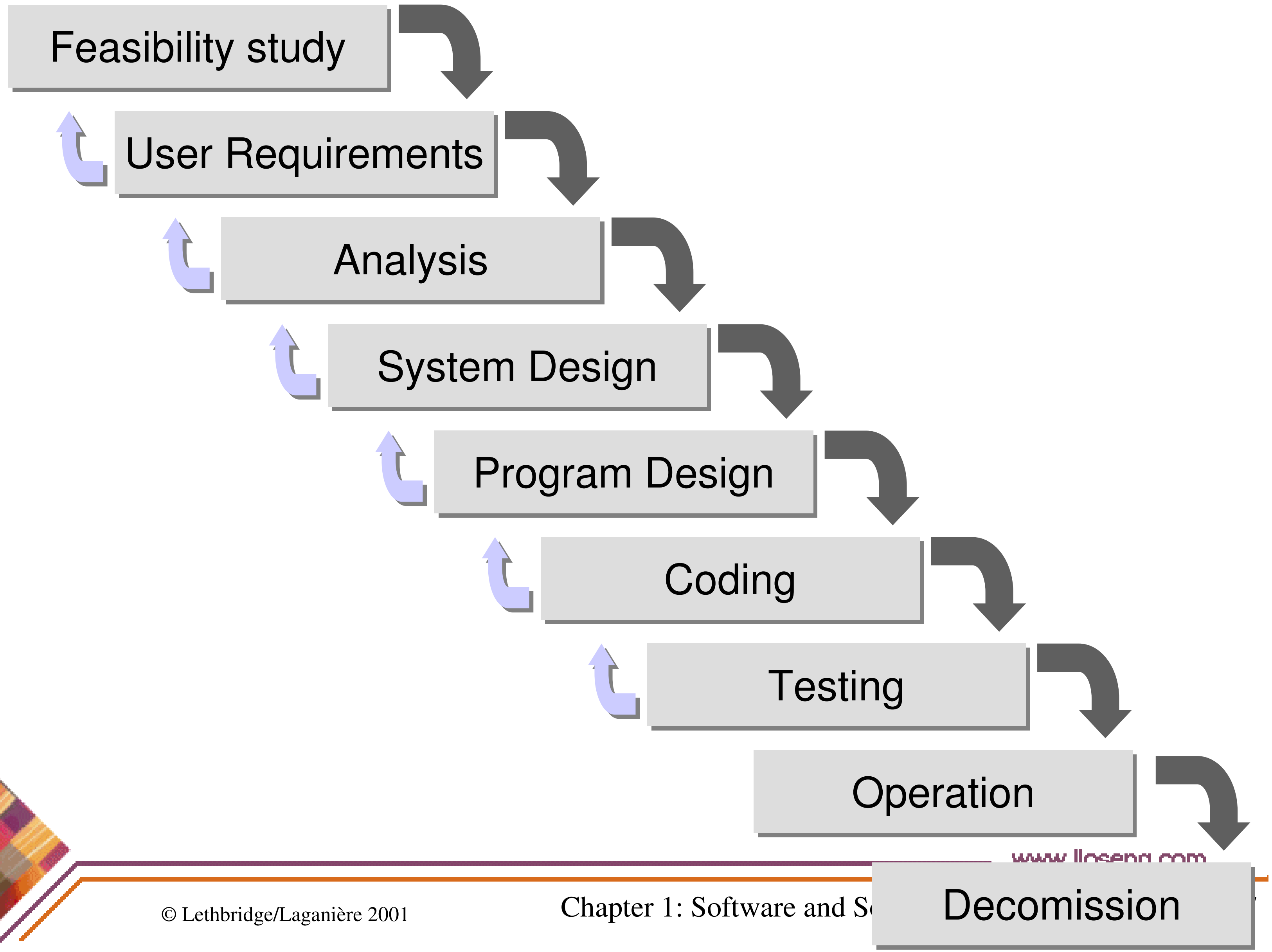


→ No iterations

→ Big bang scenario

→ First-time right

The waterfall model



Feasibility study

User Requirements

Analysis

System Design

Program Design

Coding

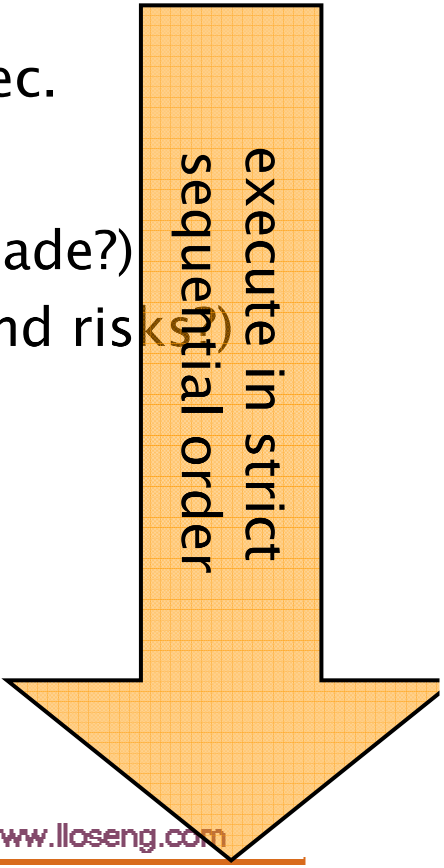
Testing

Operation

Decomission

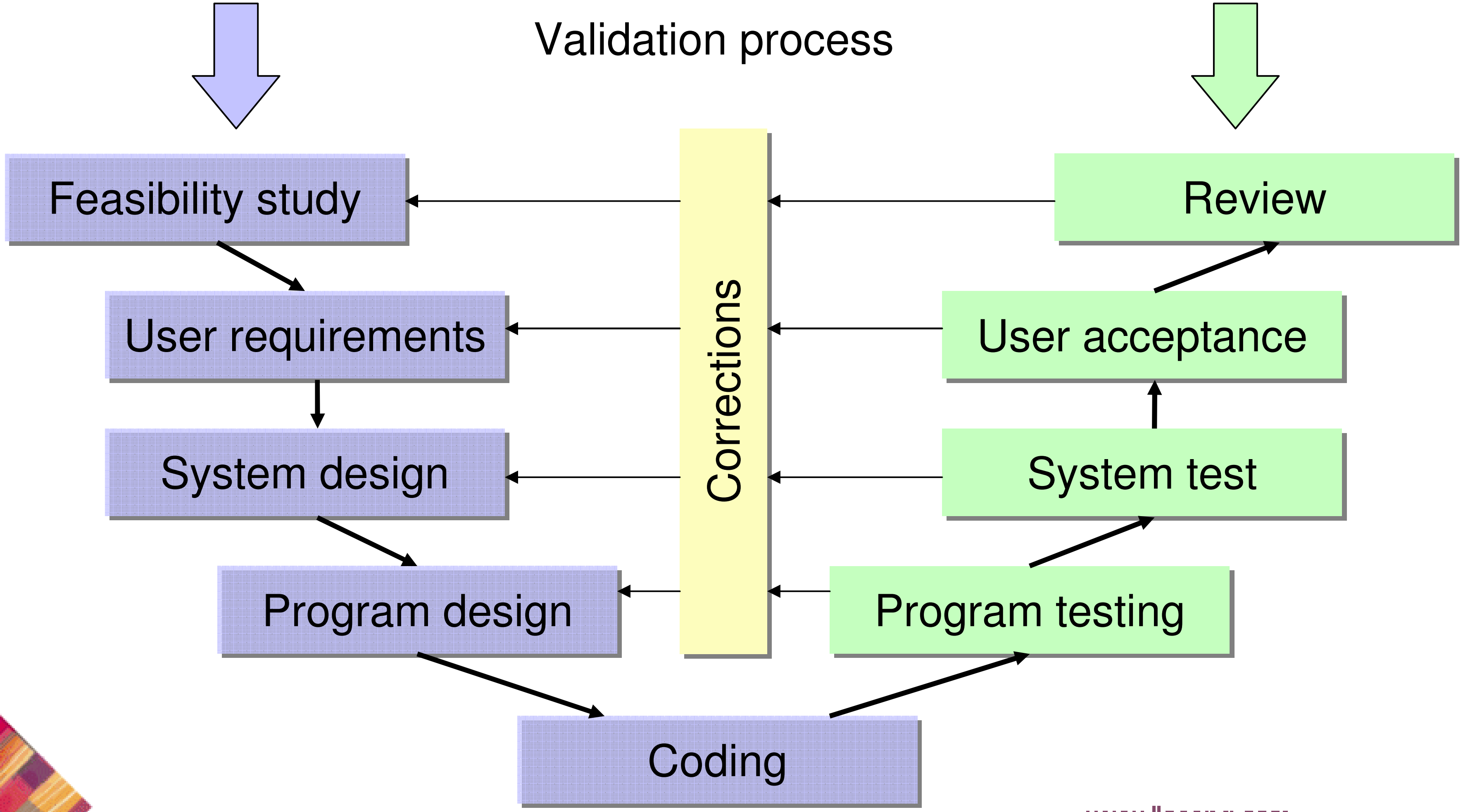
The Classical Waterfall Model (Example)

Requirements	Vision & first idea
Analysis	Requirements Document (WHAT) Context model & Requirements Spec.
Architectural Model (HOW)	
	Feasibility Study (can product be made?) Risk Assessment (project threats and risks?)
Design & Specification	
	System Spec. (WHAT): Design (HOW)
Implementation	Coding & Testing (HOW):
Test	Integration and acceptance Test



The V-process model

Another way of looking at the waterfall model



© Lethbridge/Laganière 2001

Problems of the Waterfall Process (1)

The milestones did not fit in many project situations, leading to:

- Gold-plating → Iterative development

  Extensive written requirements spec's cause overemphasis on "complete" requirements and invite "just-in-case" additions
- Inflexible point solutions

  – Fixed requirements spec's produce inflexible solutions optimized around the initial problem statement

  – Forced early design decisions
- Bad usability → A prototype is worth a 100.000 words

  Written req. spec's are not nearly as effective as a prototype

  Requirements often emerge only after demonstration and feedback

www.lloseng.com

The waterfall model (cont‘d)

Pros:

Imposes structure on complex projects

Every stage needs to be checked and signed off:

- Elimination of midstream changes

Good when quality requirements dominate cost and schedule requirements

Cons:

Limited scope for flexibility / iterations

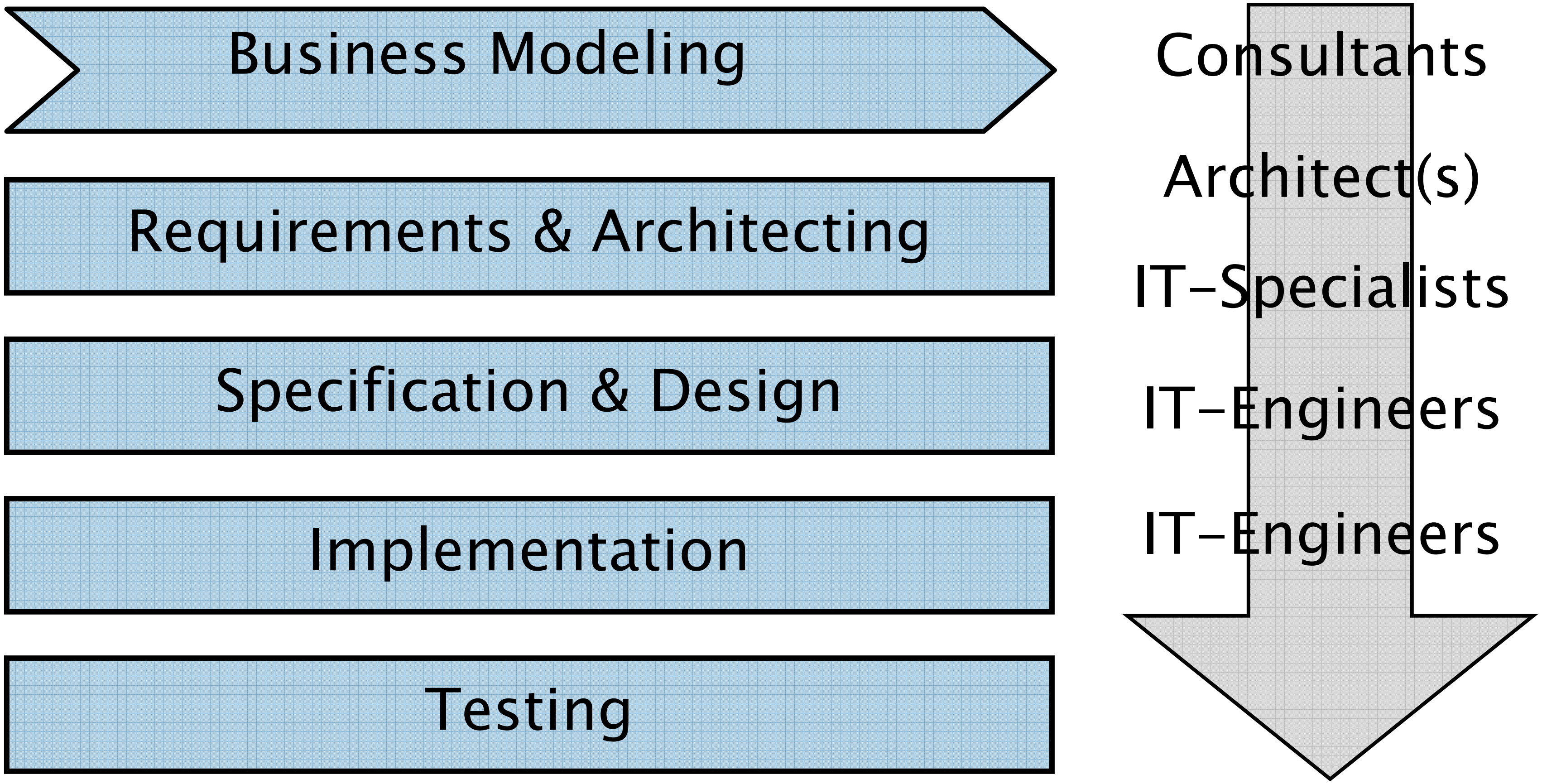
Full requirements specification at the beginning:

- User specifications

No tangible product until the end

Problems of the Waterfall Process (2)

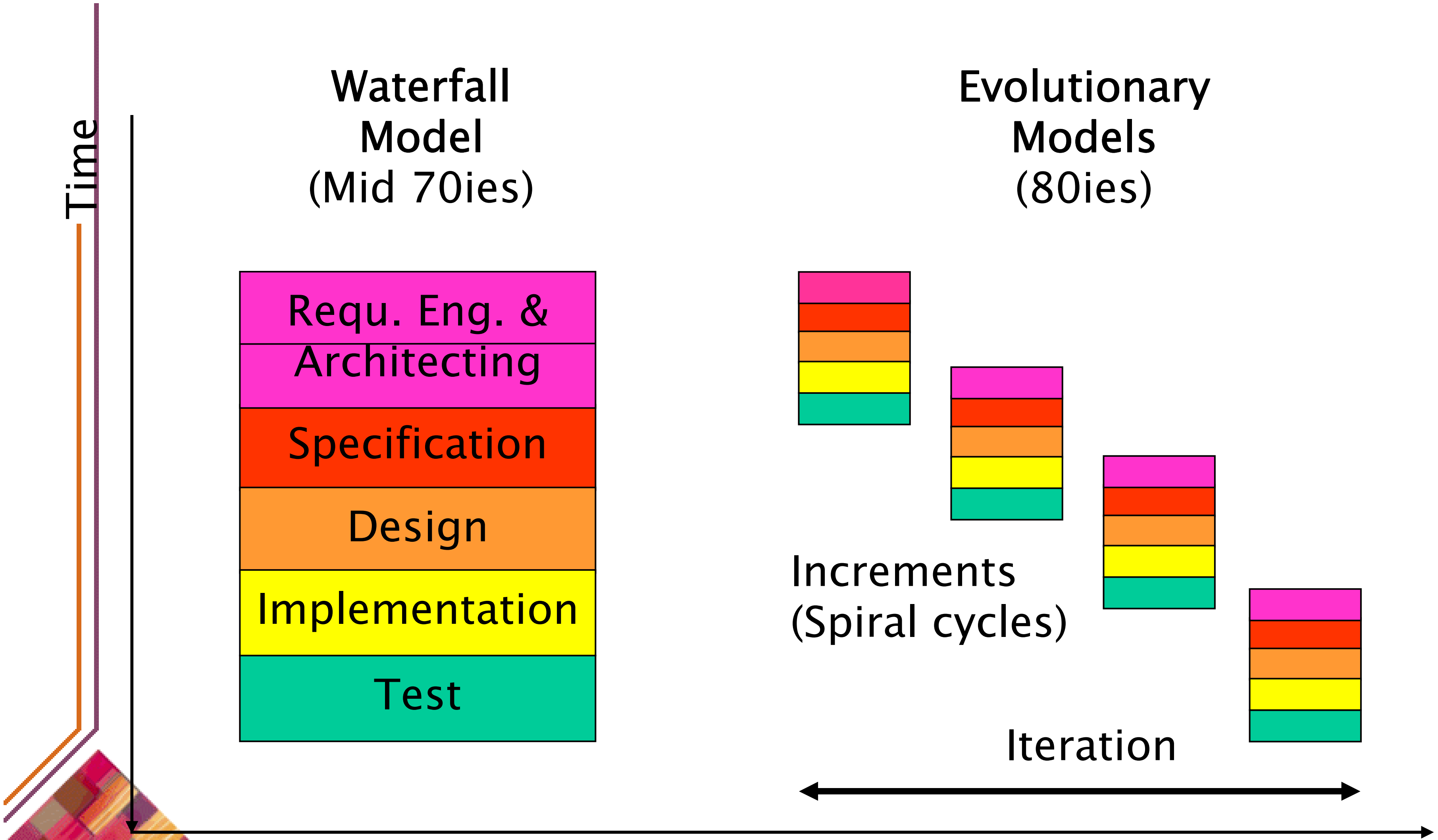
Different phases are handled by different people



Communication becomes highly critical

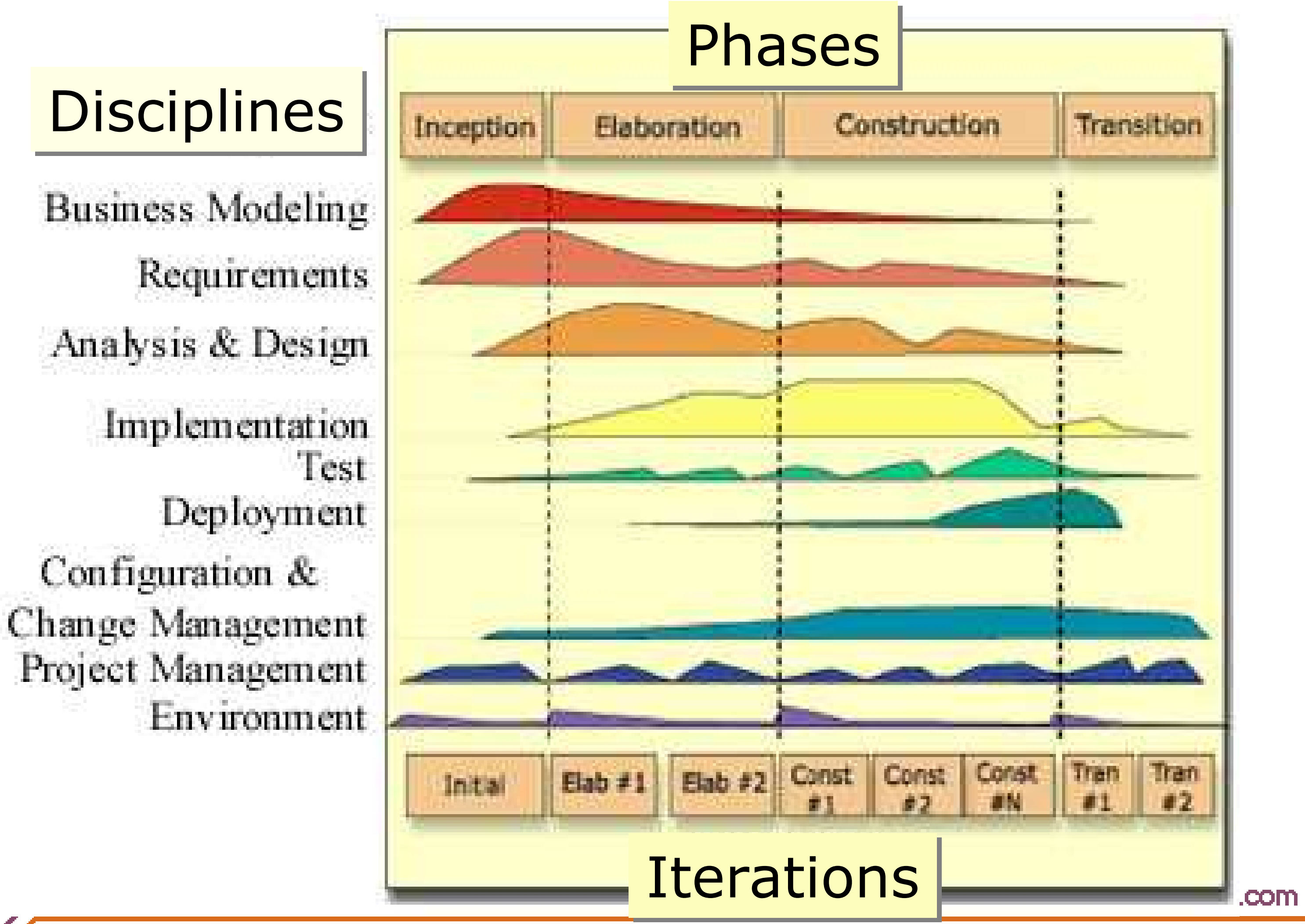
© Lethbridge/Laganière 2001

SDP Models (2)


Time
Waterfall Model (Mid 70ies)
Requ. Eng. & Architecting
Specification
Design
Implementation
Test
Evolutionary Models (80ies)
Increments (Spiral cycles)
Iteration
Scope

© Lethbridge/Laganière 2001

Rational Unified Process (RUP)



Problems of Evolutionary Models

- **Inflexible point solutions**

  The initial release is optimized for demonstration, consequently the architecture is difficult to extend

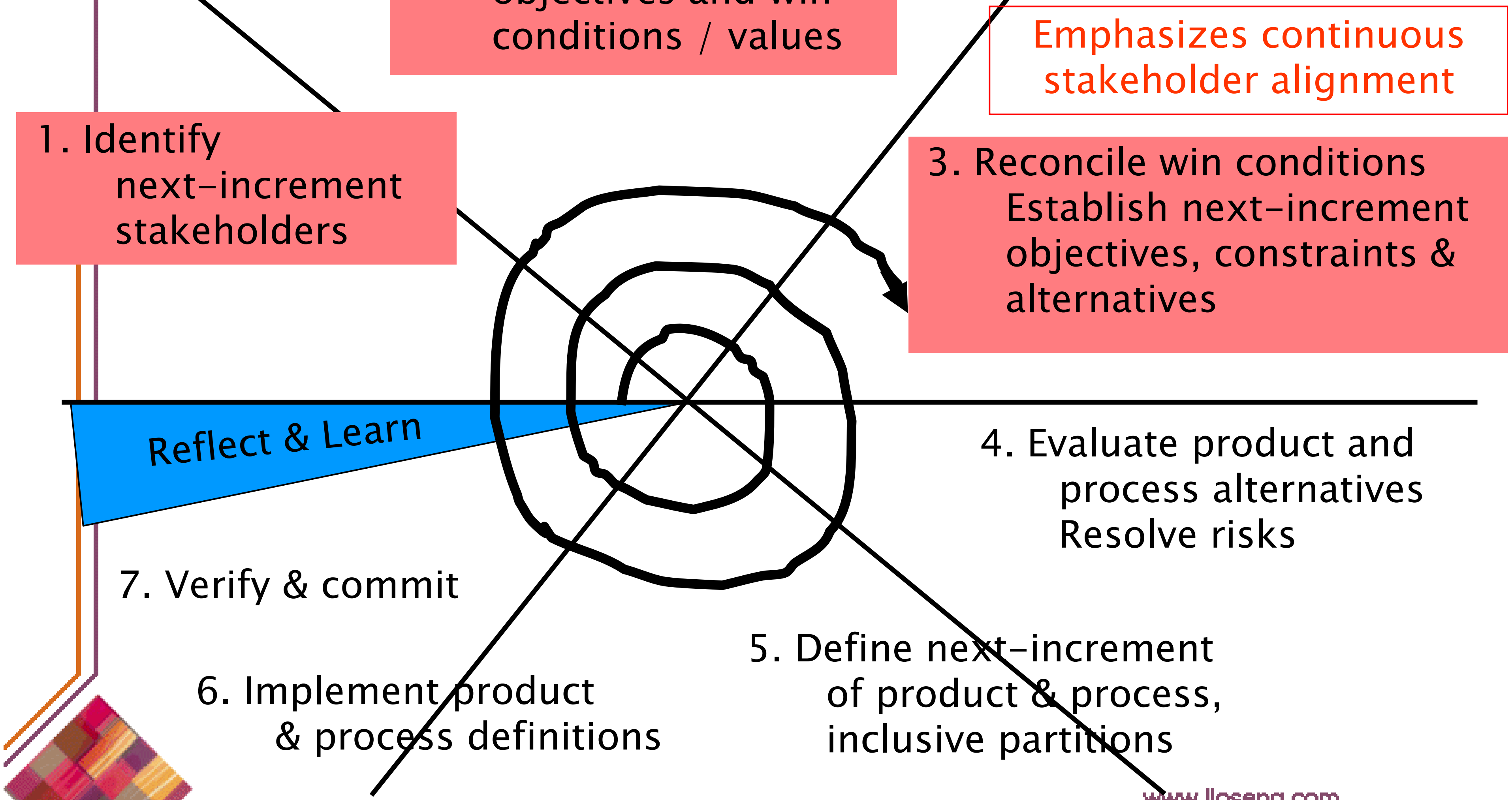
- **High-risk downstream capabilities**

  The initial release often defers quality attributes (dependability, scalability, etc.) in favor of early functionality

Win-Win Spiral Model

(Boehm, 1998)

Emphasizes continuous stakeholder alignment



1. Identify next-increment stakeholders

2. Identify stakeholders objectives and win conditions / values

3. Reconcile win conditions Establish next-increment objectives, constraints & alternatives

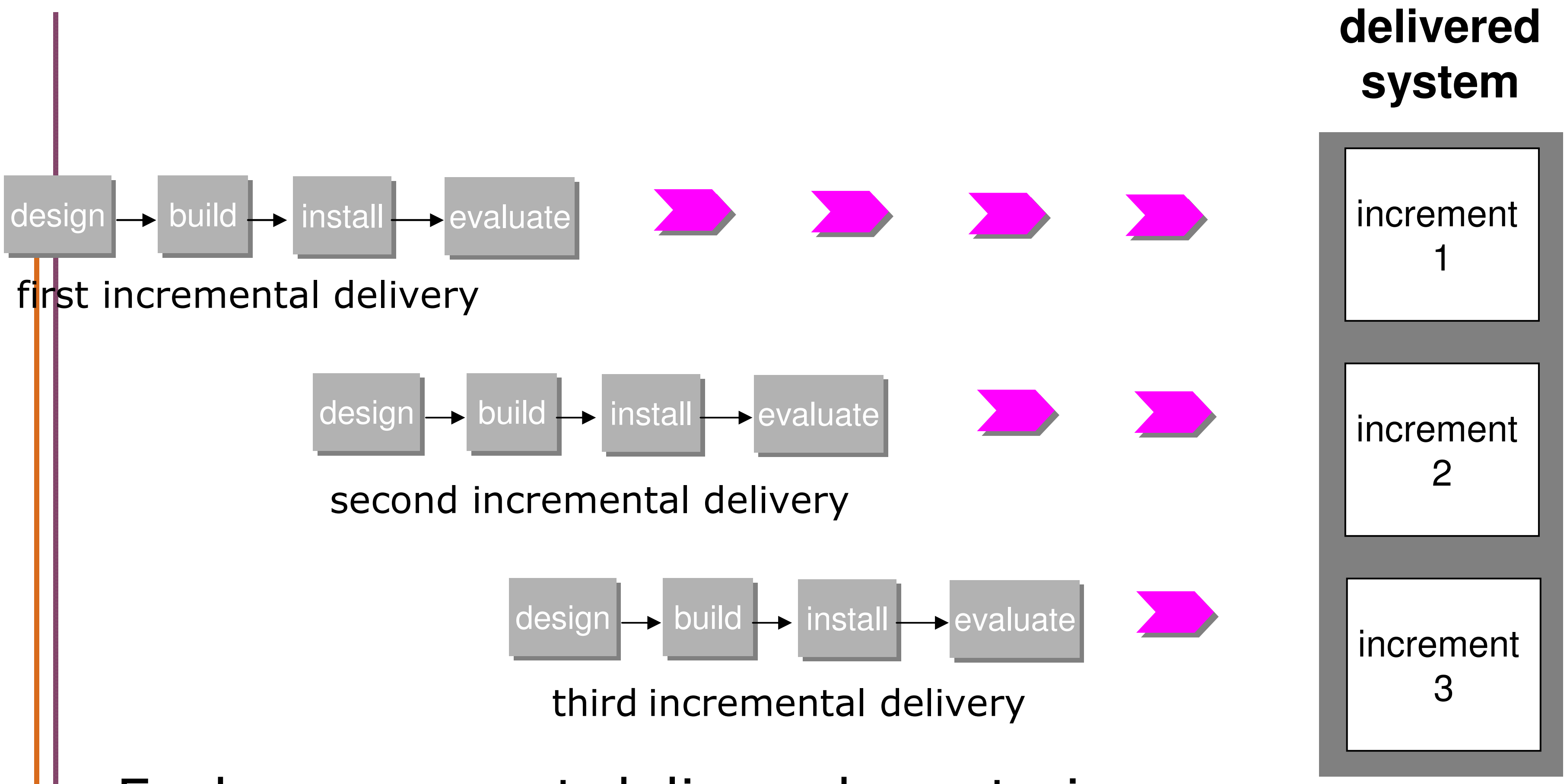
4. Evaluate product and process alternatives Resolve risks

5. Define next-increment of product & process, inclusive partitions

6. Implement product & process definitions

7. Verify & commit

Incremental delivery



Each component delivered must give some benefit to the stakeholders

© Lethbridge/Laganière 2001

Proliferation of Alternative Models

Early 1990's

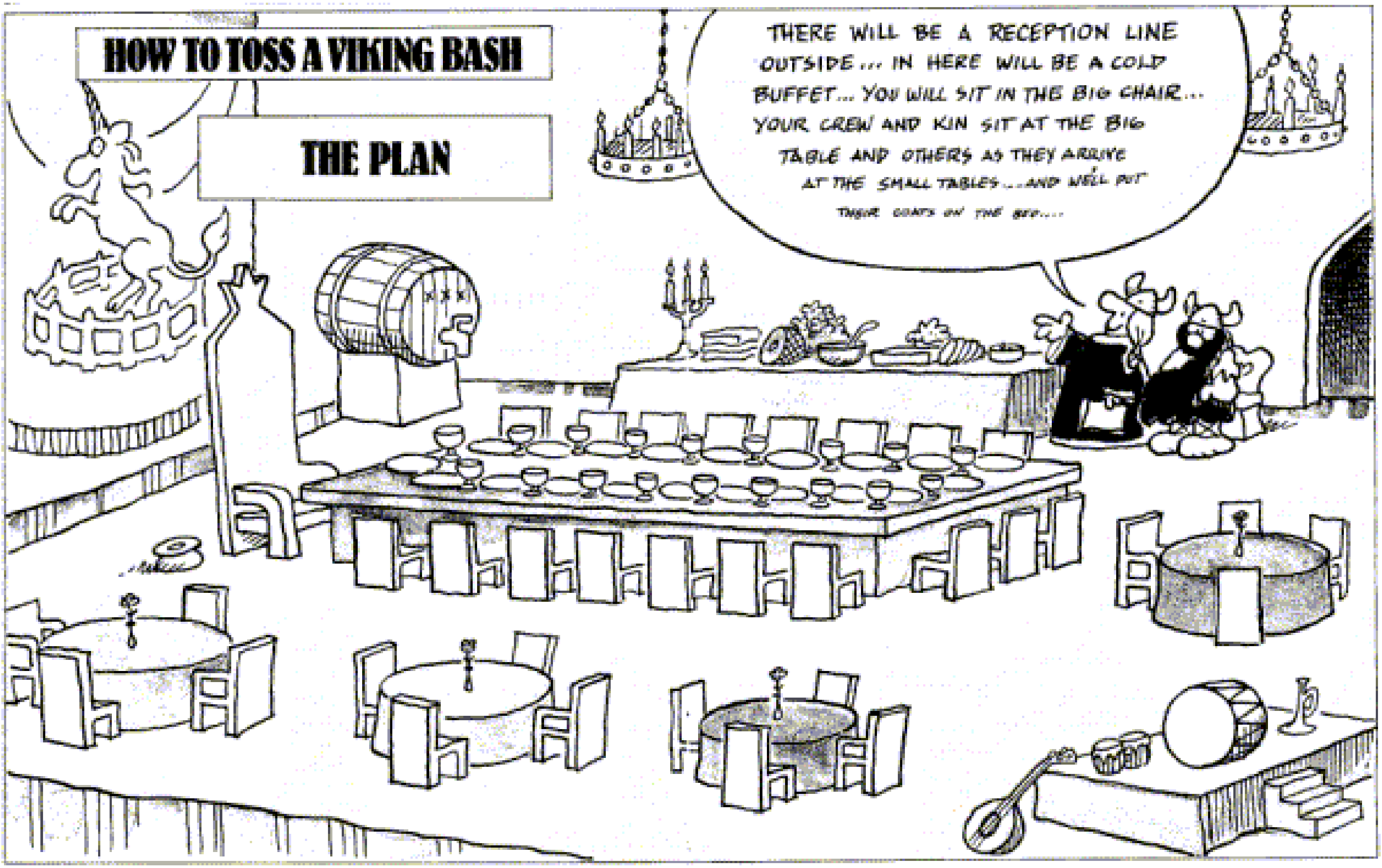
Examples:

- Risk-, reuse-, legacy- and demo-driven
- Various variants of evolutionary development
- Hybrids

SW organizations had difficulties to establish a common reference

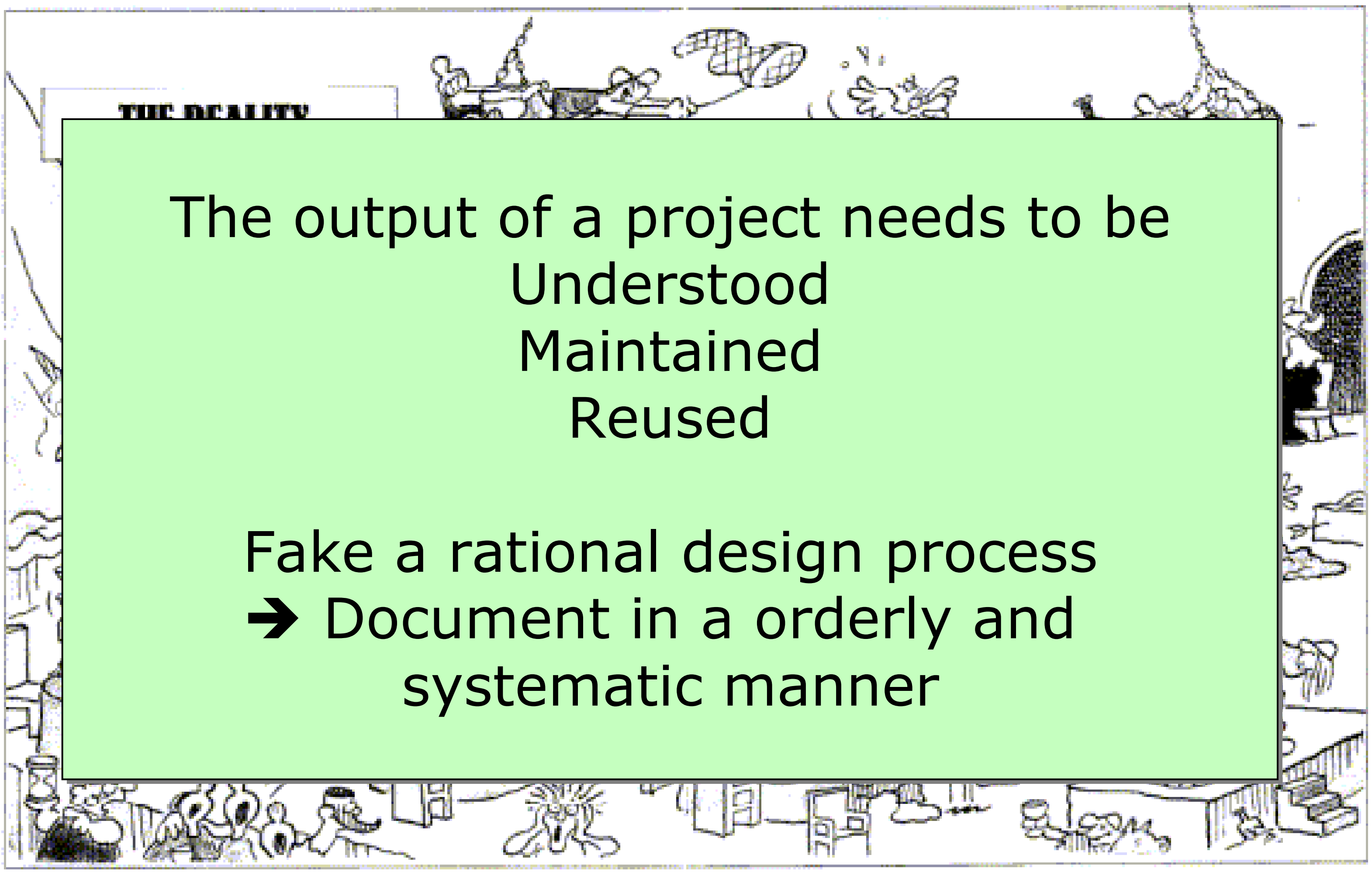
© Lethbridge/Laganière 2001

The plan


HOW TO TOSS A VIKING BASH
THE PLAN
THERE WILL BE A RECEPTION LINE OUTSIDE... IN HERE WILL BE A COLD BUFFET... YOU WILL SIT IN THE BIG CHAIR... YOUR CREW AND KIN SIT AT THE BIG TABLE AND OTHERS AS THEY ARRIVE AT THE SMALL TABLES... AND WE'LL PUT THEIR COATS ON THE BED....

www.lloseng.com

Reality



The output of a project needs to be
Understood
Maintained
Reused

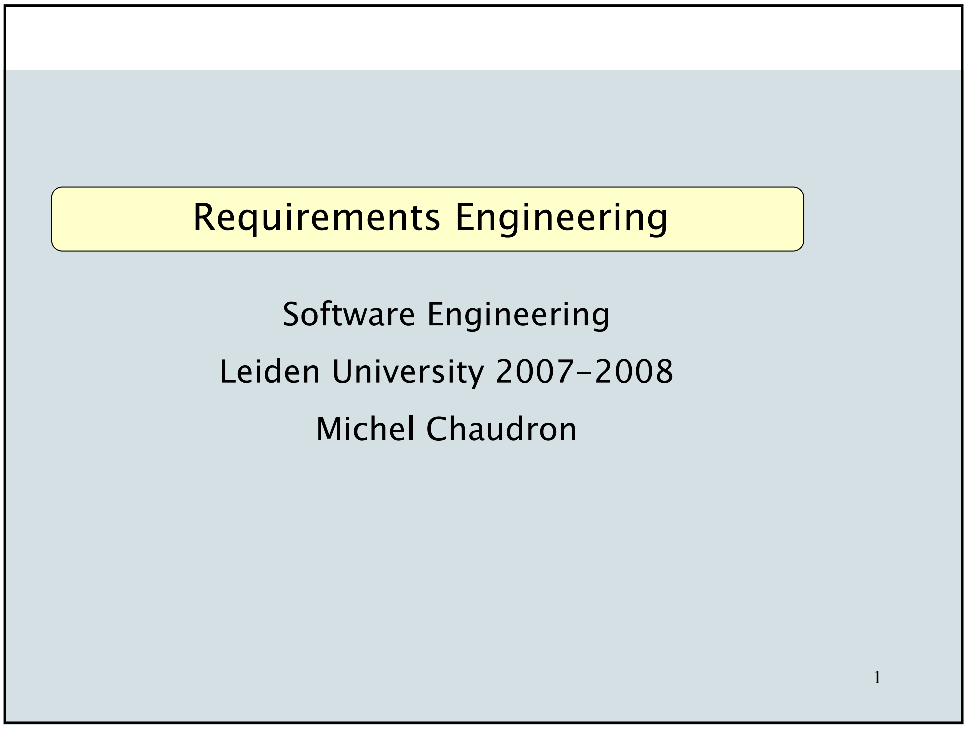
Fake a rational design process
➔ Document in a orderly and systematic manner

Questions?

Homework:

- **Read**
  - Chapter 1 Introduction Software Engineering
  - Chapter 2 Review Object Orientation

© Lethbridge/Laganière 2001



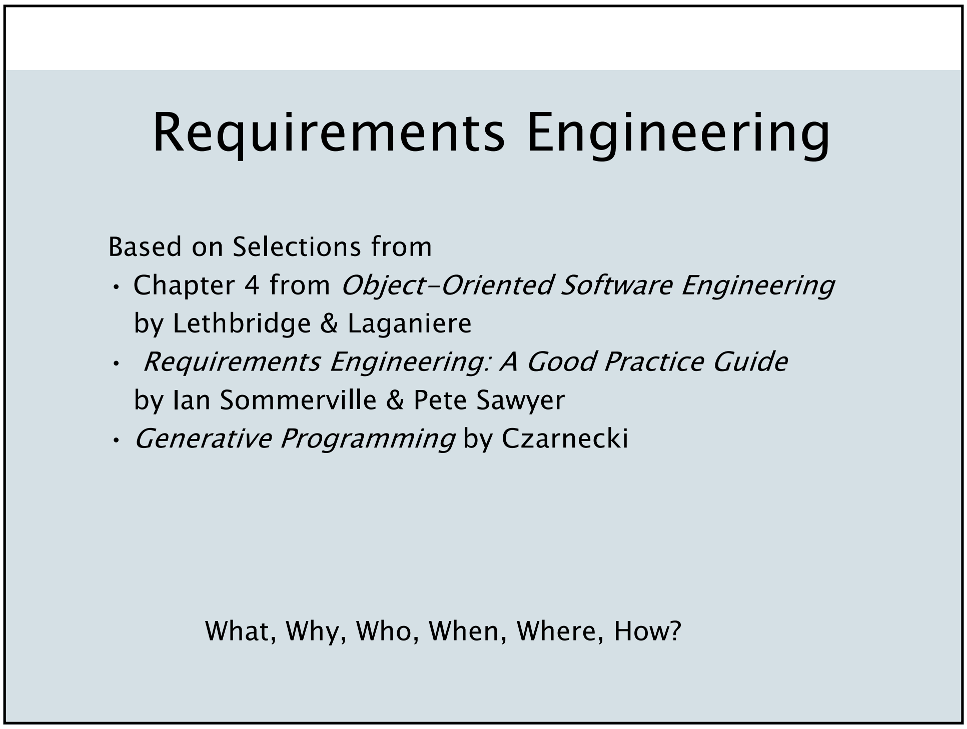
Requirements Engineering

Software Engineering

Leiden University 2007-2008

Michel Chaudron

1



Requirements Engineering

Based on Selections from
- Chapter 4 from *Object-Oriented Software Engineering* by Lethbridge & Laganiere
- *Requirements Engineering: A Good Practice Guide* by Ian Sommerville & Pete Sawyer
- *Generative Programming* by Czarnecki

What, Why, Who, When, Where, How?

Requirements engineering

- The process of establishing the services that the customer requires from a system and the constraints under which it operates and is developed.
- The requirements themselves are the descriptions of the system services and constraints that are generated during the requirements engineering process.

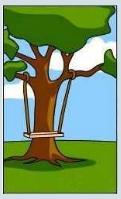
"The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the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

No other part is more difficult to rectify later".

Fred Brooks, "No Silver Bullet",
IEEE Computer,1987
Author of The Mythical Man-month

Understanding the problem

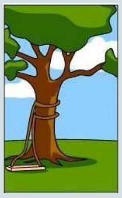
What the customer explained



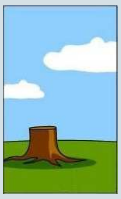
What the project leader understood

What the analyst designed

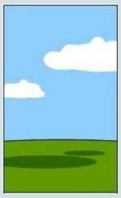
What the consultant defined



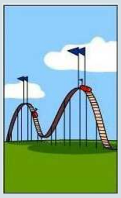
What the programmer made



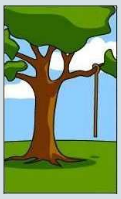
How it was maintained



What was documented



What was charged



What was installed

What the client needed

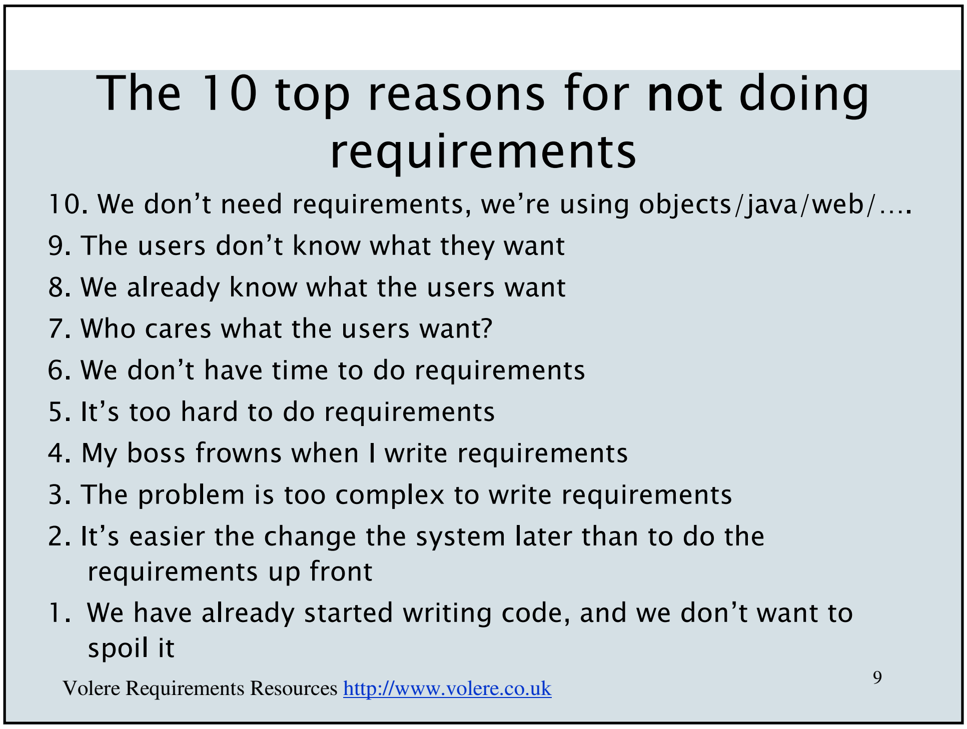
Developers' View of Users	Users' View of Developers
Users don't know what they want. Users can't articulate what they want. Users have too many needs that are politically motivated. Users want everything right now. Users can't prioritize needs. Users refuse to take responsibility for the system. Users are unable to provide a usable statement of needs. Users are not committed to system development projects. Users are unwilling to compromise. Users can't remain on schedule.	Developers don't understand operational needs. Developers place too much emphasis on technicalities. Developers try to tell us how to do our jobs. Developers can't translate clearly stated needs into a successful system. Developers say no all the time. Developers are always over budget. Developers are always late. Developers ask users for time and effort, even to the detriment of the users' important primary duties. Developers set unrealistic standards for requirements definition. Developers are unable to respond quickly to legitimately changing needs.

Learning from each other

Users, customers, managers, domain experts, and developers share different skills, backgrounds, and expectations.

Developing a shared vision

Requirements emerge from a process of co-operative learning in which they are explored, prioritized, negotiated, evaluated, and documented.



The 10 top reasons for **not** doing requirements

10. We don't need requirements, we're using objects/java/web/....

9. The users don't know what they want

8. We already know what the users want

7. Who cares what the users want?

6. We don't have time to do requirements

5. It's too hard to do requirements

4. My boss frowns when I write requirements

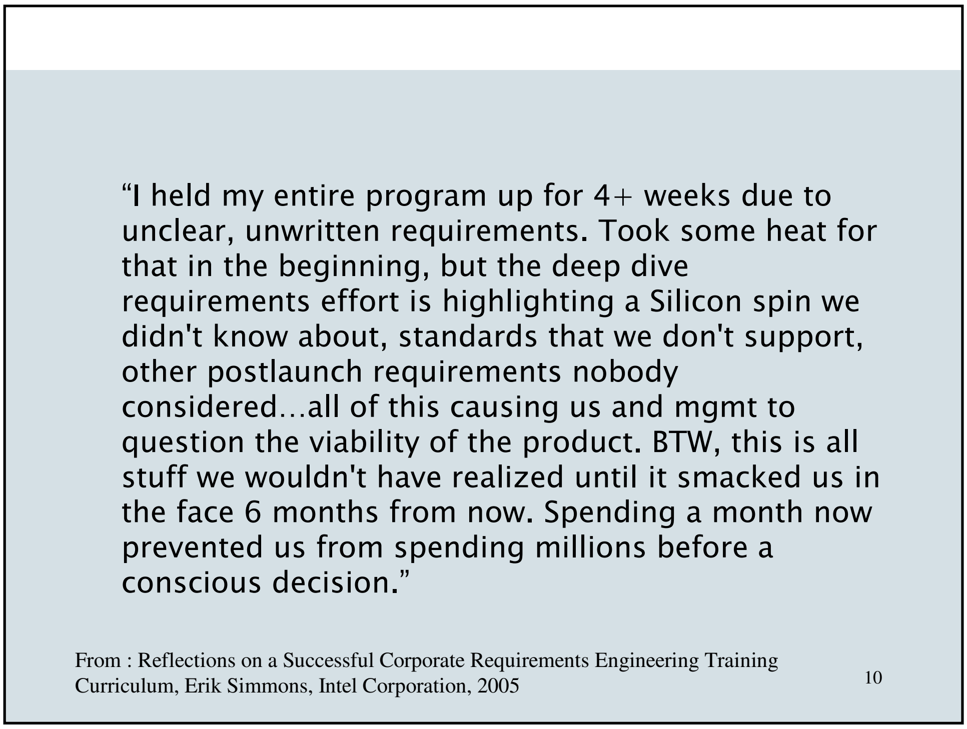
3. The problem is too complex to write requirements

2. It's easier the change the system later than to do the requirements up front

1. We have already started writing code, and we don't want to spoil it

Volere Requirements Resources http://www.volere.co.uk

9



"I held my entire program up for 4+ weeks due to unclear, unwritten requirements. Took some heat for that in the beginning, but the deep dive requirements effort is highlighting a Silicon spin we didn't know about, standards that we don't support, other postlaunch requirements nobody considered…all of this causing us and mgmt to question the viability of the product. BTW, this is all stuff we wouldn't have realized until it smacked us in the face 6 months from now. Spending a month now prevented us from spending millions before a conscious decision."

From : Reflections on a Successful Corporate Requirements Engineering Training Curriculum, Erik Simmons, Intel Corporation, 2005

10

Stakeholder issues

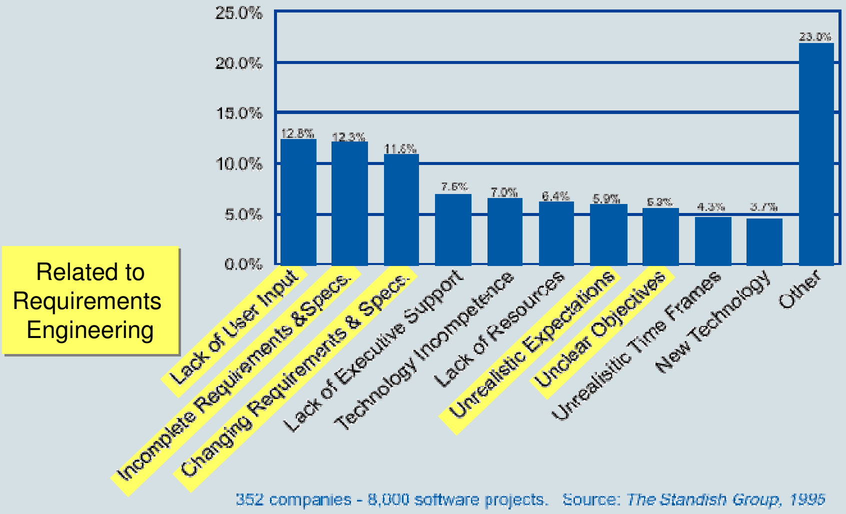
Steve McConnell, in his book *Rapid Development*, details a number of ways users can inhibit requirements gathering:

- Users don't understand what they want or users don't have a clear idea of their requirements
- Users won't commit to a set of written requirements
- Users insist on new requirements after the cost and schedule have been fixed.
- Communication with users is slow
- Users often do not participate in reviews or are incapable of doing so.
- Users are technically unsophisticated
- Users don't understand the development process.
- Users don't know about present technology.

11

Why Software Projects Fail

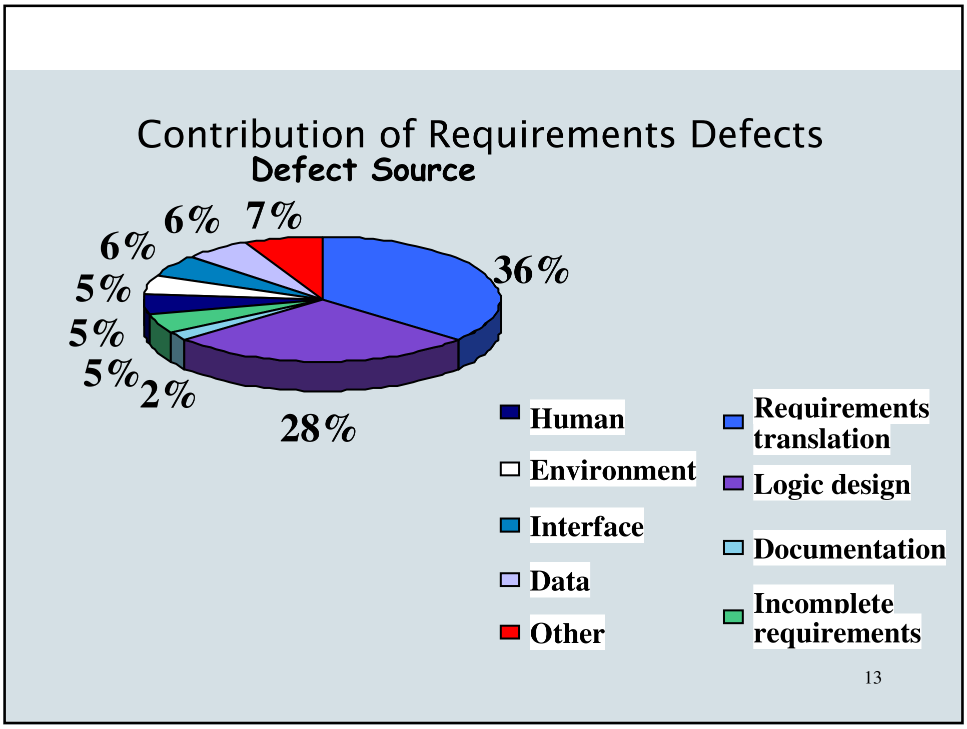
Example of empirical research



Related to Requirements Engineering

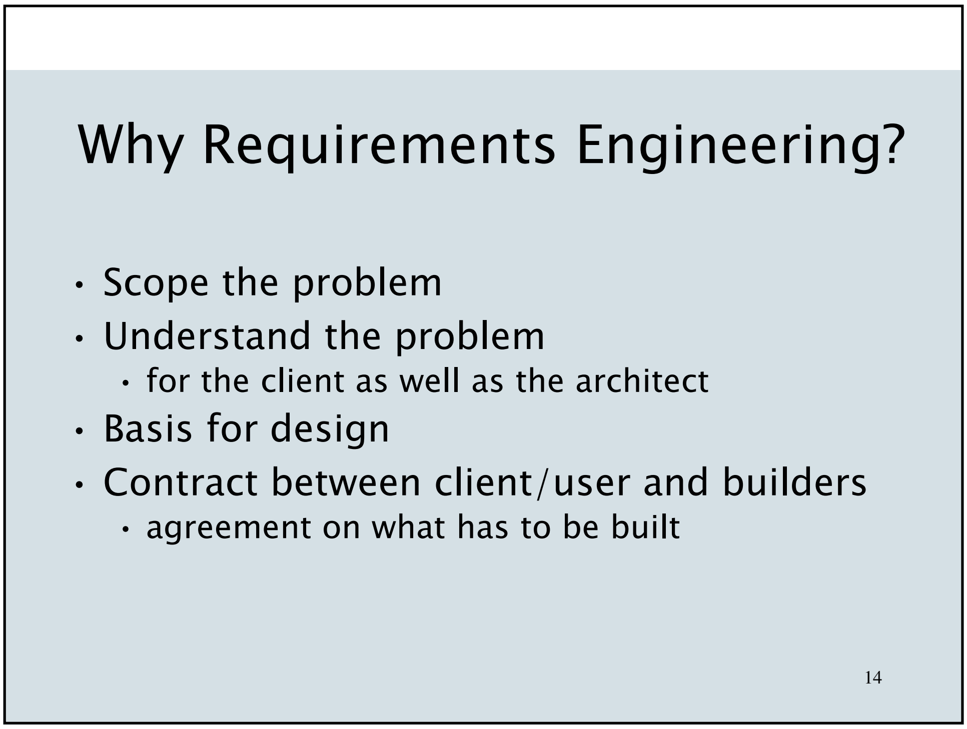
352 companies - 8,000 software projects. Source: *The Standish Group, 1995*

12



Contribution of Requirements Defects

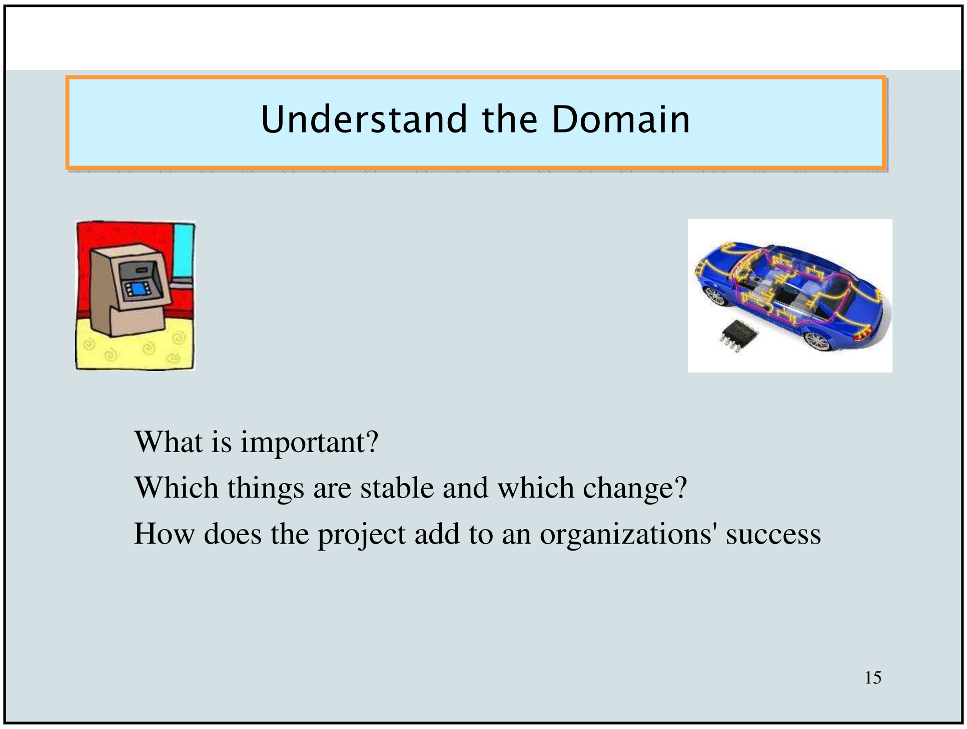
13



Why Requirements Engineering?

- Scope the problem
- Understand the problem
  - for the client as well as the architect
- Basis for design
- Contract between client/user and builders
  - agreement on what has to be built

14

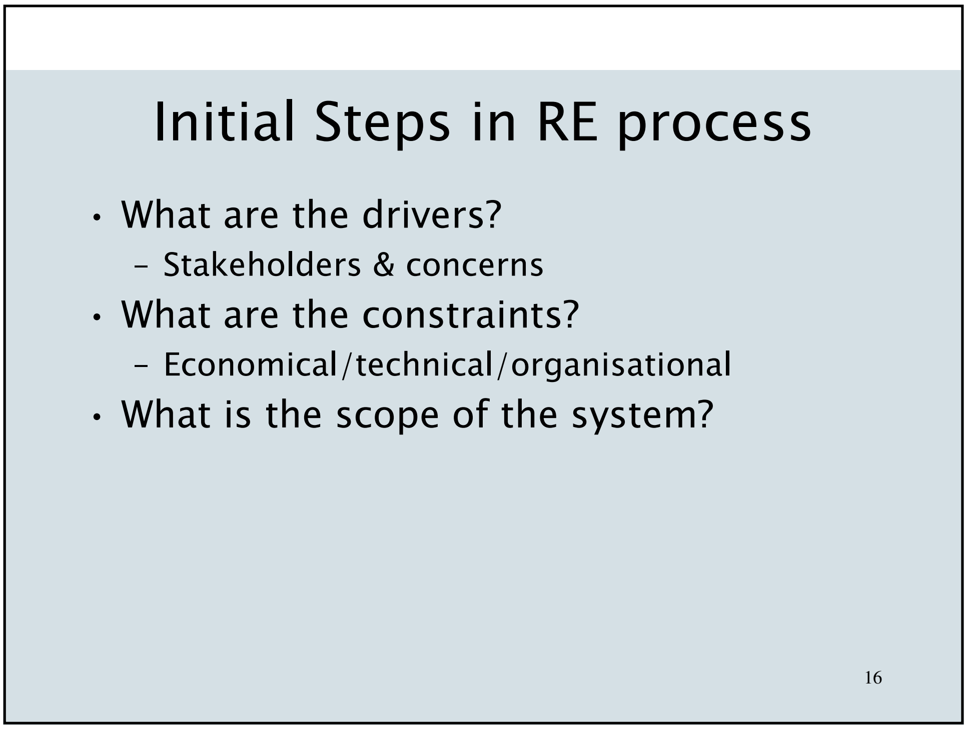


Understand the Domain

What is important?

Which things are stable and which change?

How does the project add to an organizations' success



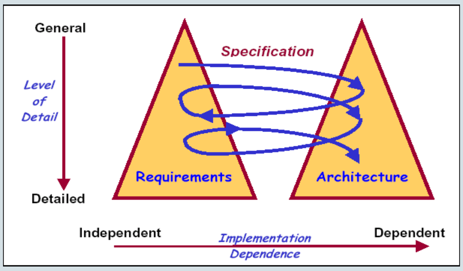
Initial Steps in RE process

- What are the drivers?
  - Stakeholders & concerns
- What are the constraints?
  - Economical/technical/organisational
- What is the scope of the system?

Twin Peaks Process

Separate but concurrent development of requirements & architecture

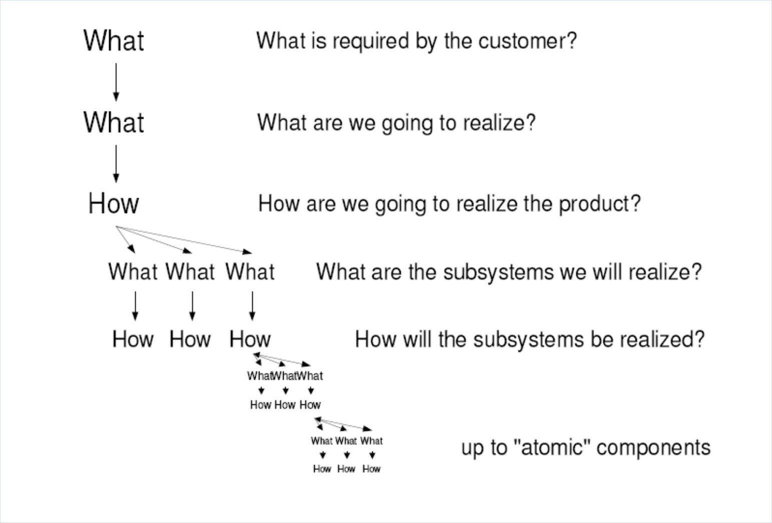
WHAT:
problem
structuring



HOW:
solution
structuring

Progressing understanding of architecture & design provides a basis for discovering further system requirements and vice versa

There is interaction between available solutions and requirements



What — What is required by the customer?

What — What are we going to realize?

How — How are we going to realize the product?

What What What — What are the subsystems we will realize?

How How How — How will the subsystems be realized?

WhatWhatWhat
How How How

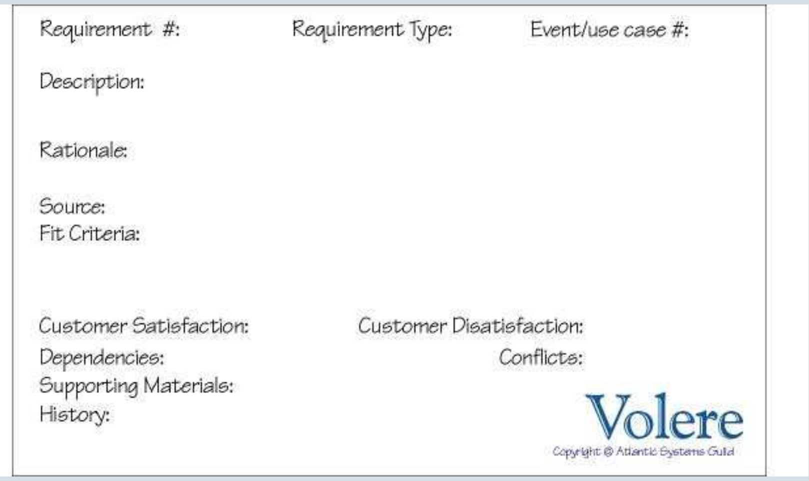
What What What
How How How — up to "atomic" components

Slide by Gerrit Muller, ESI, 2007

What is a Requirement ?

- A statement about the proposed system that all stakeholders agree must be made true in order for the customer's problem to be adequately solved.
  - Short and concise piece of information
  - Says something about the system
  - All the stakeholders have agreed that it is valid
  - It helps solve the customer's problem
  - Contract between customer and builder

Example Requirement Template



Requirement #: Requirement Type: Event/use case #:

Description:

Rationale:

Source:
Fit Criteria:

Customer Satisfaction: Customer Disatisfaction:
Dependencies: Conflicts:
Supporting Materials:
History:

Volere
Copyright © Atlantic Systems Guild

Errors

Up to 30-50% of the errors found further downstream the development process are due to errors in the requirements.

Requirements errors are typically non-clerical.

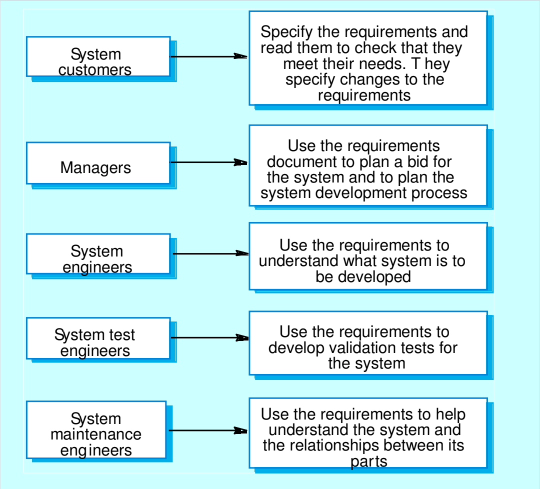
incorrect facts	49%
omissions	31%
inconsistencies	13%
ambiguities	5%

Requirements errors can be detected.

Review by authors 23%

Review by others 10%

Users of a requirements document



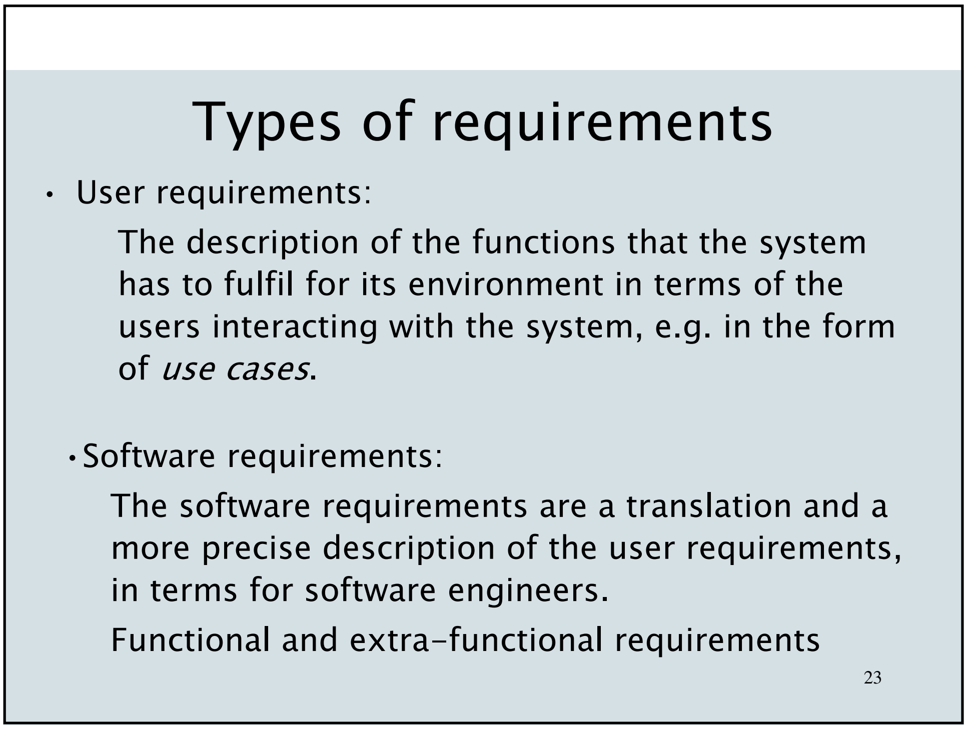
System customers → Specify the requirements and read them to check that they meet their needs. T hey specify changes to the requirements

Managers → Use the requirements document to plan a bid for the system and to plan the system development process

System engineers → Use the requirements to understand what system is to be developed

System test engineers → Use the requirements to develop validation tests for the system

System maintenance engineers → Use the requirements to help understand the system and the relationships between its parts



Types of requirements

- User requirements:

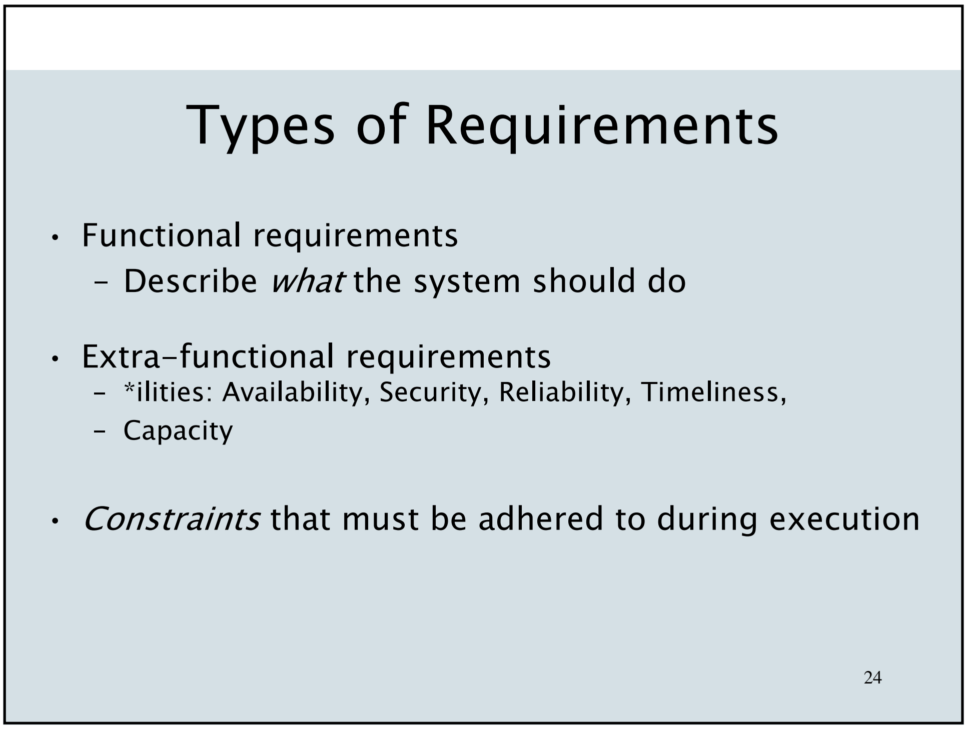
  The description of the functions that the system has to fulfil for its environment in terms of the users interacting with the system, e.g. in the form of *use cases*.

- Software requirements:

  The software requirements are a translation and a more precise description of the user requirements, in terms for software engineers.

  Functional and extra-functional requirements

23

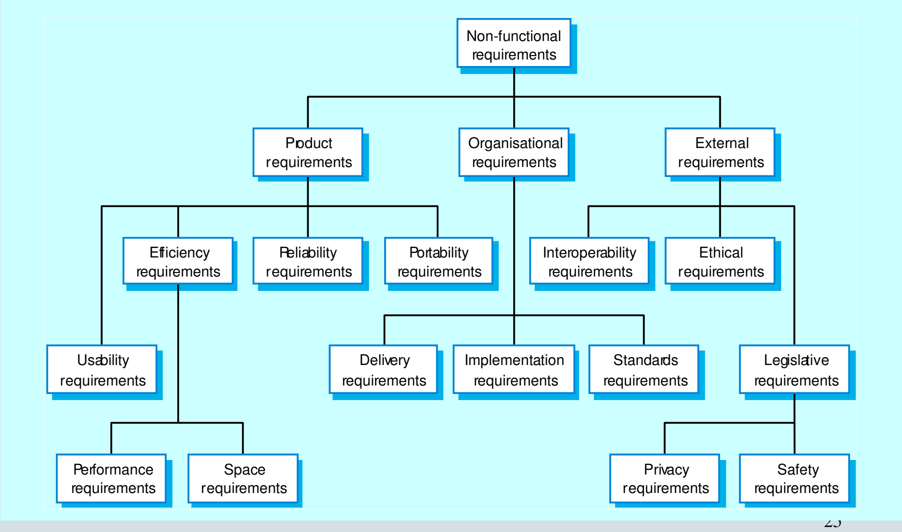


Types of Requirements

- Functional requirements
  - Describe *what* the system should do
- Extra-functional requirements
  - *ilities: Availability, Security, Reliability, Timeliness,
  - Capacity
- *Constraints* that must be adhered to during execution

24

Types of extra-functional req’rements



25

©Ian Sommerville 2004 **Software Engineering, 7th edition. Chapter 6**

Functional requirements

- What *inputs* the system should accept
- What *outputs* the system should produce
- What data the system should *store* that other systems might use
- What *computations* the system should perform

26

Examples

- The system shall allow users to search for an item by title, author, or ISBN.

Defines system functionality.

- If an item is not returned within the period of load, then the person who loans the item will be fined Euro 1 per week.

Defines (causal) relations between system functions.

Examples of XFR: Reliability

Typically expressed in terms of

for repairable systems

- **Mean Time Between Failures (MTBF)**
  - Number of hours that pass before a component fails
  - E.g. 2 failures per million hours:
    - MTBF = 10^6 / 2 = 0,5 * 10^6 hr

For non-repairable systems

- **Mean Time To Failure (MTTF)**
  - Mean time expected until the first failure of a system
  - Is a statistical value over a long period of time

- **Mean Time To Repair (MTTR)** Availability

Examples XFR: Maintainability

Maintainability

The average person time required to fix a category 3 defect (including testing and documentation upgrade) shall not exceed two person days.

System Quality Attributes

- Time To Market
- Cost and Benefits
- Projected life time
- Targeted Market
- Integration with Legacy System
- Roll back Schedule

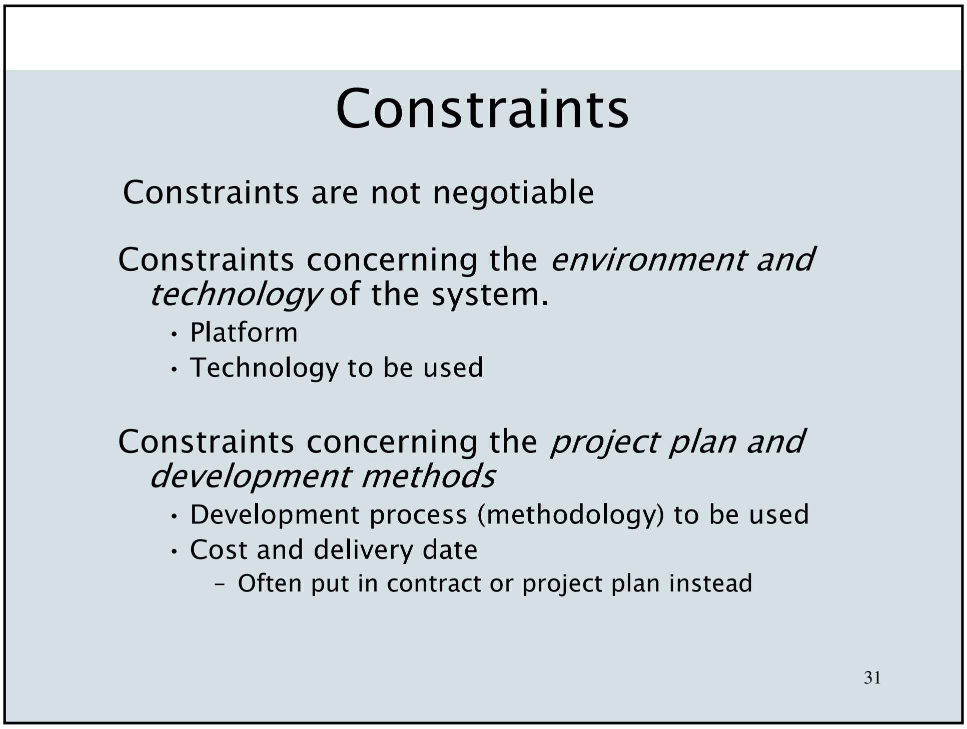
Business Community view

- Performance
- Availability
- Usability
- Security

End User's view

- Maintainability
- Portability
- Reusability
- Testability

Developer's view



Constraints

Constraints are not negotiable

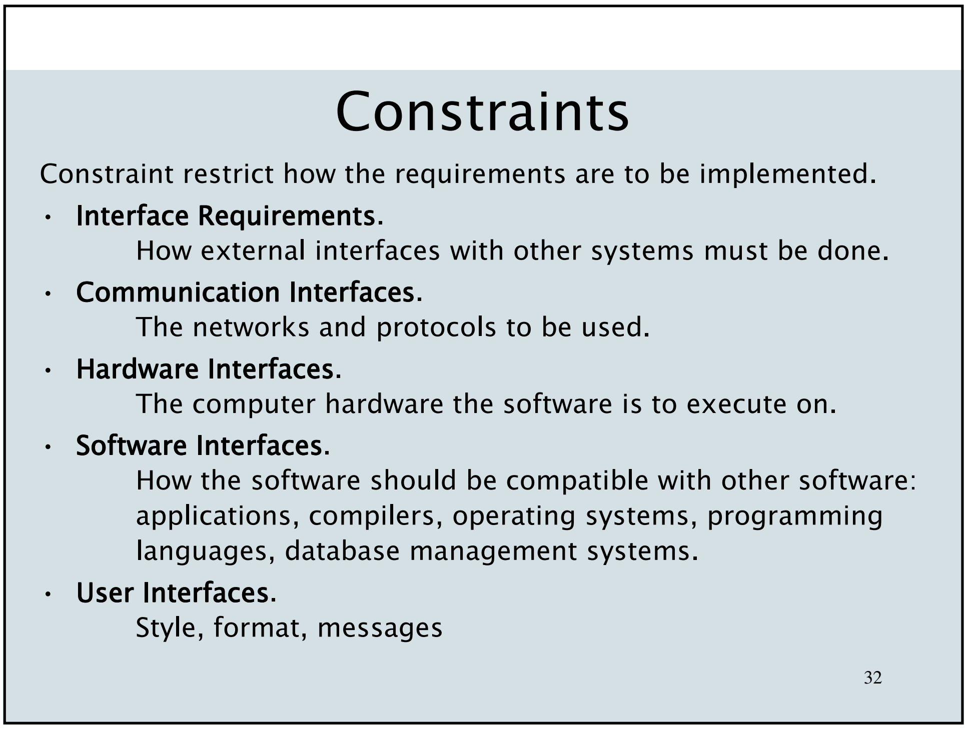
Constraints concerning the *environment and technology* of the system.

- Platform
- Technology to be used

Constraints concerning the *project plan and development methods*

- Development process (methodology) to be used
- Cost and delivery date
  - Often put in contract or project plan instead

31



Constraints

Constraint restrict how the requirements are to be implemented.

- **Interface Requirements.**
  How external interfaces with other systems must be done.
- **Communication Interfaces.**
  The networks and protocols to be used.
- **Hardware Interfaces.**
  The computer hardware the software is to execute on.
- **Software Interfaces.**
  How the software should be compatible with other software: applications, compilers, operating systems, programming languages, database management systems.
- **User Interfaces.**
  Style, format, messages

32

Requirements on Requirements (1)

Each individual requirement should be

- **Important/necessary** for the solution of the current problem
- **Unique**
- **Unambiguous**
- **Logically consistent**
- **Not over-constrain the design** of the system
- **Atomic:** not consist of multiple separate requirements

Requirements on Requirements (2)

The set of requirements together should be:

- **Complete**
- Expressed using a **clear and consistent notation**
  - at the same level of detail
- Without duplication

Requirements on Requirements (3)

S **Specific**
To-the-point, precise

M **Measurable**
Quantifiable and verifiable

A **Acceptable** (to the stakeholders)
Accessible, understandable (for the user)
Achievable (technically/planning/economically)

R **Realistic**
Deducible to the real business drivers

T **Testable**

Let’s consider

- “All communication between client and server is secure”
- “It is easy to extend”
- “The system should respond quickly
- “The user should not have to wait more than a few second …”
- “Determine solution within 0.3 sec”
- “The system should …”
- “The syst… can h…
- “The system can handle 100 co… rent …
- “The system should be state-of-the-art …”

not measurable

not precise

vague:to what?

attainable

subjective

time-dependent;
means something else tomorrow

doing what?

Requirements Prioritization

The Cost of Traditional BRUF

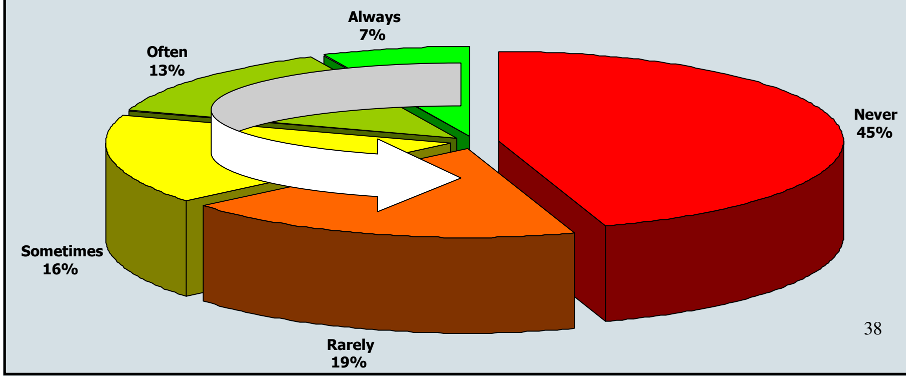
Big Requirements Up Front

Source: Jim Johnson of the Standish Group, Keynote Speech XP 2002

Pie chart shows percentage of functionality used by stakeholders

“Successful” Projects Still Have Significant Waste

Pareto-rule applies: 20% of functionality delivers 80% of value



Prioritizing Requirements

- MIL STD:
  - Must have, will have, may have
- RUP: MoSCoW

  Must have
  Should have
  Could have
  Won’t have

Criteria: indicate importance

Alternative criteria: volitility, cost to realize, risk, ..

Cost-Value Prioritization of Requirements

Motivation for Prioritization:

- Focus development effort
  - Allocate resources based on importance
- Make trade-offs between conflicting goals, such as quality, cost and time-to-market

Cost-Value Prioritization of Requirements

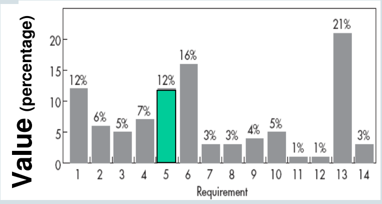
Process:

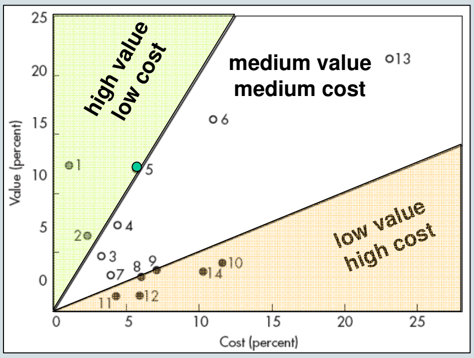
1. Review **requirements** for **clarity** and **completeness** (by Requirements Engineers)
2. Assess **relative value** of requirements in pair wise manner (Customers and users)
3. Assess **relative cost** of realizing requirements in pair wise manner (by experienced SW Engineers)
4. Calculate (value, cost)-pairs (using AHP*)
5. Plot requirements as (value, cost)-pairs
6. Prioritize

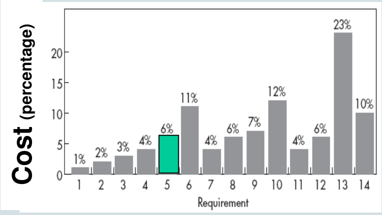
* Analytic Hierarchy Process

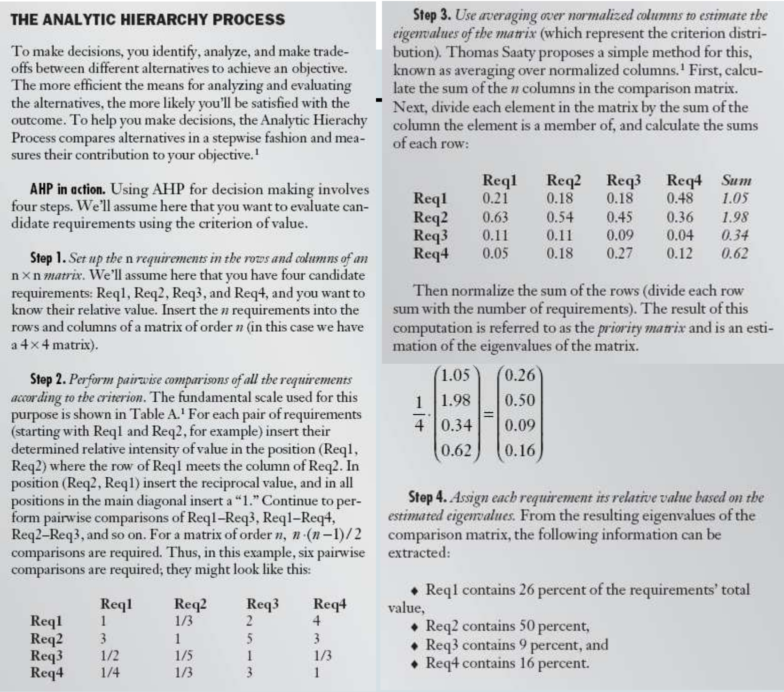
Requirements Prioritization Example

- 14 Requirements









THE ANALYTIC HIERARCHY PROCESS

To make decisions, you identify, analyze, and make tradeoffs between different alternatives to achieve an objective. The more efficient the means for analyzing and evaluating the alternatives, the more likely you'll be satisfied with the outcome. To help you make decisions, the Analytic Hierachy Process compares alternatives in a stepwise fashion and measures their contribution to your objective.[1]

AHP in action. Using AHP for decision making involves four steps. We'll assume here that you want to evaluate candidate requirements using the criterion of value.

Step 1. *Set up the* n *requirements in the rows and columns of an* $n \times n$ *matrix.* We'll assume here that you have four candidate requirements: Req1, Req2, Req3, and Req4, and you want to know their relative value. Insert the n requirements into the rows and columns of a matrix of order n (in this case we have a 4×4 matrix).

Step 2. *Perform pairwise comparisons of all the requirements according to the criterion.* The fundamental scale used for this purpose is shown in Table A.[1] For each pair of requirements (starting with Req1 and Req2, for example) insert their determined relative intensity of value in the position (Req1, Req2) where the row of Req1 meets the column of Req2. In position (Req2, Req1) insert the reciprocal value, and in all positions in the main diagonal insert a "1." Continue to perform pairwise comparisons of Req1–Req3, Req1–Req4, Req2–Req3, and so on. For a matrix of order n, $n \cdot (n-1)/2$ comparisons are required. Thus, in this example, six pairwise comparisons are required; they might look like this:

	Req1	Req2	Req3	Req4
Req1	1	1/3	2	4
Req2	3	1	5	3
Req3	1/2	1/5	1	1/3
Req4	1/4	1/3	3	1

Step 3. *Use averaging over normalized columns to estimate the eigenvalues of the matrix* (which represent the criterion distribution). Thomas Saaty proposes a simple method for this, known as averaging over normalized columns.[1] First, calculate the sum of the n columns in the comparison matrix. Next, divide each element in the matrix by the sum of the column the element is a member of, and calculate the sums of each row:

	Req1	Req2	Req3	Req4	*Sum*
Req1	0.21	0.18	0.18	0.48	*1.05*
Req2	0.63	0.54	0.45	0.36	*1.98*
Req3	0.11	0.11	0.09	0.04	*0.34*
Req4	0.05	0.18	0.27	0.12	*0.62*

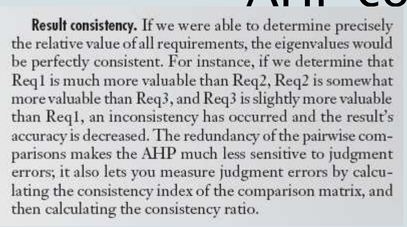
Then normalize the sum of the rows (divide each row sum with the number of requirements). The result of this computation is referred to as the *priority matrix* and is an estimation of the eigenvalues of the matrix.

$$\frac{1}{4} \cdot \begin{pmatrix} 1.05 \\ 1.98 \\ 0.34 \\ 0.62 \end{pmatrix} = \begin{pmatrix} 0.26 \\ 0.50 \\ 0.09 \\ 0.16 \end{pmatrix}$$

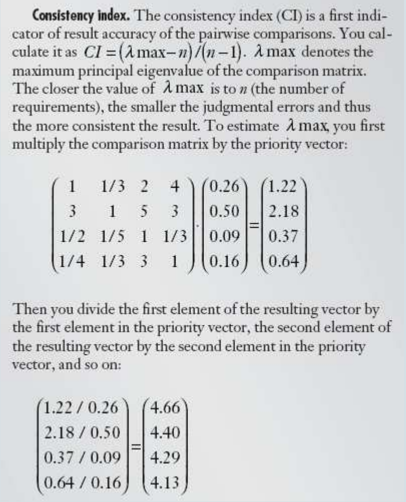
Step 4. *Assign each requirement its relative value based on the estimated eigenvalues.* From the resulting eigenvalues of the comparison matrix, the following information can be extracted:

- Req1 contains 26 percent of the requirements' total value,
- Req2 contains 50 percent,
- Req3 contains 9 percent, and
- Req4 contains 16 percent.

AHP consistency



Result consistency. If we were able to determine precisely the relative value of all requirements, the eigenvalues would be perfectly consistent. For instance, if we determine that Req1 is much more valuable than Req2, Req2 is somewhat more valuable than Req3, and Req3 is slightly more valuable than Req1, an inconsistency has occurred and the result's accuracy is decreased. The redundancy of the pairwise comparisons makes the AHP much less sensitive to judgment errors; it also lets you measure judgment errors by calculating the consistency index of the comparison matrix, and then calculating the consistency ratio.

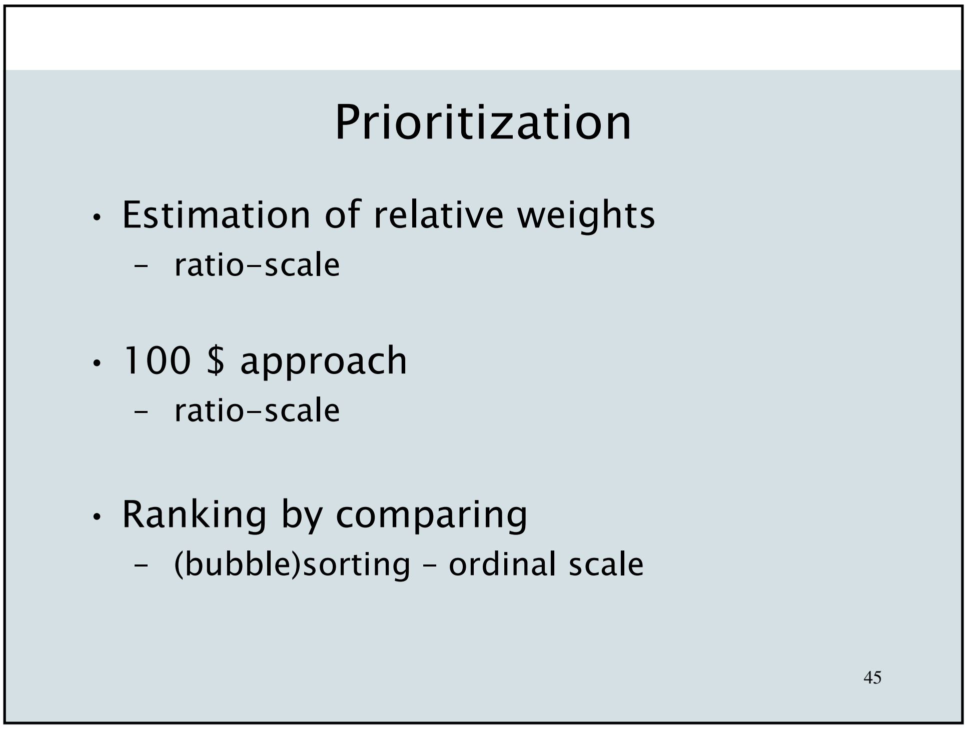


Consistency index. The consistency index (CI) is a first indicator of result accuracy of the pairwise comparisons. You calculate it as $CI = (\lambda\max - n)/(n-1)$. $\lambda\max$ denotes the maximum principal eigenvalue of the comparison matrix. The closer the value of $\lambda\max$ is to n (the number of requirements), the smaller the judgmental errors and thus the more consistent the result. To estimate $\lambda\max$, you first multiply the comparison matrix by the priority vector:

$$\begin{pmatrix} 1 & 1/3 & 2 & 4 \\ 3 & 1 & 5 & 3 \\ 1/2 & 1/5 & 1 & 1/3 \\ 1/4 & 1/3 & 3 & 1 \end{pmatrix} \cdot \begin{pmatrix} 0.26 \\ 0.50 \\ 0.09 \\ 0.16 \end{pmatrix} = \begin{pmatrix} 1.22 \\ 2.18 \\ 0.37 \\ 0.64 \end{pmatrix}$$

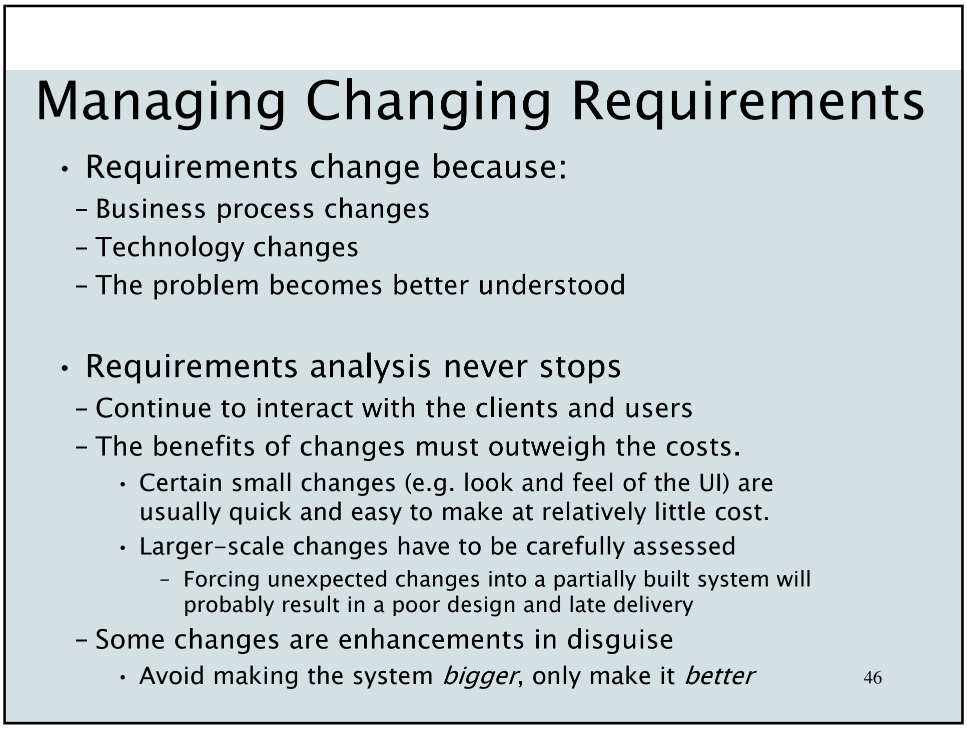
Then you divide the first element of the resulting vector by the first element in the priority vector, the second element of the resulting vector by the second element in the priority vector, and so on:

$$\begin{pmatrix} 1.22 / 0.26 \\ 2.18 / 0.50 \\ 0.37 / 0.09 \\ 0.64 / 0.16 \end{pmatrix} = \begin{pmatrix} 4.66 \\ 4.40 \\ 4.29 \\ 4.13 \end{pmatrix}$$



Prioritization

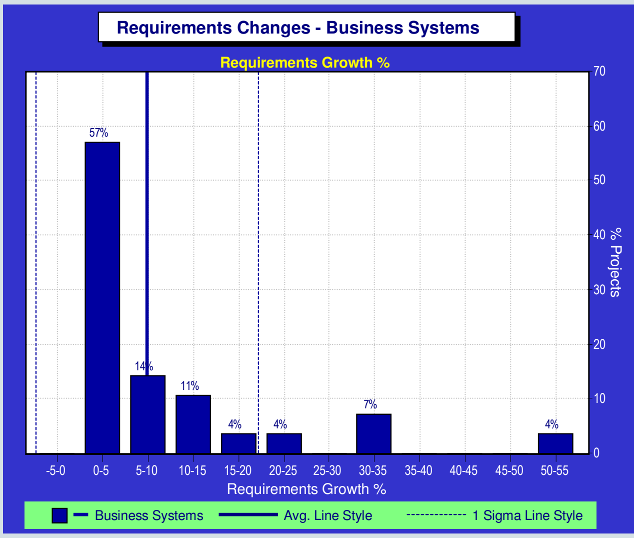
- Estimation of relative weights
  - ratio-scale
- 100 $ approach
  - ratio-scale
- Ranking by comparing
  - (bubble)sorting – ordinal scale



Managing Changing Requirements

- Requirements change because:
  - Business process changes
  - Technology changes
  - The problem becomes better understood
- Requirements analysis never stops
  - Continue to interact with the clients and users
  - The benefits of changes must outweigh the costs.
    - Certain small changes (e.g. look and feel of the UI) are usually quick and easy to make at relatively little cost.
    - Larger-scale changes have to be carefully assessed
      - Forcing unexpected changes into a partially built system will probably result in a poor design and late delivery
  - Some changes are enhancements in disguise
    - Avoid making the system *bigger*, only make it *better*

Requirement Changes



Data from 23 projects of the Indian IT industry for overseas customers (Jan 2000)

Traceability

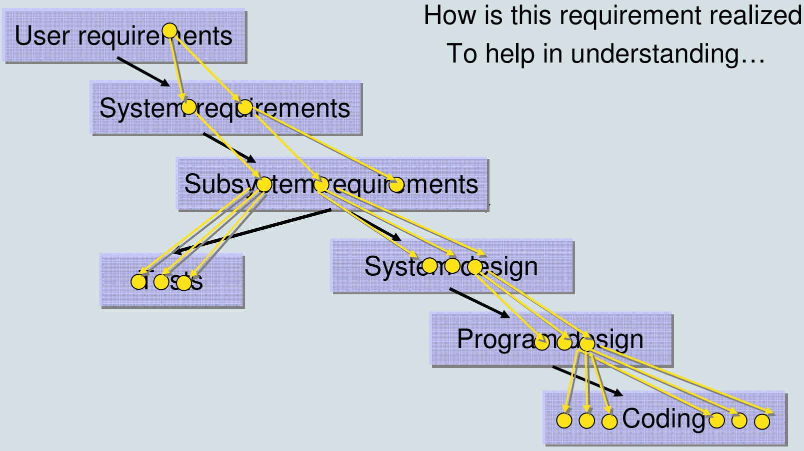
- From req to arch choices/features
- From features to req's

- Check
  - Completeness of system
  - Analyze impact of changing requirements

Forward Traceability

How is this requirement realized?

To help in understanding…

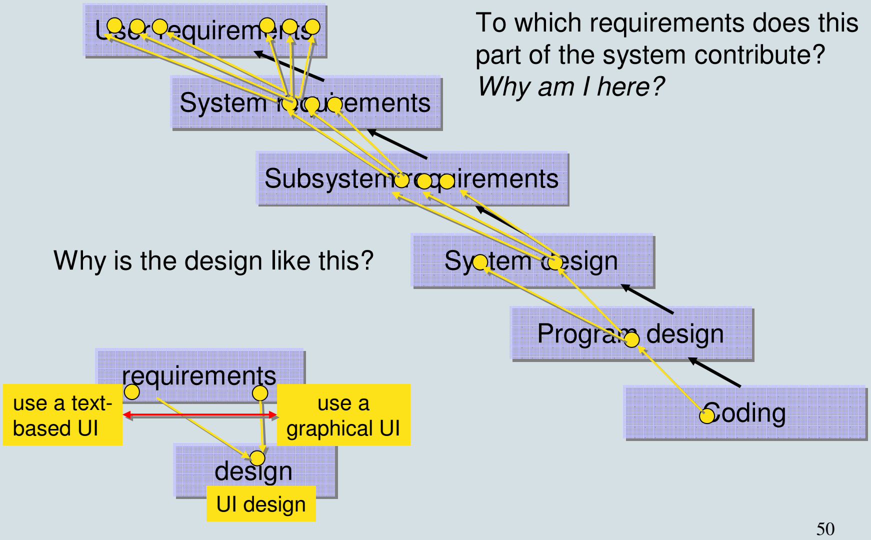


Backward Traceability

To which requirements does this part of the system contribute?

Why am I here?

Why is the design like this?



Why Traceability?

- Accountability: where did this requirement come from?
  - The source of a requirements may be needed for clarification, negotiation, conflict resolution
- Matching solution to problem
  - For monitoring completeness of system:
    - Acceptance test: are all requirements addressed?
    - are there unnecessary requirements/features?
- Analyze impact of changes (in req'mt's / design decions)
  - Change request: What parts of the design need to change, if a requirement changes?
- Reuse of requirements

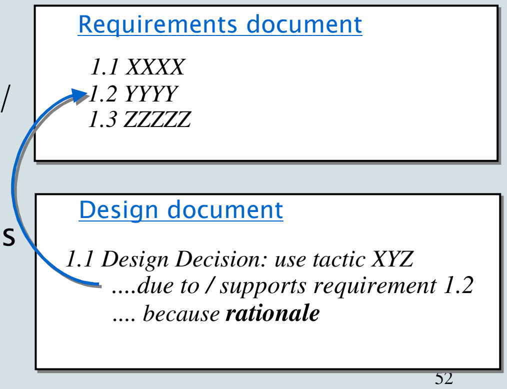
51

How Traceability: Hyperlinks

design document and requirements document contains hyperlinks to each other

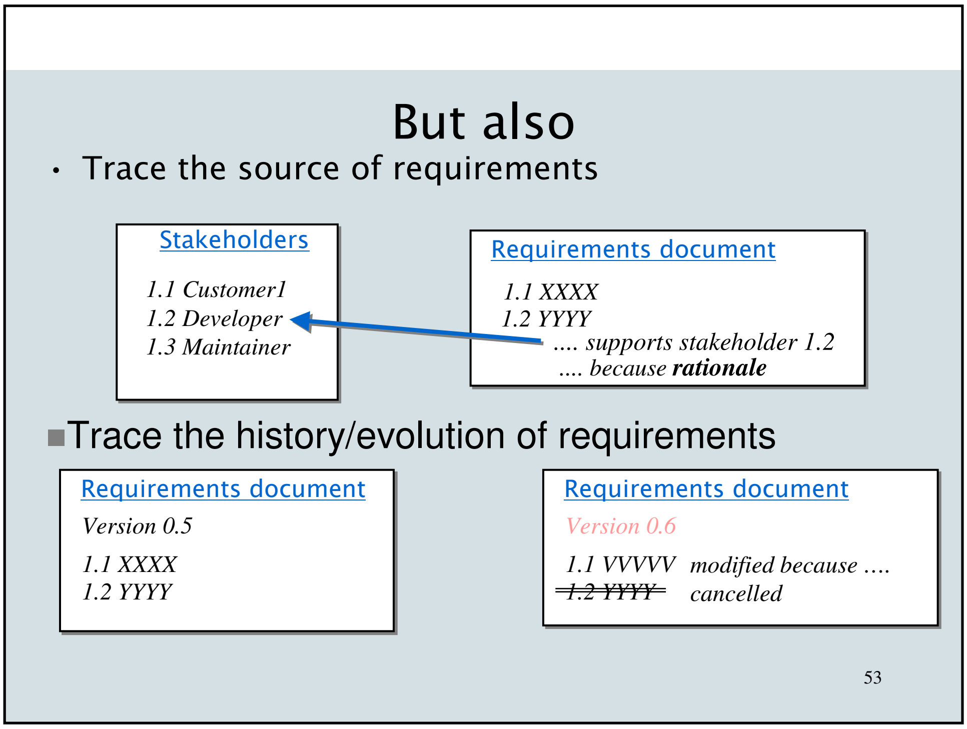
Typical use:
 interactive exploring / browsing req.docs

Using .html documents & browsers



Requirements document
1.1 XXXX
1.2 YYYY
1.3 ZZZZZ
Design document
1.1 Design Decision: use tactic XYZ
....due to / supports requirement 1.2
.... because **rationale**

52



But also

- Trace the source of requirements

Stakeholders

1.1 Customer1
1.2 Developer
1.3 Maintainer

Requirements document

1.1 XXXX
1.2 YYYY
.... supports stakeholder 1.2
.... because ***rationale***

- Trace the history/evolution of requirements

Requirements document

Version 0.5

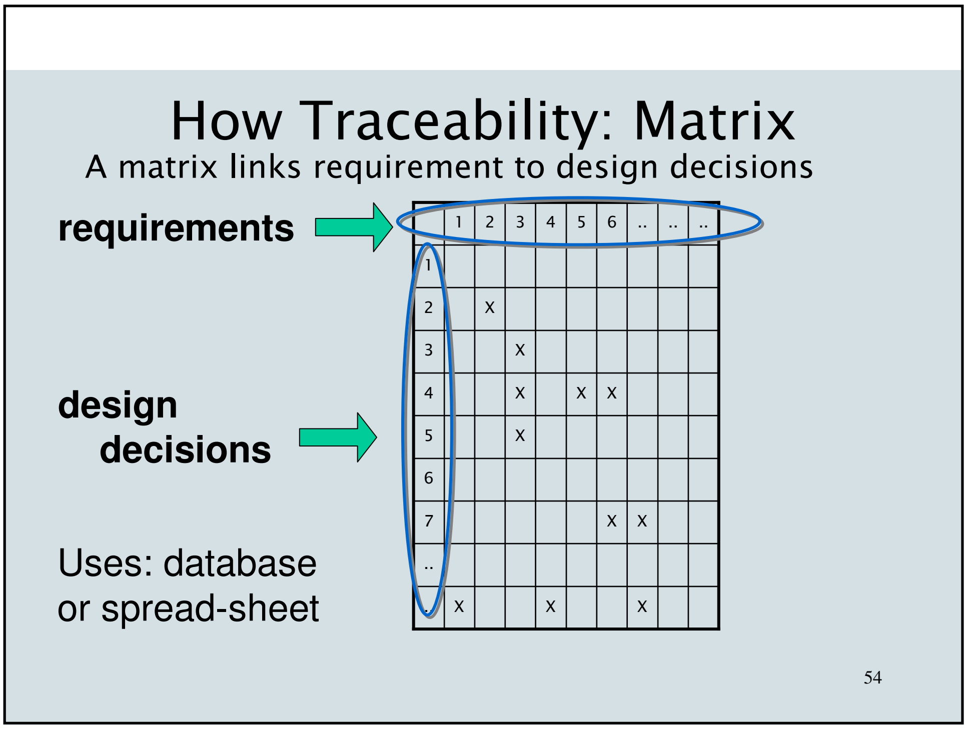
1.1 XXXX
1.2 YYYY

Requirements document

Version 0.6

1.1 VVVVV *modified because ....*
~~*1.2 YYYY*~~ *cancelled*

53



How Traceability: Matrix

A matrix links requirement to design decisions

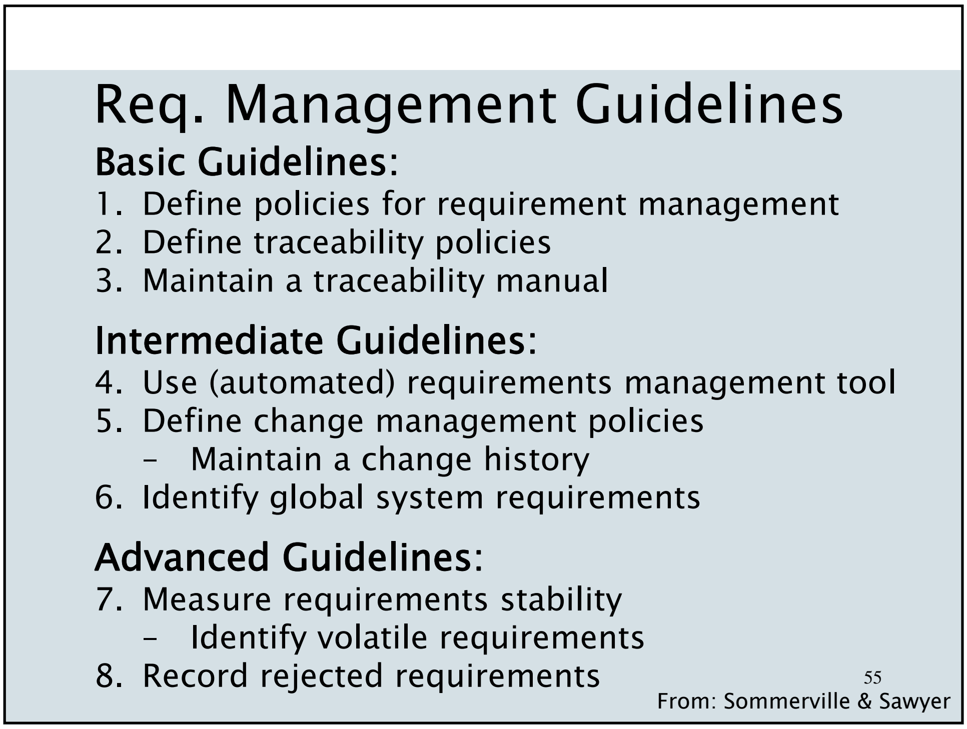
requirements

design decisions

Uses: database or spread-sheet

	1	2	3	4	5	6	..	..	..
1									
2		X							
3			X						
4			X		X	X			
5			X						
6									
7						X	X		
..									
.	X			X			X		

54



Req. Management Guidelines

Basic Guidelines:

1. Define policies for requirement management
2. Define traceability policies
3. Maintain a traceability manual

Intermediate Guidelines:

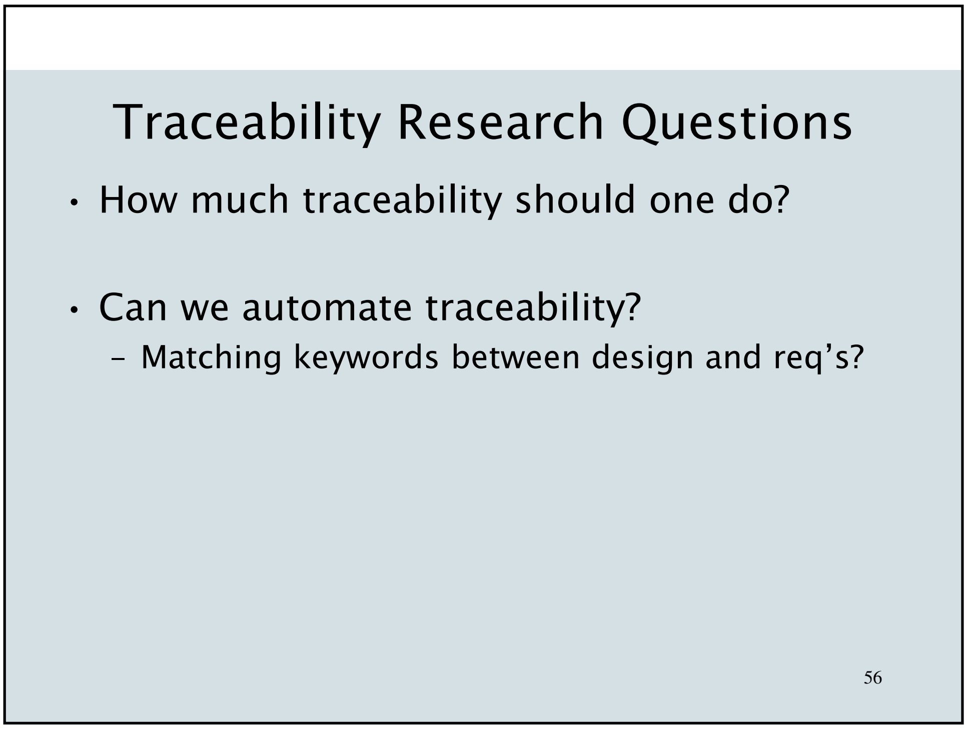
4. Use (automated) requirements management tool
5. Define change management policies
   - Maintain a change history
6. Identify global system requirements

Advanced Guidelines:

7. Measure requirements stability
   - Identify volatile requirements
8. Record rejected requirements

55

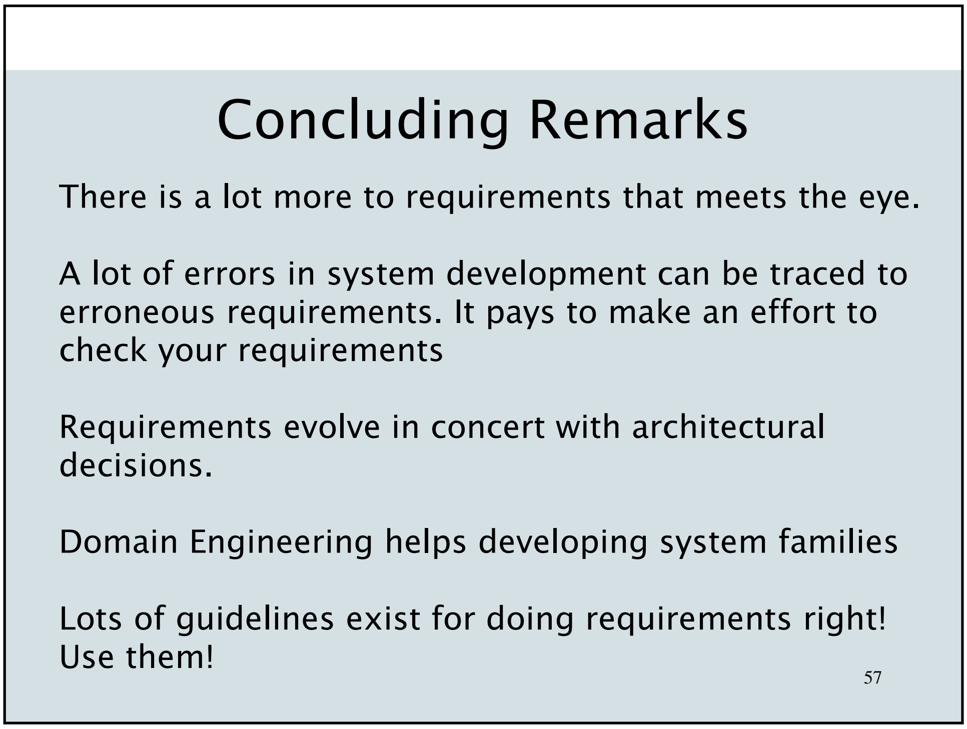
From: Sommerville & Sawyer



Traceability Research Questions

- How much traceability should one do?
- Can we automate traceability?
  - Matching keywords between design and req's?

56



Concluding Remarks

There is a lot more to requirements that meets the eye.

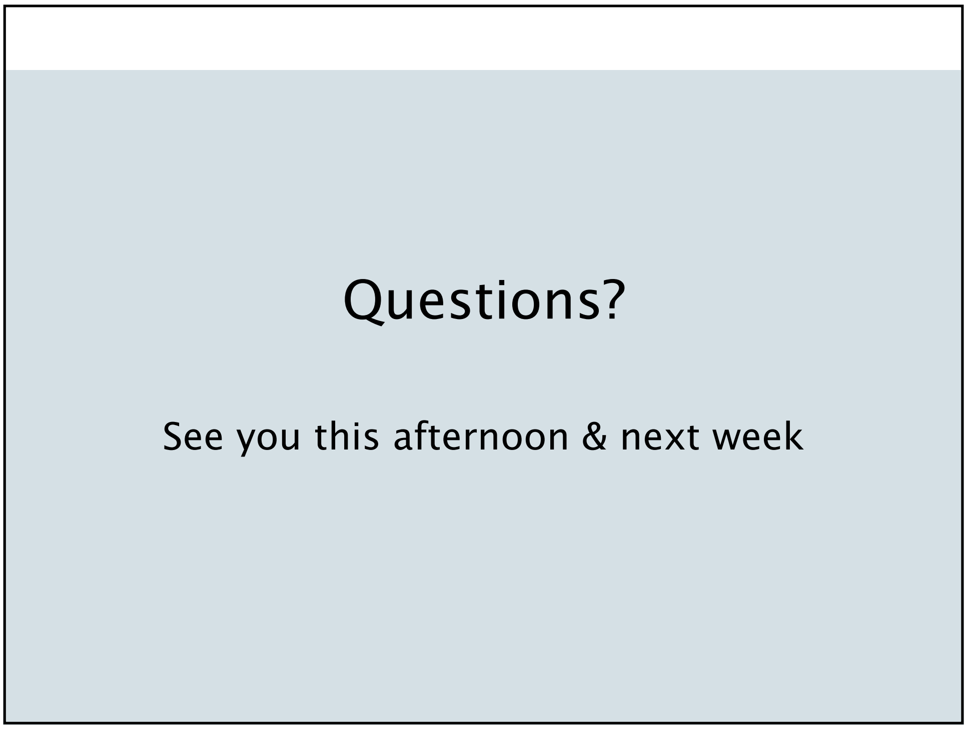
A lot of errors in system development can be traced to erroneous requirements. It pays to make an effort to check your requirements

Requirements evolve in concert with architectural decisions.

Domain Engineering helps developing system families

Lots of guidelines exist for doing requirements right! Use them!

57



Questions?

See you this afternoon & next week

[Gacek et al 1995] present the results of a survey of people who are somehow involved in software development processes (developers, customers, maintainers, aquisitioners, etc.).

There they found that, with respect to architects, the three major concerns were

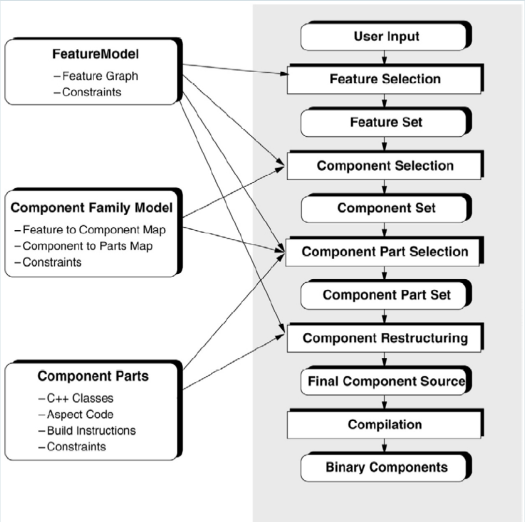
“1) requirements traceability;

2) support of tradeoff analyses; and

3) completeness, consistency of architecture.”

Gacek, C., Abd-Allah, A., Clark, B.K., and Boehm, B. (1995)

“On the Definition of Software System Architecture,” in *Proceedings of the First International Workshop on Architectures for Software Systems - 17th ICSE*, Seattle, 24-25 April 1995, pp. 85-95.



FeatureModel
- Feature Graph
- Constraints

Component Family Model
- Feature to Component Map
- Component to Parts Map
- Constraints

Component Parts
- C++ Classes
- Aspect Code
- Build Instructions
- Constraints

User Input

Feature Selection

Feature Set

Component Selection

Component Set

Component Part Selection

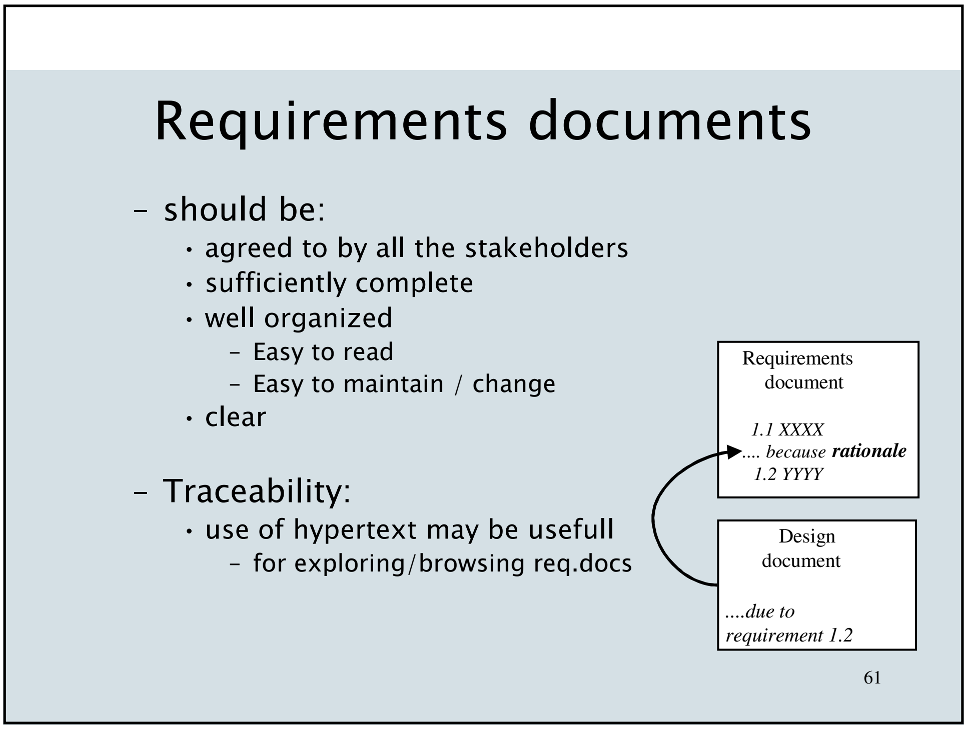
Component Part Set

Component Restructuring

Final Component Source

Compilation

Binary Components



Requirements documents

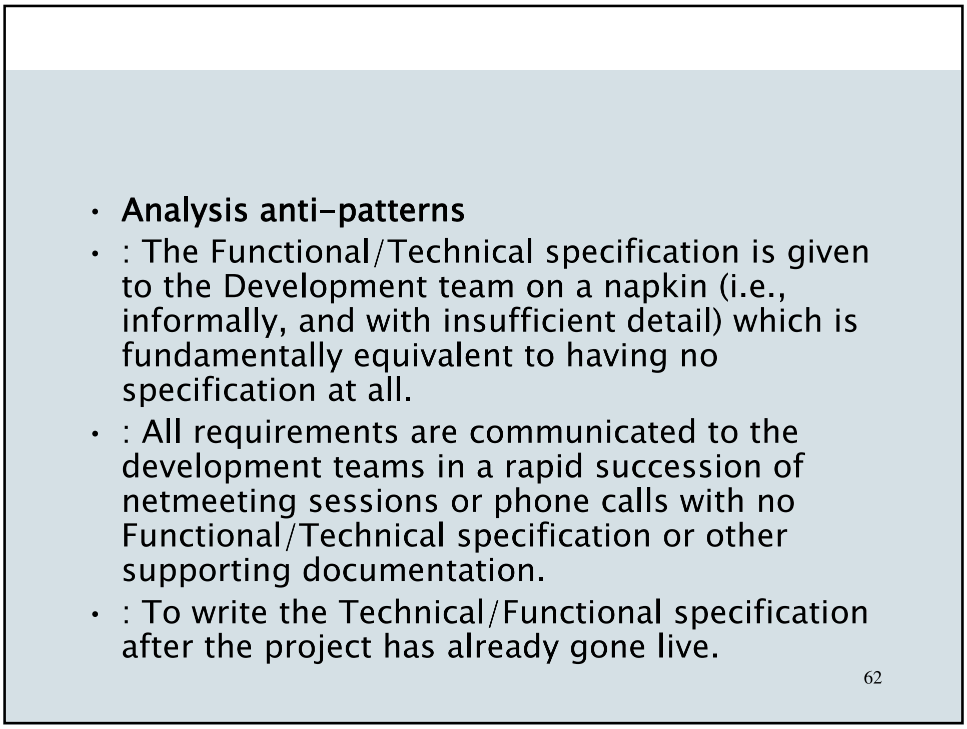
- should be:
  - agreed to by all the stakeholders
  - sufficiently complete
  - well organized
    - Easy to read
    - Easy to maintain / change
  - clear
- Traceability:
  - use of hypertext may be usefull
    - for exploring/browsing req.docs

Requirements document

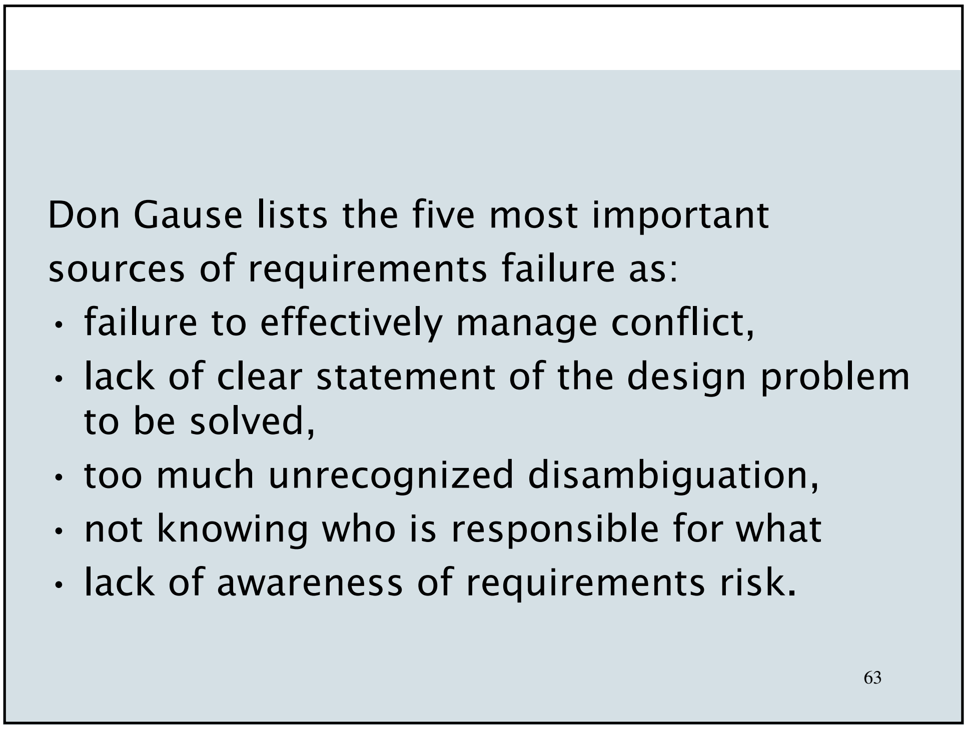
1.1 XXXX
.... because ***rationale***
1.2 YYYY

Design document

....due to requirement 1.2

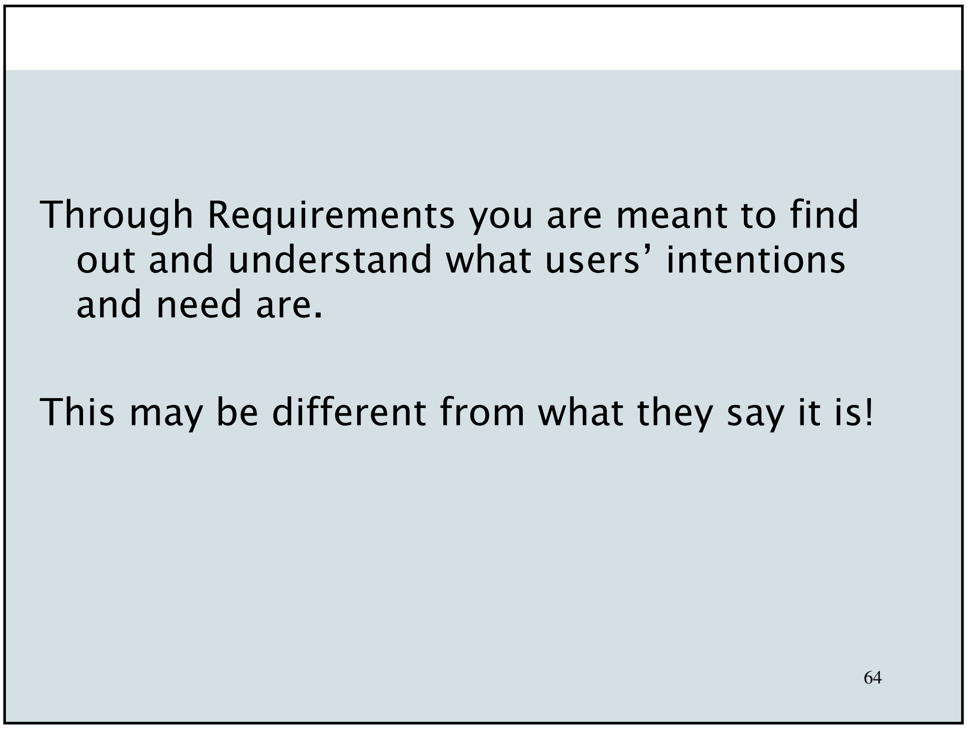


- **Analysis anti-patterns**
- : The Functional/Technical specification is given to the Development team on a napkin (i.e., informally, and with insufficient detail) which is fundamentally equivalent to having no specification at all.
- : All requirements are communicated to the development teams in a rapid succession of netmeeting sessions or phone calls with no Functional/Technical specification or other supporting documentation.
- : To write the Technical/Functional specification after the project has already gone live.



Don Gause lists the five most important sources of requirements failure as:

- failure to effectively manage conflict,
- lack of clear statement of the design problem to be solved,
- too much unrecognized disambiguation,
- not knowing who is responsible for what
- lack of awareness of requirements risk.



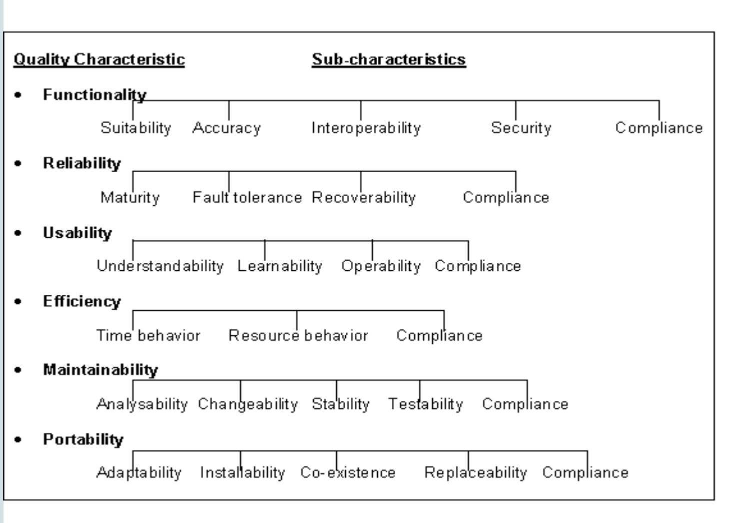
Through Requirements you are meant to find out and understand what users' intentions and need are.

This may be different from what they say it is!

Ezelsbruggetje

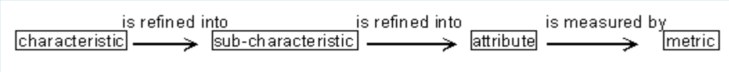
- Het woord is waarschijnlijk afkomstig van het feit dat de ezel maar een heel klein randje nodig heeft om snel op de plek van bestemming te komen; een plank over een sloot volstaat al. Het woord ezelsbrug is al heel oud en bestond in het Latijn al (*pons asinorum*).

- English translation welcome …

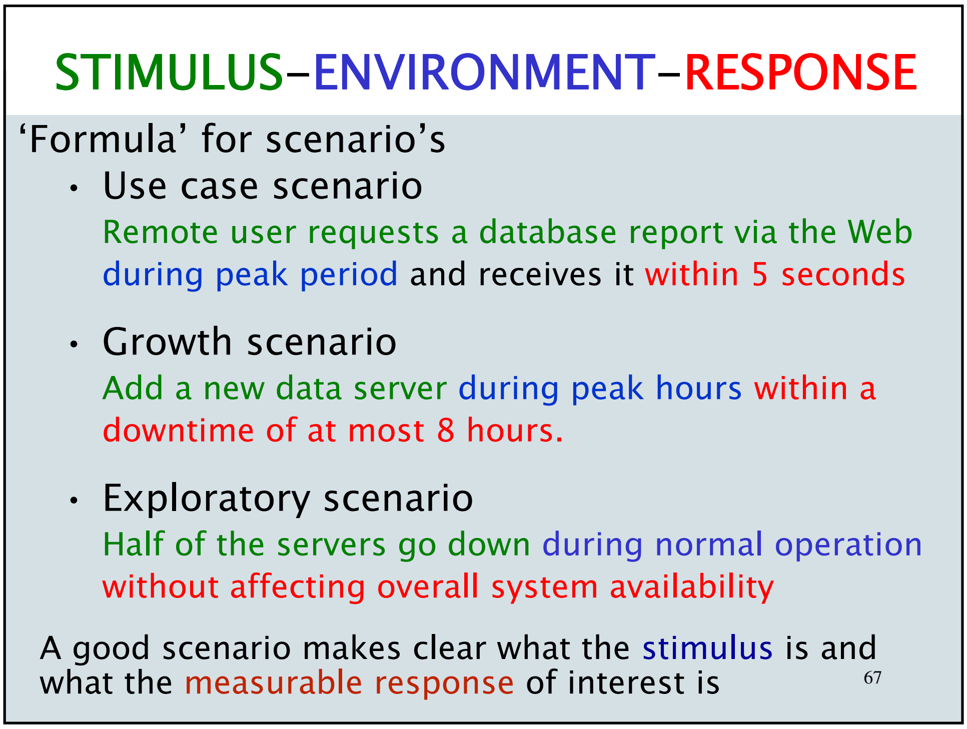


Quality Characteristic **Sub-characteristics**

- **Functionality**
  Suitability Accuracy Interoperability Security Compliance
- **Reliability**
  Maturity Fault tolerance Recoverability Compliance
- **Usability**
  Understandability Learnability Operability Compliance
- **Efficiency**
  Time behavior Resource behavior Compliance
- **Maintainability**
  Analysability Changeability Stability Testability Compliance
- **Portability**
  Adaptability Installability Co-existence Replaceability Compliance



characteristic → (is refined into) sub-characteristic → (is refined into) attribute → (is measured by) metric



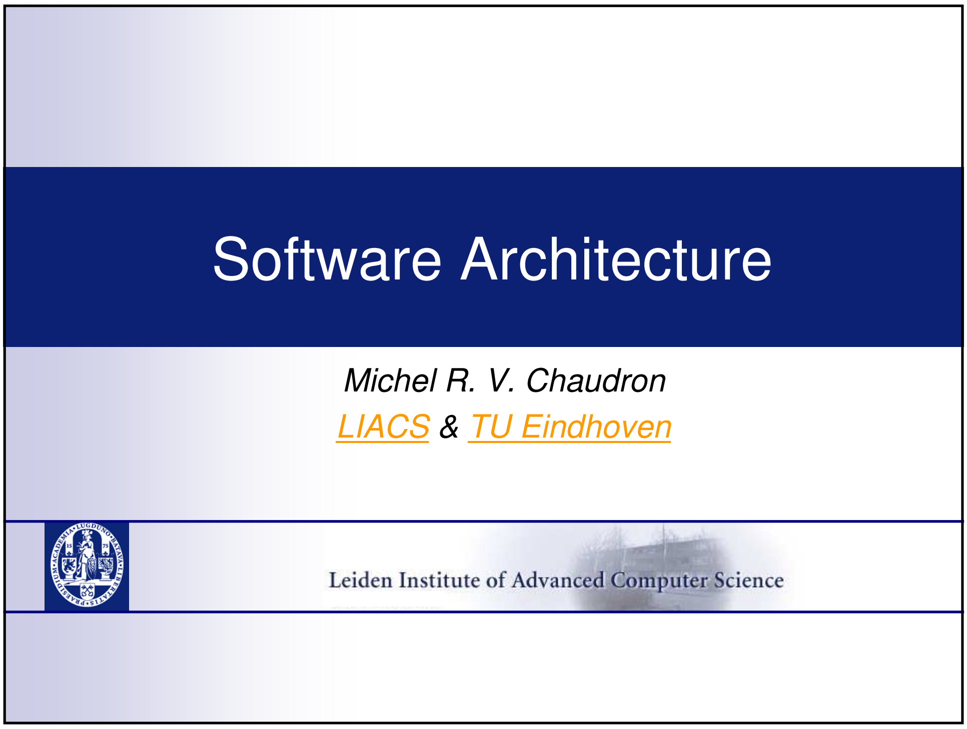
STIMULUS–ENVIRONMENT–RESPONSE

‘Formula’ for scenario’s

- Use case scenario
  Remote user requests a database report via the Web during peak period and receives it within 5 seconds
- Growth scenario
  Add a new data server during peak hours within a downtime of at most 8 hours.
- Exploratory scenario
  Half of the servers go down during normal operation without affecting overall system availability

A good scenario makes clear what the stimulus is and what the measurable response of interest is

67

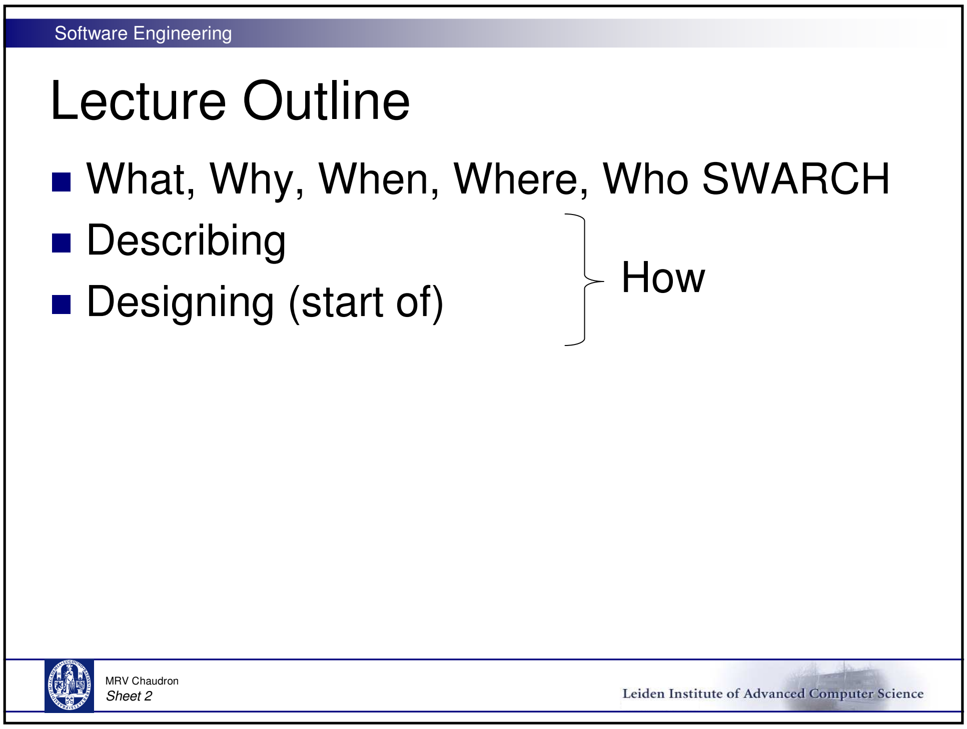


Software Architecture

Michel R. V. Chaudron

LIACS & TU Eindhoven

Leiden Institute of Advanced Computer Science



Software Engineering

Lecture Outline

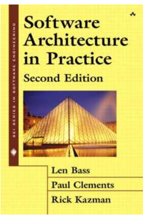
- What, Why, When, Where, Who SWARCH
- Describing
- Designing (start of)

} How

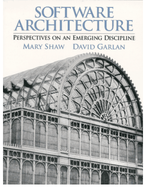
MRV Chaudron
Sheet 2

Leiden Institute of Advanced Computer Science

Software Architecture Books



- Software Architecture in Practice, **Second Edition**, L. Bass, P. Clements, R. Kazman, SEI Series in Software Engineering, Addison-Wesley, 2003



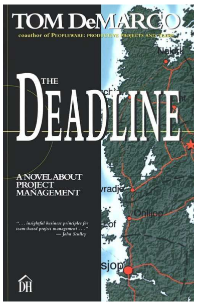
- Software Architecture: Perspectives on an Emerging Discipline, Mary Shaw, David Garlan, 242 pages, 1996, Prentice Hall

- Recommended Practice for Architectural Description, IEEE STD 1471-2000, 23 pages

Software Project Management

The Deadline
by Tom DeMarco

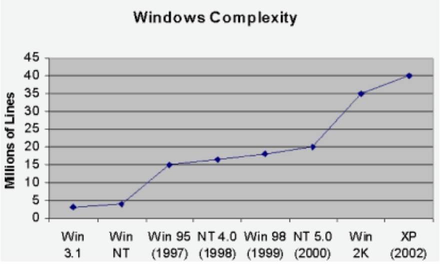
• easy & fun reading
• lots of lessons from practical experience

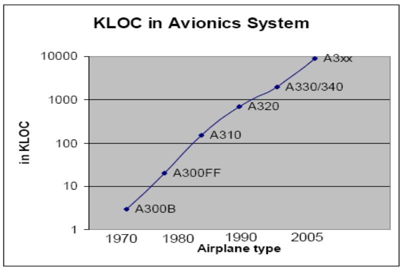


Also very though provoking:

Peopleware
by DeMarco & Lister,
Dorset House Publ., 2nd ed, 1999

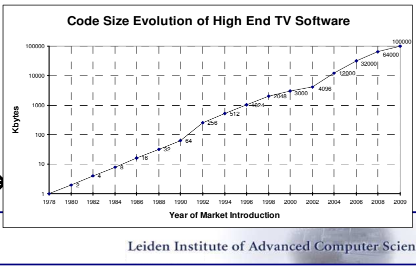
Increasing amount of software in systems





The amount of software increases 10 fold every 10 years.
Abstractions are needed.

Nb: logarithmic scale →



What is Software Architecture?

Classic Definitions 1

An architecture is the set of significant decisions about
- the organization of a software system,
- the selection of the structural elements and their interfaces by which the system is composed, together with their behaviour as specified in the collaborations among those elements,
- the composition of these structural and behavioural elements into progressively larger subsystems,
- the architectural style that guides this organization

The UML Modeling Language User Guide, Addison-Wesley, 1999
Booch, Rumbaugh, and Jacobson

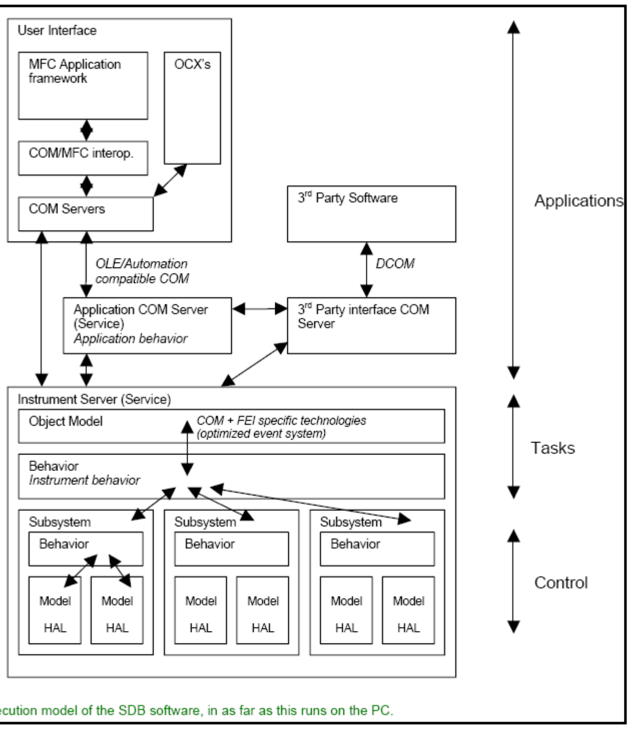
What is Software Architecture?

Classic Definitions 2

The structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

David Garlan and Dewayne Perry
April 1995 IEEE Transactions on Software Engineering

Example: FEI electron microscopes



Full Architecture Description

Figure 4: Execution model of the SDB software, in as far as this runs on the PC.

Example: FEI electron microscopes

Full Architecture Description

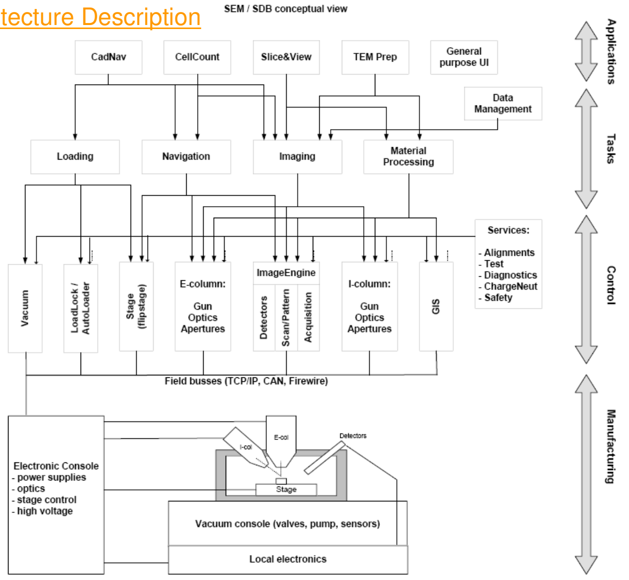
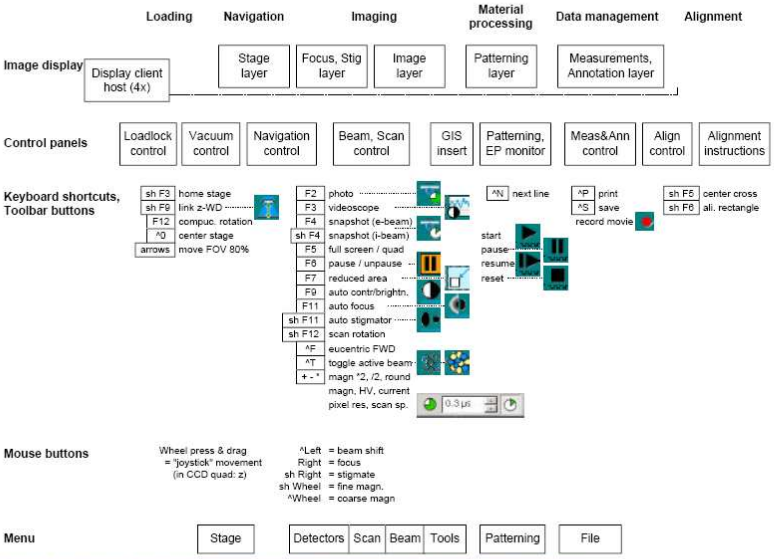


Figure 1: the SDB family system decomposition and interrelationships on the different levels of abstraction.



sh F3	home stage
sh F9	link z-WD
F12	compuc. rotation
^0	center stage
arrows	move FOV 80%

F2	photo
F3	videoscope
F4	snapshot (e-beam)
sh F4	snapshot (i-beam)
F5	full screen / quad
F6	pause / unpause
F7	reduced area
F9	auto contr/brightn.
F11	auto focus
sh F11	auto stigmator
sh F12	scan rotation
^F	eucentric FWD
^T	toggle active beam
+ - *	magn *2, /2, round
	magn, HV, current
	pixel res, scan sp.

^N	next line

start
pause
resume
reset

^P	print
^S	save

record movie

sh F5	center cross
sh F6	ali. rectangle

Wheel press & drag
= "joystick" movement
(in CCD quad: z)

^Left	= beam shift
Right	= focus
sh Right	= stigmate
sh Wheel	= fine magn.
^Wheel	= coarse magn

Figure 3: Functions offered through the user interface, arranged according to task and control means

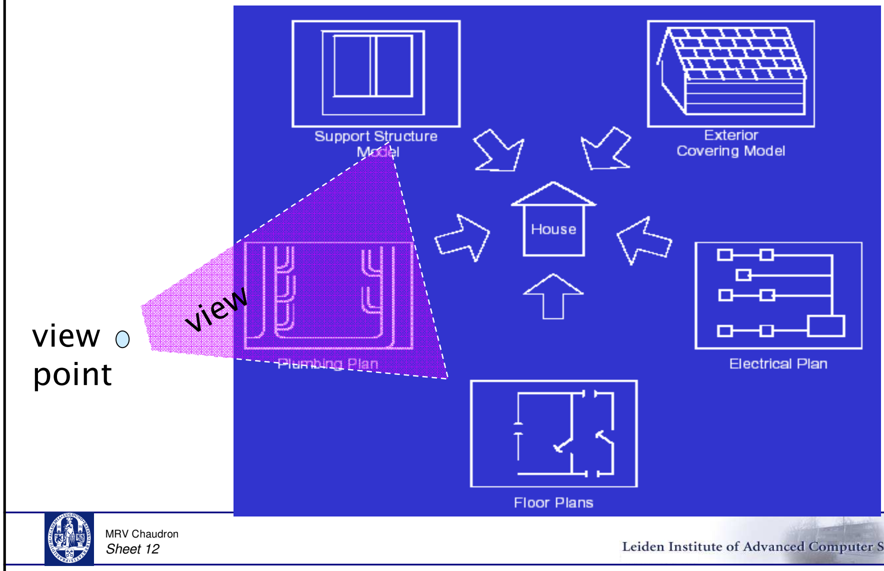
MC2

Contents of a good architectural model

- A system’s architecture will often be expressed in terms of several different *views*
  - □ The logical breakdown into subsystems
    - □ conceptual abstract view
    - □ functional decomposition
    - □ responsibility distribution
    - The interfaces among the subsystems
  - □ The dynamics of the interaction among components
  - □ The data that will be shared among the components
  - □ The components that will exist at run time, and the machines or devices on which they will be located

Slide by Lethbridge and Laganiere

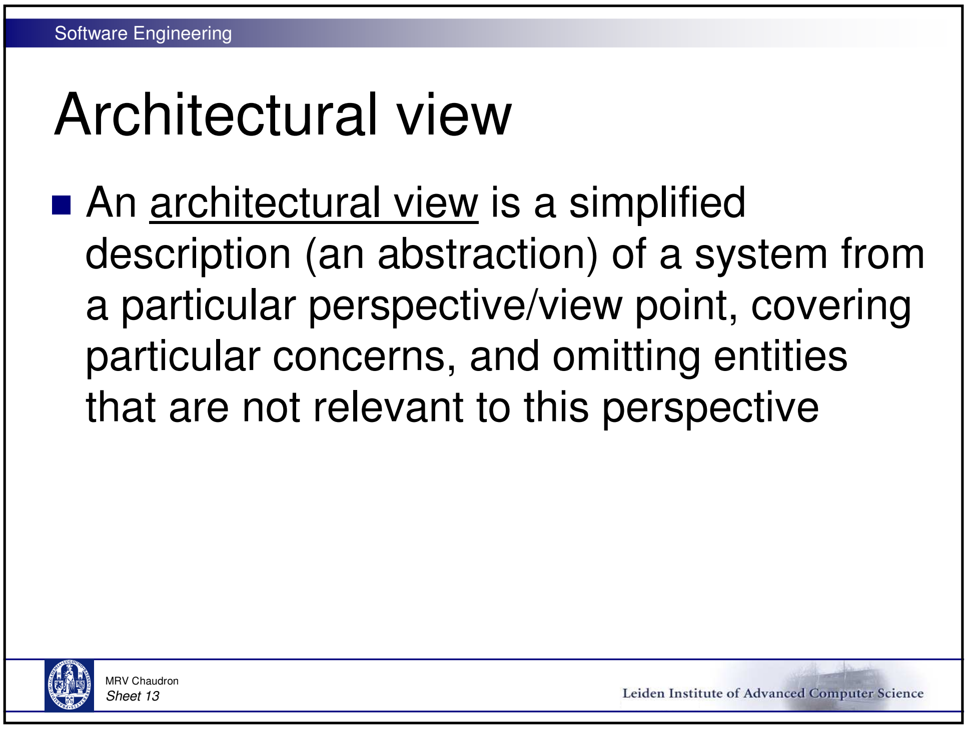
Viewpoints & views



Slide 11

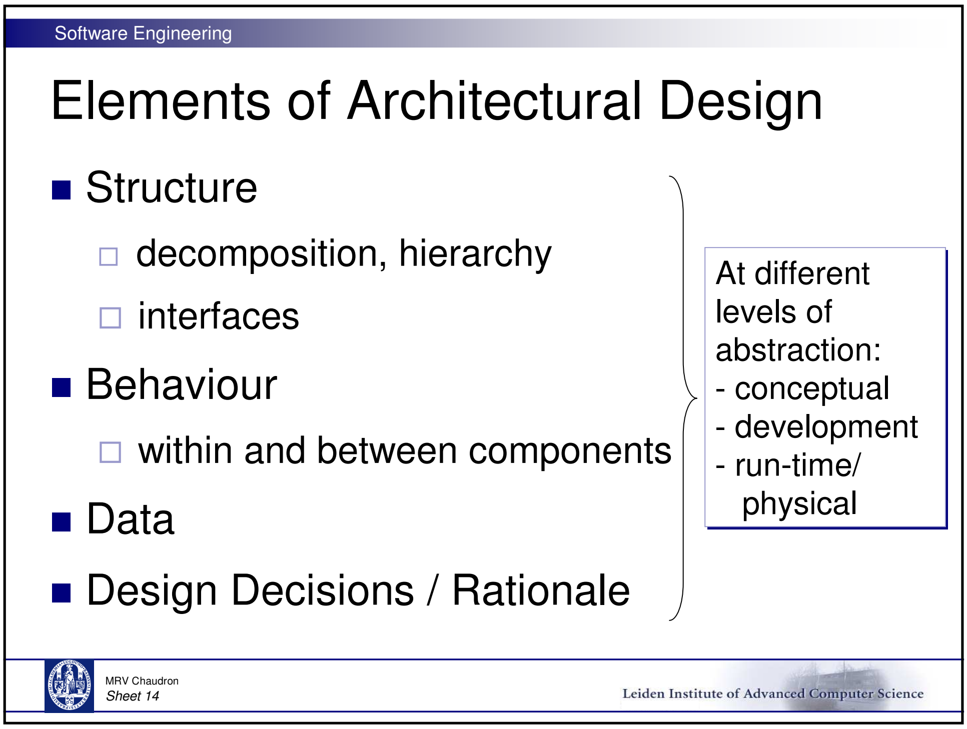
MC2 hierarchy
- itself a view
- may apply to different views

Michel Chaudron; 27-2-2008



Architectural view

- An architectural view is a simplified description (an abstraction) of a system from a particular perspective/view point, covering particular concerns, and omitting entities that are not relevant to this perspective



Elements of Architectural Design

- Structure
  - □ decomposition, hierarchy
  - □ interfaces
- Behaviour
  - □ within and between components
- Data
- Design Decisions / Rationale

At different levels of abstraction:
- conceptual
- development
- run-time/ physical

Making design decisions

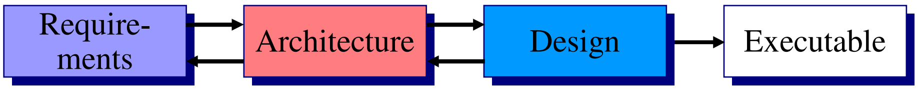
- To make each design decision, the software engineer uses:
  - □ Knowledge of
    - the application domain
    - the requirements
    - the design as created so far
    - the technology available
    - software design principles and 'best practices'
    - what has worked well in the past

Combination of top-down and bottom-up

Slide by Lethbridge and Laganiere

Positioning Architecture

The question: *The answer:* *Implementation:* *Deployment:*



- Features
- Use cases
- Dependability
  Timing
  Reliability
  Security
- Quality
- Standards
- Etc.

- HL-Design
  Components
  Interfaces
  Interactions
- Styles
- Constraints
- Guidelines
- Reuse
- Etc.

- Decomposition
- Algorithms
- Data structures
- Distribution
- Scheduling
- Recovery
- Language
- Encryption
- Etc.

- Memory allocation
- Dynamic Instantiation
- Call stacks
- Garbage collection
- Machine code
- Etc.

Architecturally Significant Elements

An architecturally significant element has a significant impact on the **structure, performance, robustness, scalability, maintainability** and **evolvability** of the system.

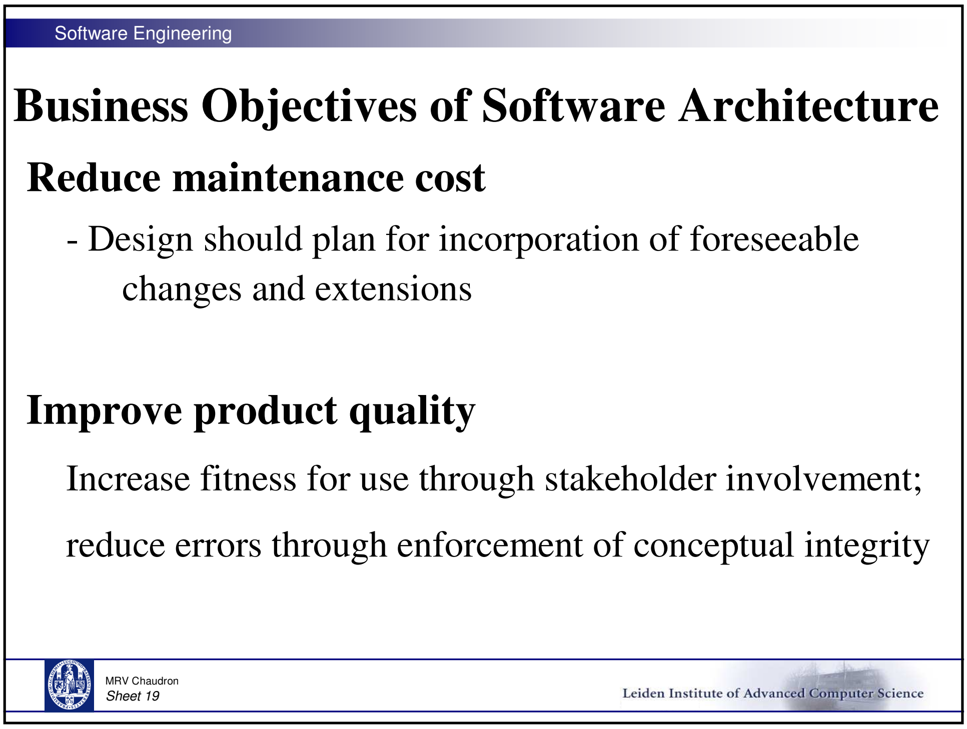
Business Objectives of Software Architecture

Reduce development cost

- improved communication between developers, and
- earlier assessment of design alternatives and assessment of system risks

Reduce time-to-market

- allowing concurrent development of different subsystems
- enabling reuse



Business Objectives of Software Architecture

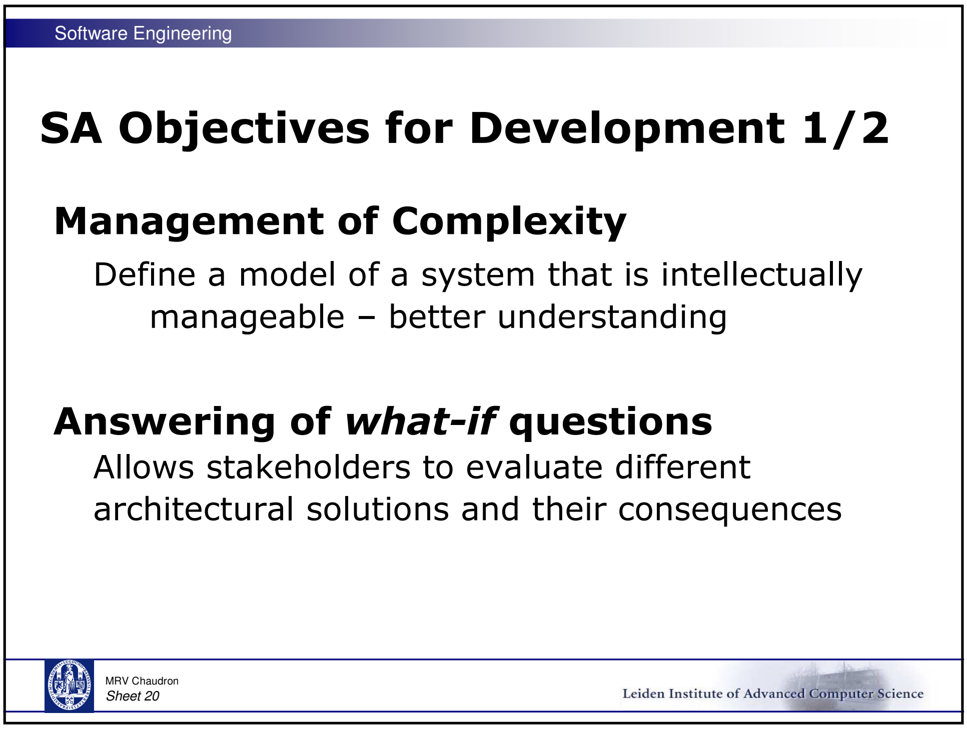
Reduce maintenance cost

- Design should plan for incorporation of foreseeable changes and extensions

Improve product quality

Increase fitness for use through stakeholder involvement;

reduce errors through enforcement of conceptual integrity



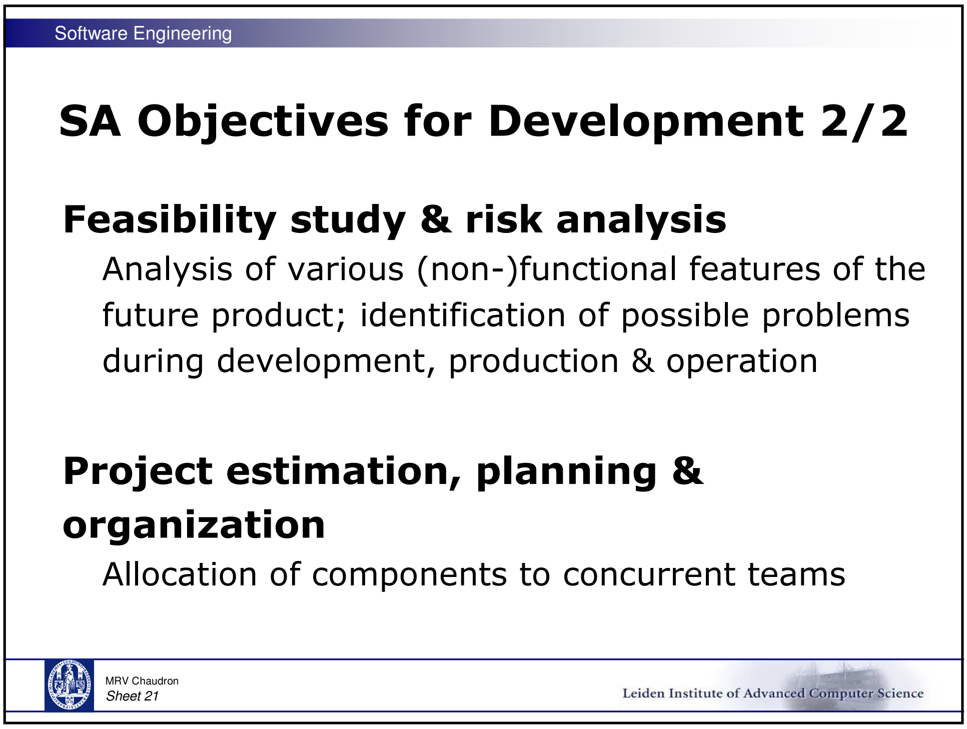
SA Objectives for Development 1/2

Management of Complexity

Define a model of a system that is intellectually manageable – better understanding

Answering of *what-if* questions

Allows stakeholders to evaluate different architectural solutions and their consequences



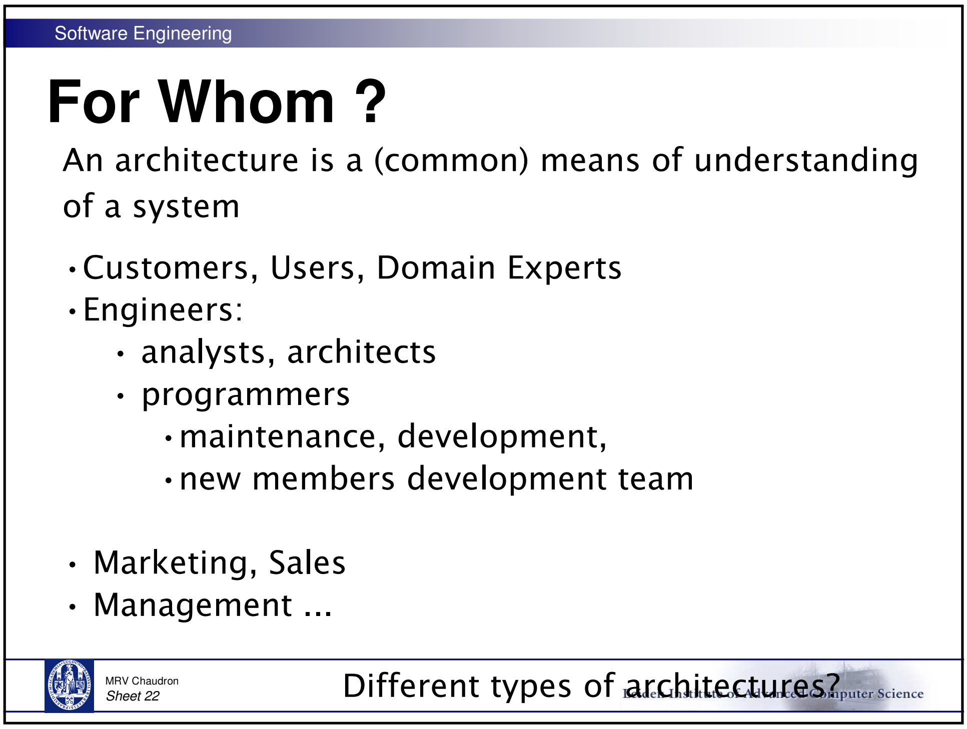
SA Objectives for Development 2/2

Feasibility study & risk analysis

Analysis of various (non-)functional features of the future product; identification of possible problems during development, production & operation

Project estimation, planning & organization

Allocation of components to concurrent teams



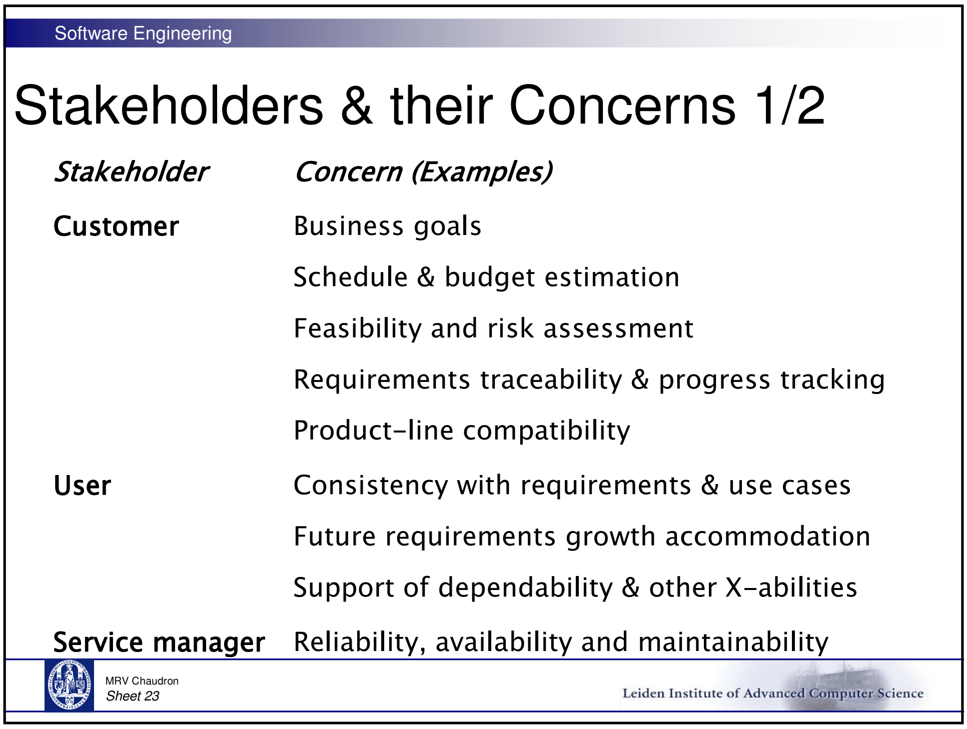
For Whom ?

An architecture is a (common) means of understanding of a system

- Customers, Users, Domain Experts
- Engineers:
  - analysts, architects
  - programmers
    - maintenance, development,
    - new members development team

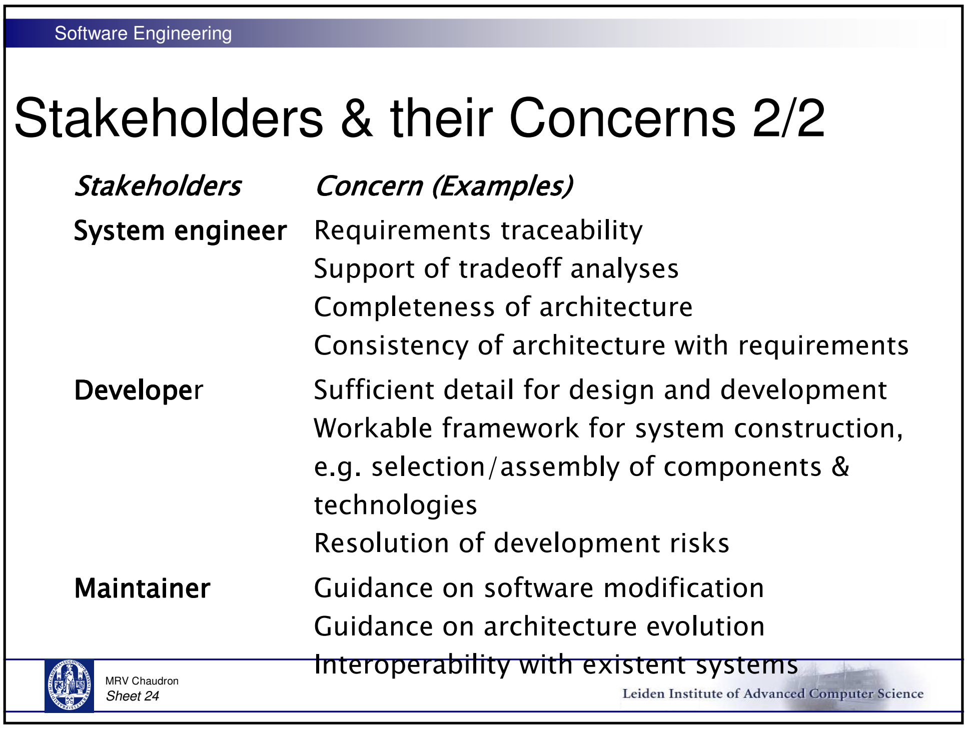
- Marketing, Sales
- Management ...

Different types of architectures?



Stakeholders & their Concerns 1/2

Stakeholder	*Concern (Examples)*
Customer	Business goals
	Schedule & budget estimation
	Feasibility and risk assessment
	Requirements traceability & progress tracking
	Product-line compatibility
User	Consistency with requirements & use cases
	Future requirements growth accommodation
	Support of dependability & other X-abilities
Service manager	Reliability, availability and maintainability

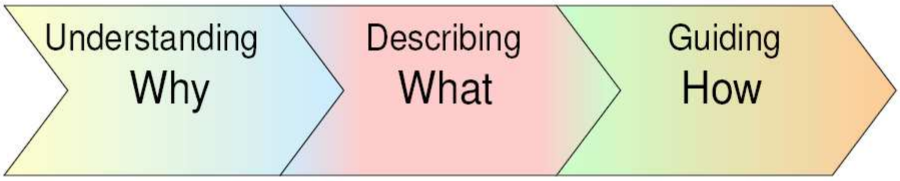


Stakeholders & their Concerns 2/2

Stakeholders	*Concern (Examples)*
System engineer	Requirements traceability Support of tradeoff analyses Completeness of architecture Consistency of architecture with requirements
Developer	Sufficient detail for design and development Workable framework for system construction, e.g. selection/assembly of components & technologies Resolution of development risks
Maintainer	Guidance on software modification Guidance on architecture evolution Interoperability with existent systems

Multiple Purposes of Architecture

Understanding
+ Analyzing + Communicating + Constructing



Picture from Gerrit Muller, How to Create a Managable Platform Architecture

When Architecting?

- **When developing a new system**
- **When changing a system**
  - if an architecture description is not available, or insufficient, as a basis for change
  - adapt the architecture documentation to changes
- **When integrating existing systems**
- **For special communication needs**
  to provide a common ground for understanding

… giving people the appropriate tools to frame and structure their discussion and decision making is an enormous benefit to the disciplined development of complex systems.

Software Architecture in Practice 2nd ed.
Bass, Clements, Kazman

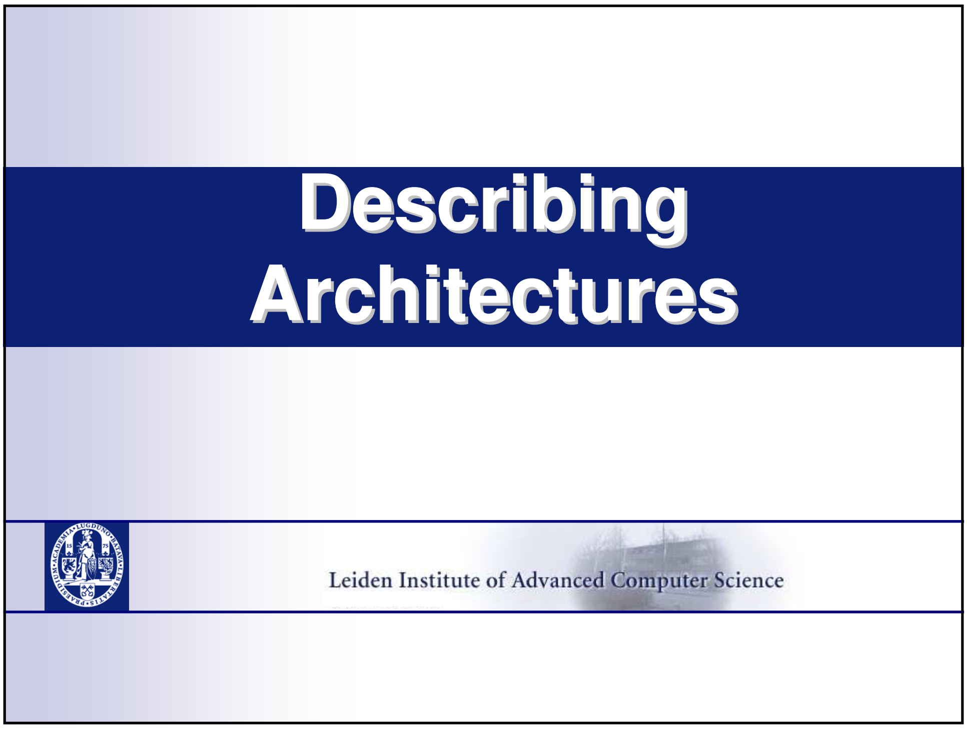
What is Software Architecture?

Definition 3

Architecture of software is a collection of design decisions that are expensive to change.

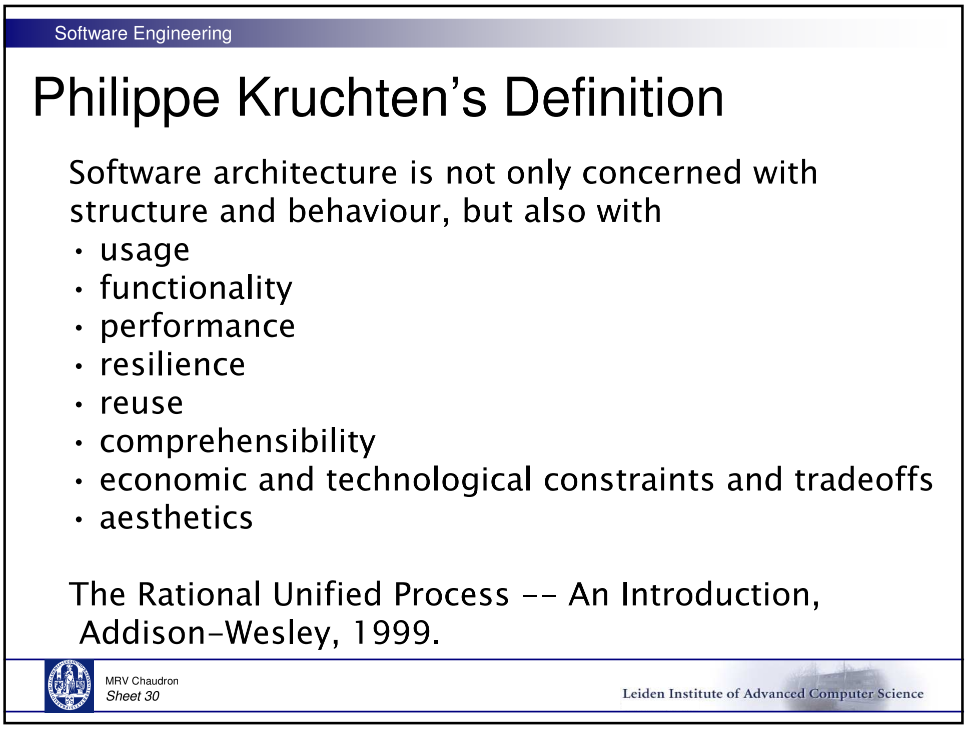
Alexander Ran, Nokia Research
September 2001 European Conference on Software Engineering

“The things that are fixed”



Describing Architectures

Leiden Institute of Advanced Computer Science



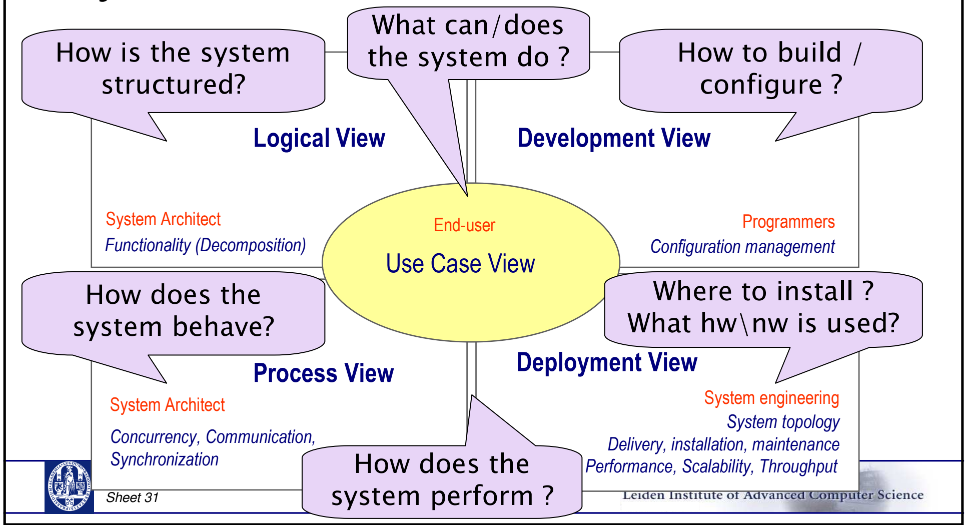
Philippe Kruchten's Definition

Software architecture is not only concerned with structure and behaviour, but also with

- usage
- functionality
- performance
- resilience
- reuse
- comprehensibility
- economic and technological constraints and tradeoffs
- aesthetics

The Rational Unified Process -- An Introduction, Addison-Wesley, 1999.

4+1 Views Representation of System Architecture



How is the system structured?

Logical View

System Architect

Functionality (Decomposition)

What can/does the system do ?

End-user

Use Case View

How to build / configure ?

Development View

Programmers

Configuration management

How does the system behave?

Process View

System Architect

Concurrency, Communication, Synchronization

How does the system perform ?

Where to install ? What hw\nw is used?

Deployment View

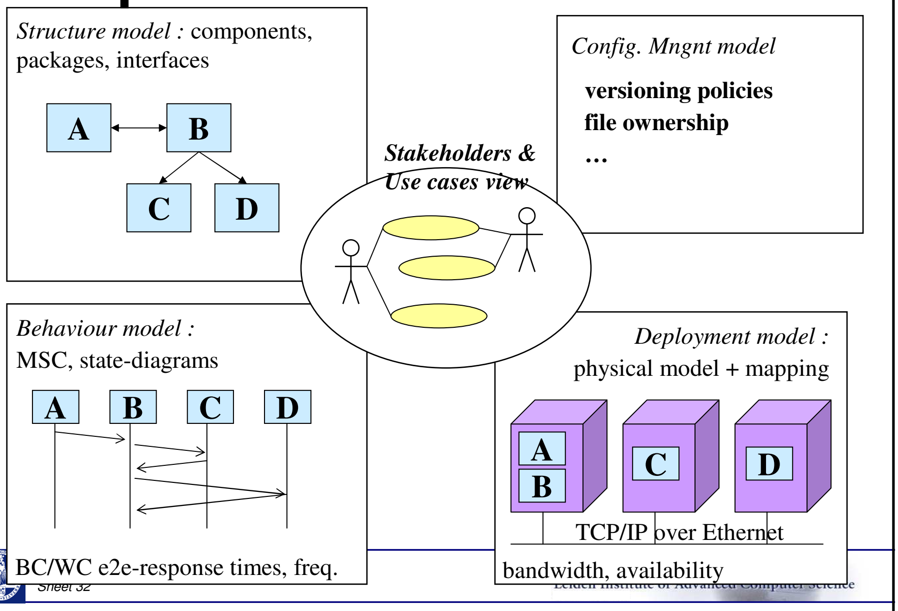
System engineering

System topology

Delivery, installation, maintenance

Performance, Scalability, Throughput

Example 4+1 model



Structure model : components, packages, interfaces

Config. Mngnt model

versioning policies

file ownership

…

Stakeholders & Use cases view

Behaviour model : MSC, state-diagrams

BC/WC e2e-response times, freq.

Deployment model : physical model + mapping

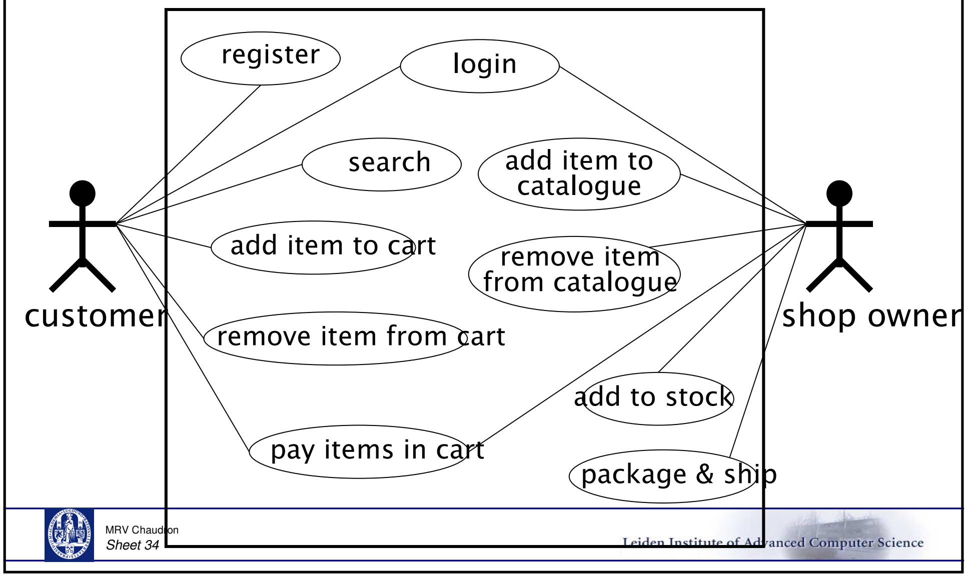
TCP/IP over Ethernet

bandwidth, availability

Use Case Diagram

- Captures system functionality as seen by users
- Built in early stages of development
- Purpose
  - □ Specify the context of a system
  - □ Capture the requirements of a system
  - □ Validate an architecture's completeness
  - □ Drive implementation and generate test cases
- Developed by analysts and domain experts

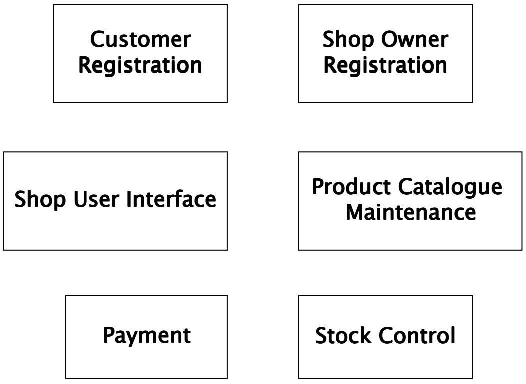
Case: Web shop

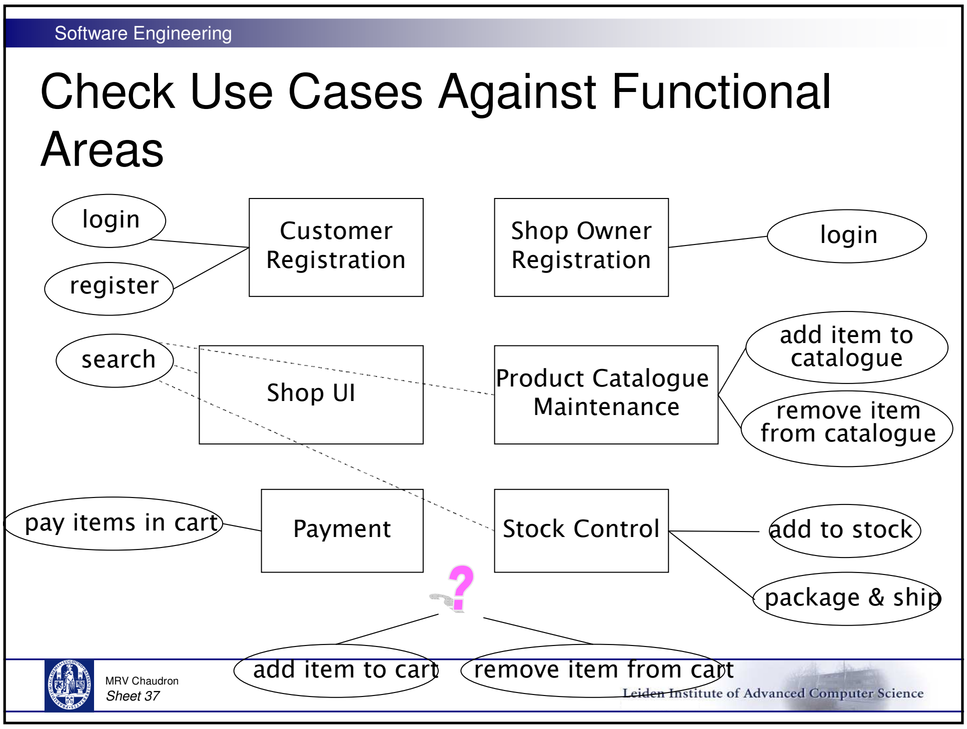


Structure Diagram

- Defines subsystems of functionality
- Purpose
  - □ Define decomposition into subsystems
  - □ Provide support for use-cases
- Use Component diagram
  - □ May use Class, but this suggests OO implem.

Web Shop: Functional Areas (V0.1)





Software Engineering

Check Use Cases Against Functional Areas

login

register

Customer Registration

Shop Owner Registration

login

search

Shop UI

Product Catalogue Maintenance

add item to catalogue

remove item from catalogue

pay items in cart

Payment

Stock Control

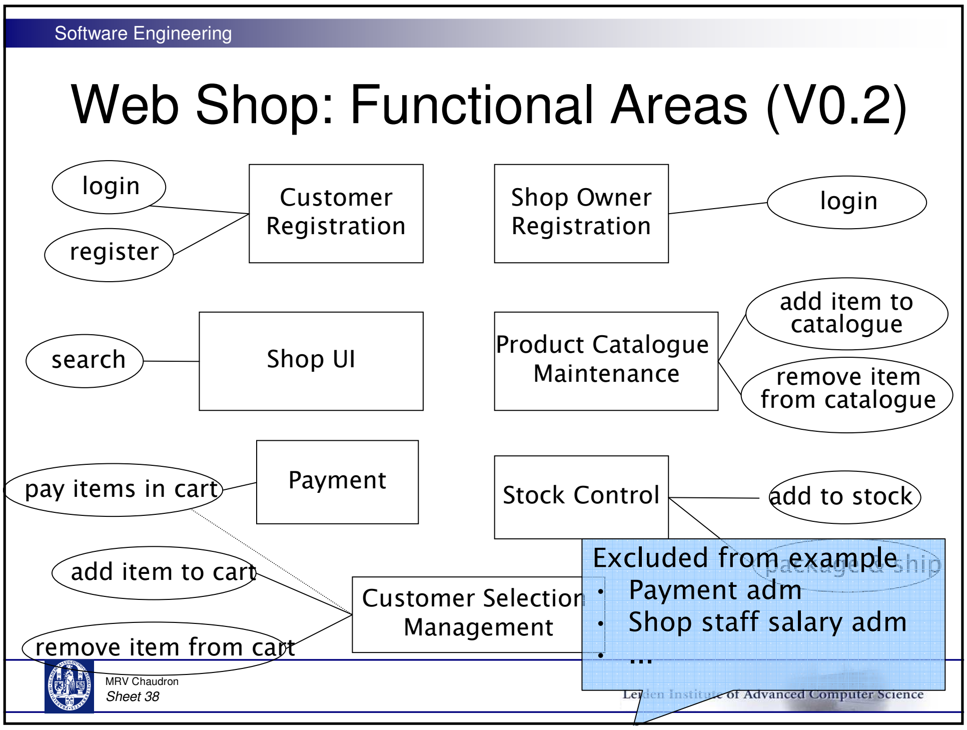
add to stock

package & ship

?

add item to cart

remove item from cart



Software Engineering

Web Shop: Functional Areas (V0.2)

login

register

Customer Registration

Shop Owner Registration

login

search

Shop UI

Product Catalogue Maintenance

add item to catalogue

remove item from catalogue

pay items in cart

Payment

Stock Control

add to stock

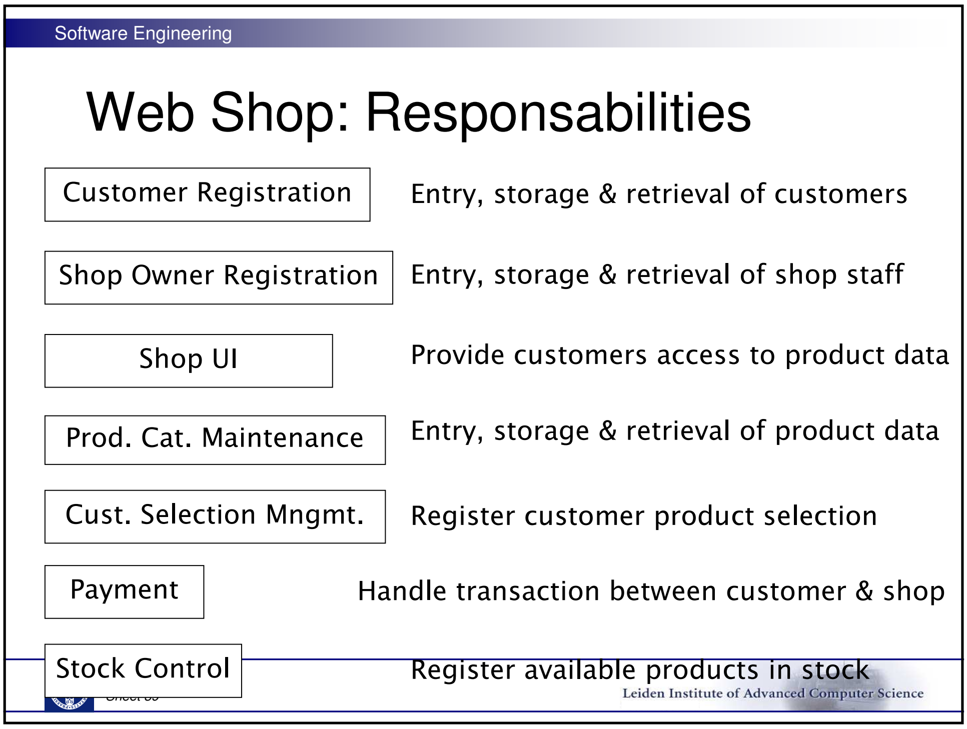
package & ship

add item to cart

remove item from cart

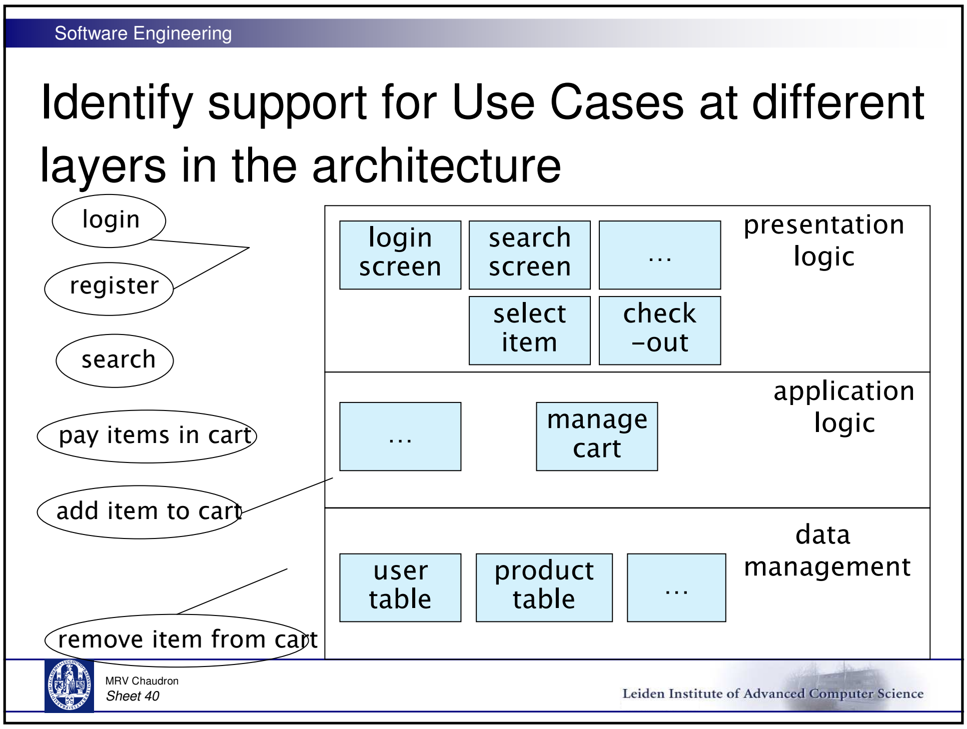
Customer Selection Management

Excluded from example
- Payment adm
- Shop staff salary adm
- ...



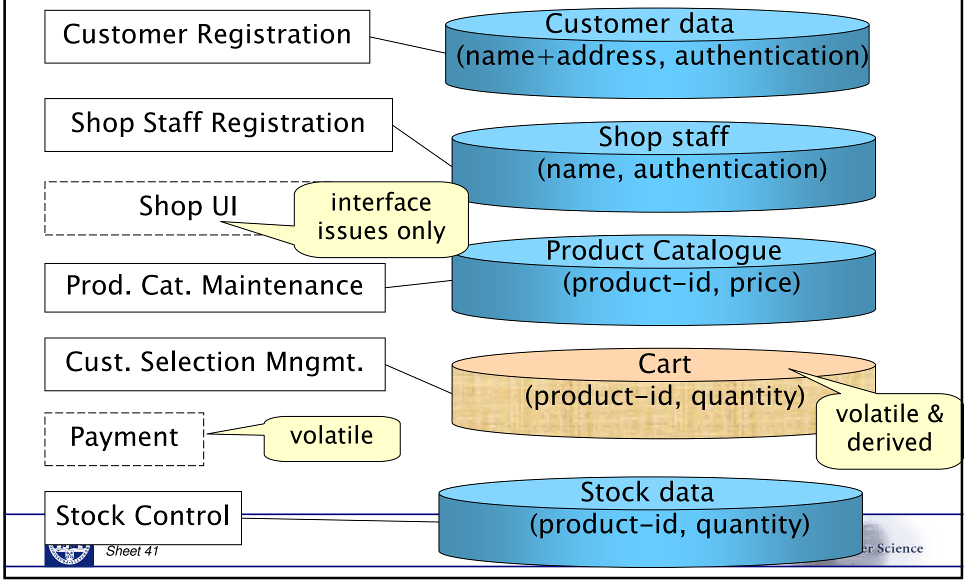
Web Shop: Responsabilities

Customer Registration	Entry, storage & retrieval of customers
Shop Owner Registration	Entry, storage & retrieval of shop staff
Shop UI	Provide customers access to product data
Prod. Cat. Maintenance	Entry, storage & retrieval of product data
Cust. Selection Mngmt.	Register customer product selection
Payment	Handle transaction between customer & shop
Stock Control	Register available products in stock



Identify support for Use Cases at different layers in the architecture

Identification of Data Domains



Customer Registration

Customer data (name+address, authentication)

Shop Staff Registration

Shop staff (name, authentication)

Shop UI

interface issues only

Prod. Cat. Maintenance

Product Catalogue (product–id, price)

Cust. Selection Mngmt.

Cart (product–id, quantity)

volatile & derived

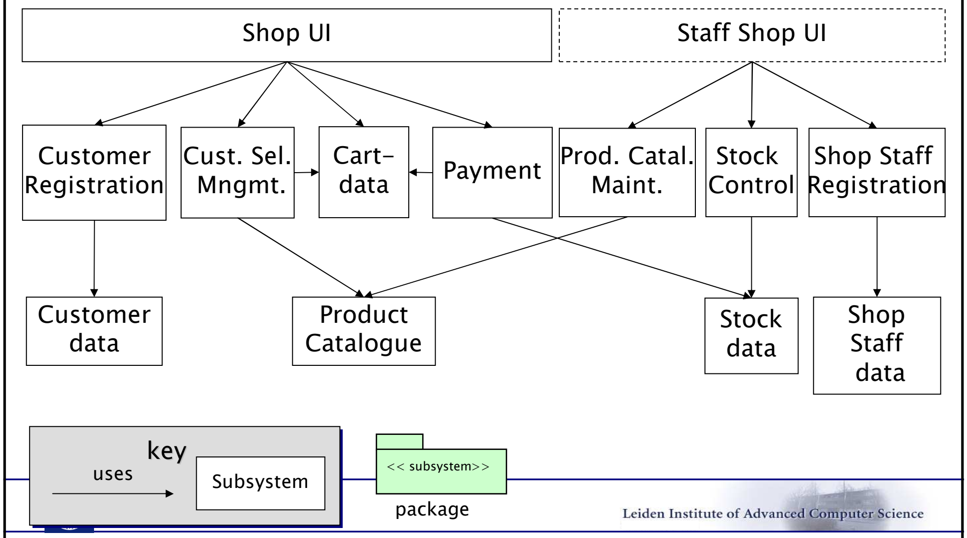
Payment

volatile

Stock Control

Stock data (product–id, quantity)

Identification of Dependencies



Shop UI

Staff Shop UI

Customer Registration

Cust. Sel. Mngmt.

Cart–data

Payment

Prod. Catal. Maint.

Stock Control

Shop Staff Registration

Customer data

Product Catalogue

Stock data

Shop Staff data

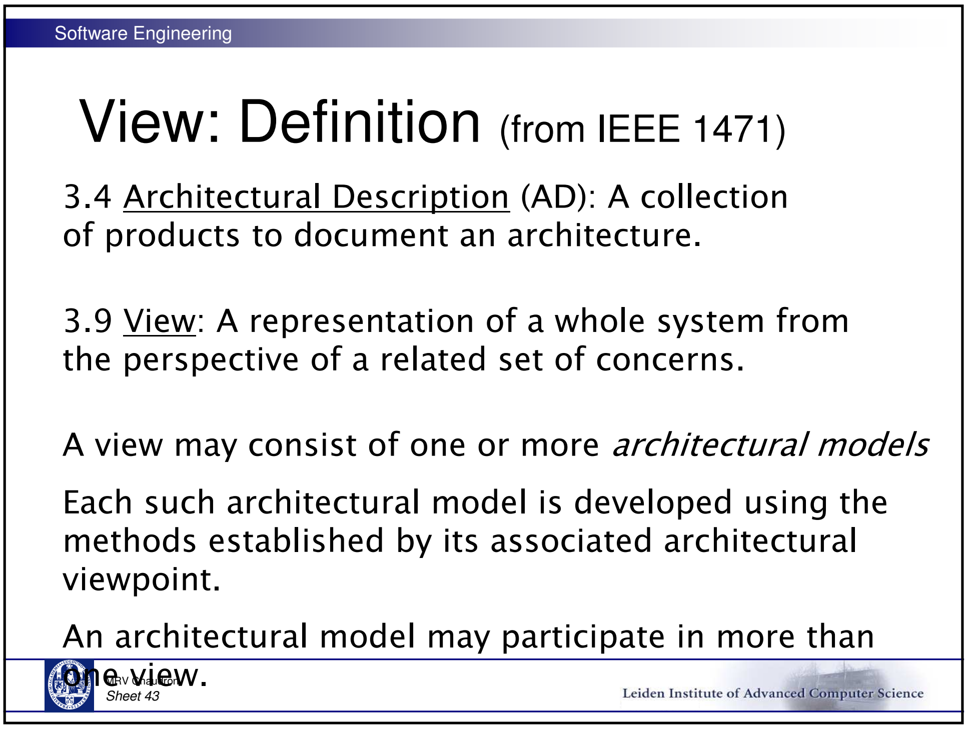
key

uses

Subsystem

<< subsystem>>

package



View: Definition (from IEEE 1471)

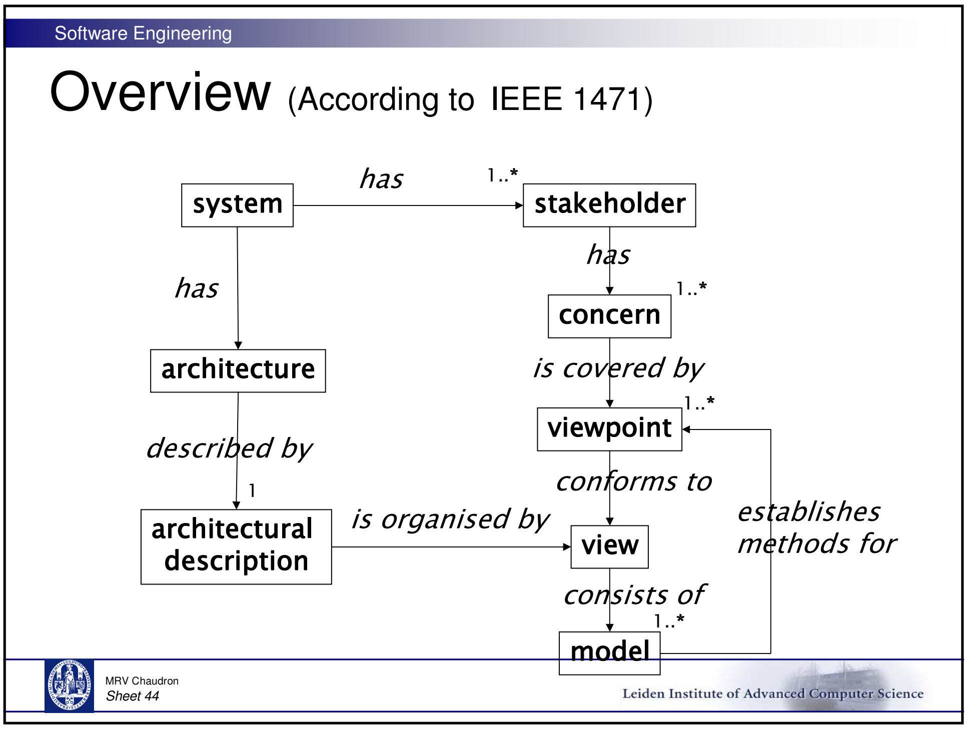
3.4 Architectural Description (AD): A collection of products to document an architecture.

3.9 View: A representation of a whole system from the perspective of a related set of concerns.

A view may consist of one or more *architectural models*

Each such architectural model is developed using the methods established by its associated architectural viewpoint.

An architectural model may participate in more than one view.

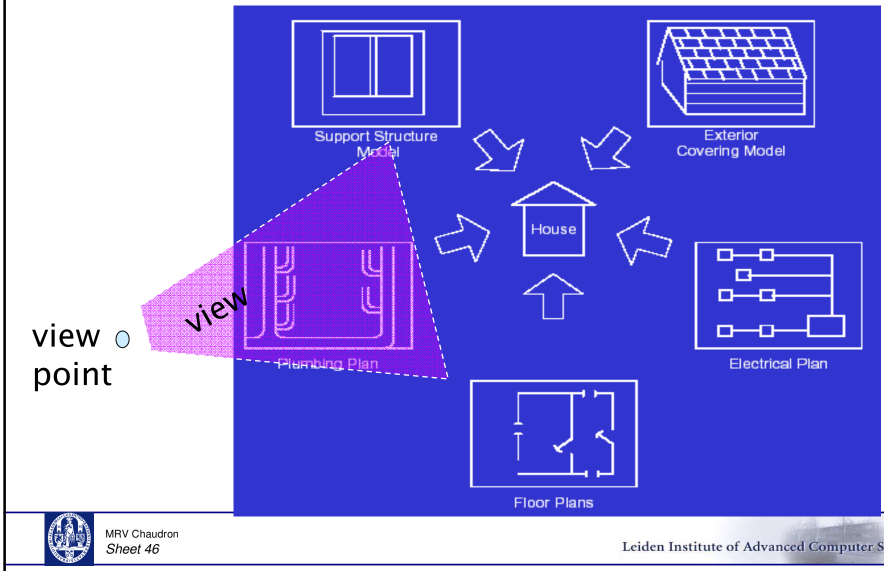


Overview (According to IEEE 1471)

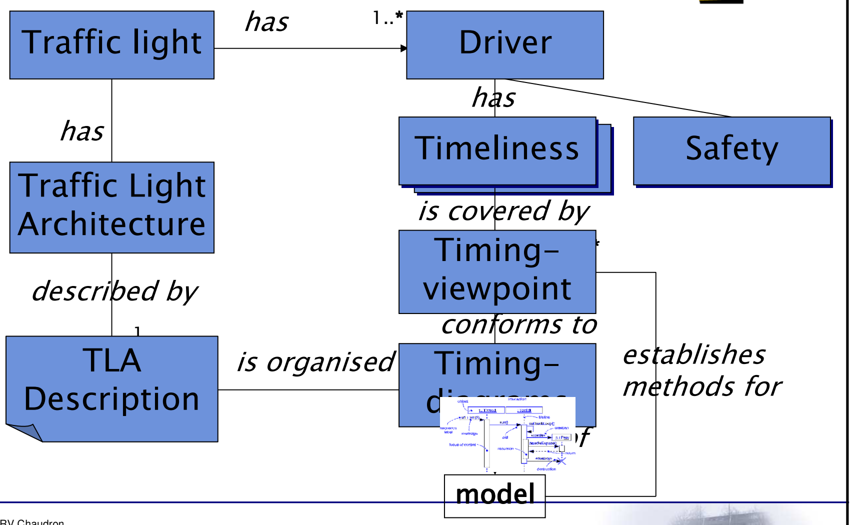
Architectural view

- An architectural view is a simplified description (an abstraction) of a system from a particular perspective/view point, covering particular concerns, and omitting entities that are not relevant to this perspective

Viewpoints & views



Overview - example



Recommendations for Architecture Description

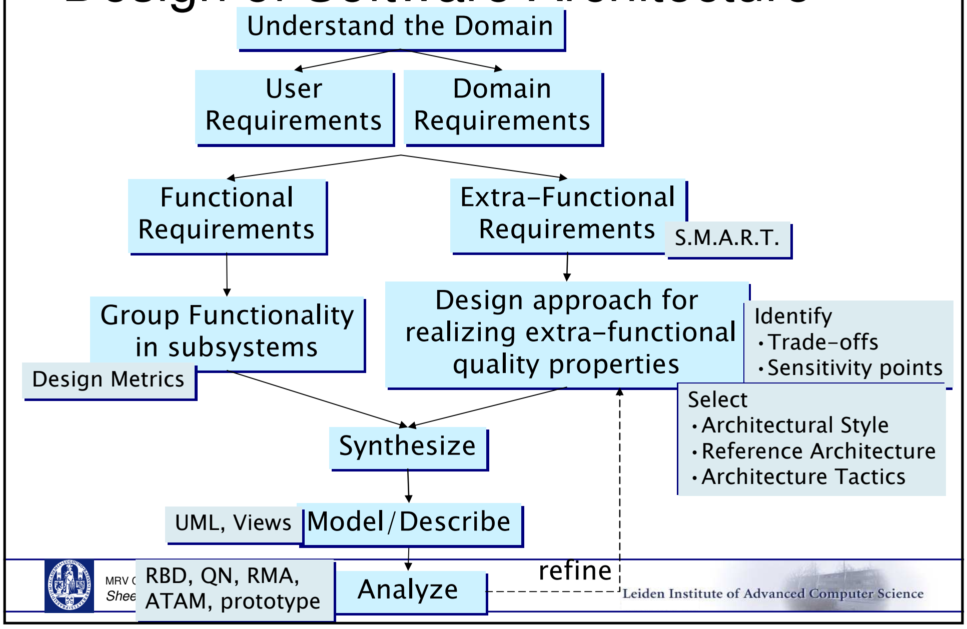
- describe the system **goals** & the **assumptions on the environment**
- describe the design **principles, decisions, guidelines**
  - and their **rationale**
- describe **several views** that can be combined in a consistent model at least the following views should be given:
  - **functional/structural (decomposition) view**
    - include a description of the interfaces between (sub)systems
  - **process/dynamical/behaviour view**
  - **deployment view**
- prevent mixing of views
- address **non-functional** (*ilities) aspects
- use a well-defined notation and include its **key**/**legend**
  - this aids systematic use of notation/avoids inconsistent use
  - improves common understanding
  - prevents mixing of different levels of abstraction
- add explanation in **natural language**

Concluding Remarks

Experience is the hardest kind of teacher.
It gives the test first and the lesson afterward.
Susan Ruth, 1993

- Software Architecture is a critical aspect in the design and development of software
- We discussed definitions and objectives of Sw.Arch.
- Good architectural design requires human creativity, hard work, and a critical attitude.
- Understanding of basic principles of architecture design, analysis, documentation, and process are necessary, but experience is hard to beat.

Design of Software Architecture



WWW References Software Architecture

- **Software architecture resources (Gert Florijn, Serc)**
  http://www.serc.nl/people/florijn/interests/arch.html
- **Software Architecture at the SEI**
  http://www.sei.cmu.edu/ata/ata_init.html
  inspired by practice; focus on architecture evaluation; lots of papers
- **Software Architecting Process / Success Factors & Pitfalls**
  http://www.bredemeyer.com/howto.htm
- **Architectural Blueprints: the 4+1 view model of Software Architecture** http://www.rational.com/media/whitepapers/Pbk4p1.pdf.
  The original paper by Kruchten: nice examples, but old (pre-UML) notation

Questions

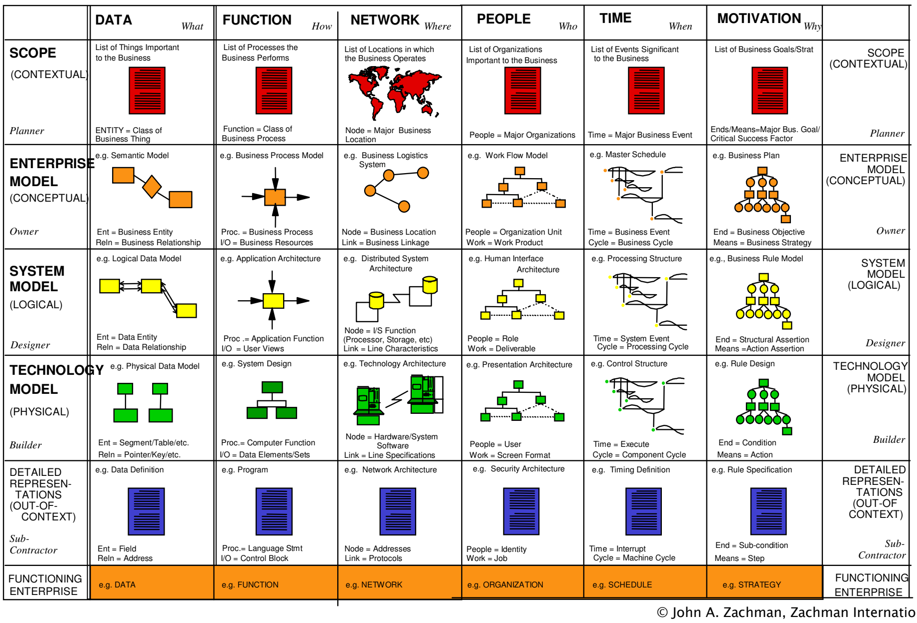
Recommendations for Architecture Description

- describe the system **goals** & the **assumptions on the environment**
- describe the design **principles, decisions, guidelines**
  - and their **rationale**
- describe **several views** that can be combined in a consistent model

  at least the following views should be given:
  - **functional/structural (decomposition) view**
    - include a description of the interfaces between (sub)systems
  - **process/dynamical view**
  - **deployment view**
- prevent mixing of views
- address **non-functional** (*ilities) aspects
- use a well-defined notation and include its **key/legend**
    - this aids systematic use of notation/avoids inconsistent use
    - improves common understanding
    - prevents mixing of different levels of abstraction
- add explanation in **natural language**

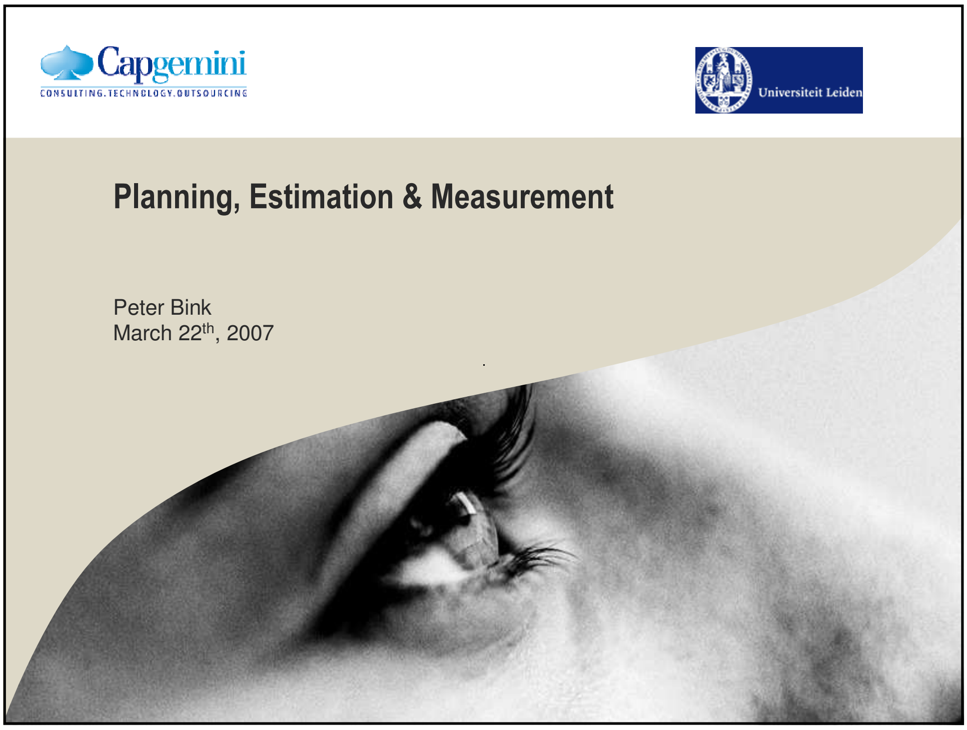
Zachman Enterprise Architecture Framework

1987, extended: 1992



	DATA *What*	FUNCTION *How*	NETWORK *Where*	PEOPLE *Who*	TIME *When*	MOTIVATION *Why*	
SCOPE (CONTEXTUAL) *Planner*	List of Things Important to the Business ENTITY = Class of Business Thing	List of Processes the Business Performs Function = Class of Business Process	List of Locations in which the Business Operates Node = Major Business Location	List of Organizations Important to the Business People = Major Organizations	List of Events Significant to the Business Time = Major Business Event	List of Business Goals/Strat Ends/Means=Major Bus. Goal/ Critical Success Factor	SCOPE (CONTEXTUAL) *Planner*
ENTERPRISE MODEL (CONCEPTUAL) *Owner*	e.g. Semantic Model Ent = Business Entity Reln = Business Relationship	e.g. Business Process Model Proc. = Business Process I/O = Business Resources	e.g. Business Logistics System Node = Business Location Link = Business Linkage	e.g. Work Flow Model People = Organization Unit Work = Work Product	e.g. Master Schedule Time = Business Event Cycle = Business Cycle	e.g. Business Plan End = Business Objective Means = Business Strategy	ENTERPRISE MODEL (CONCEPTUAL) *Owner*
SYSTEM MODEL (LOGICAL) *Designer*	e.g. Logical Data Model Ent = Data Entity Reln = Data Relationship	e.g. Application Architecture Proc .= Application Function I/O = User Views	e.g. Distributed System Architecture Node = I/S Function (Processor, Storage, etc) Link = Line Characteristics	e.g. Human Interface Architecture People = Role Work = Deliverable	e.g. Processing Structure Time = System Event Cycle = Processing Cycle	e.g., Business Rule Model End = Structural Assertion Means =Action Assertion	SYSTEM MODEL (LOGICAL) *Designer*
TECHNOLOGY MODEL (PHYSICAL) *Builder*	e.g. Physical Data Model Ent = Segment/Table/etc. Reln = Pointer/Key/etc.	e.g. System Design Proc.= Computer Function I/O = Data Elements/Sets	e.g. Technology Architecture Node = Hardware/System Software Link = Line Specifications	e.g. Presentation Architecture People = User Work = Screen Format	e.g. Control Structure Time = Execute Cycle = Component Cycle	e.g. Rule Design End = Condition Means = Action	TECHNOLOGY MODEL (PHYSICAL) *Builder*
DETAILED REPRESEN- TATIONS (OUT-OF- CONTEXT) *Sub-Contractor*	e.g. Data Definition Ent = Field Reln = Address	e.g. Program Proc.= Language Stmt I/O = Control Block	e.g. Network Architecture Node = Addresses Link = Protocols	e.g. Security Architecture People = Identity Work = Job	e.g. Timing Definition Time = Interrupt Cycle = Machine Cycle	e.g. Rule Specification End = Sub-condition Means = Step	DETAILED REPRESEN- TATIONS (OUT-OF CONTEXT) *Sub-Contractor*
FUNCTIONING ENTERPRISE	e.g. DATA	e.g. FUNCTION	e.g. NETWORK	e.g. ORGANIZATION	e.g. SCHEDULE	e.g. STRATEGY	FUNCTIONING ENTERPRISE

© John A. Zachman, Zachman International

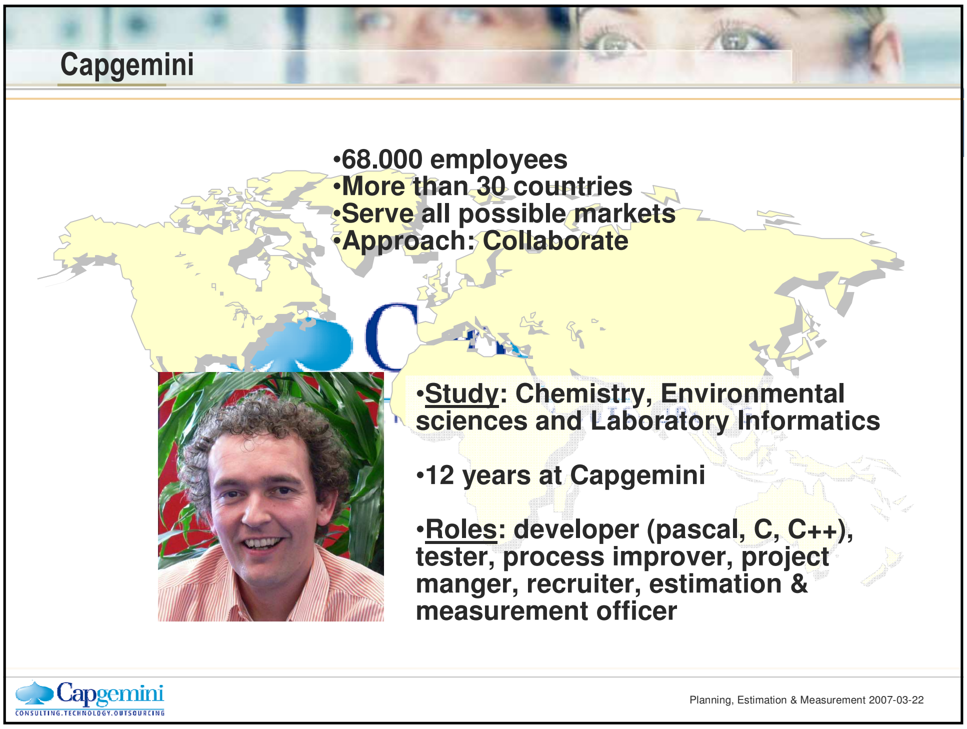


Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING

Universiteit Leiden

Planning, Estimation & Measurement

Peter Bink
March 22th, 2007



Capgemini

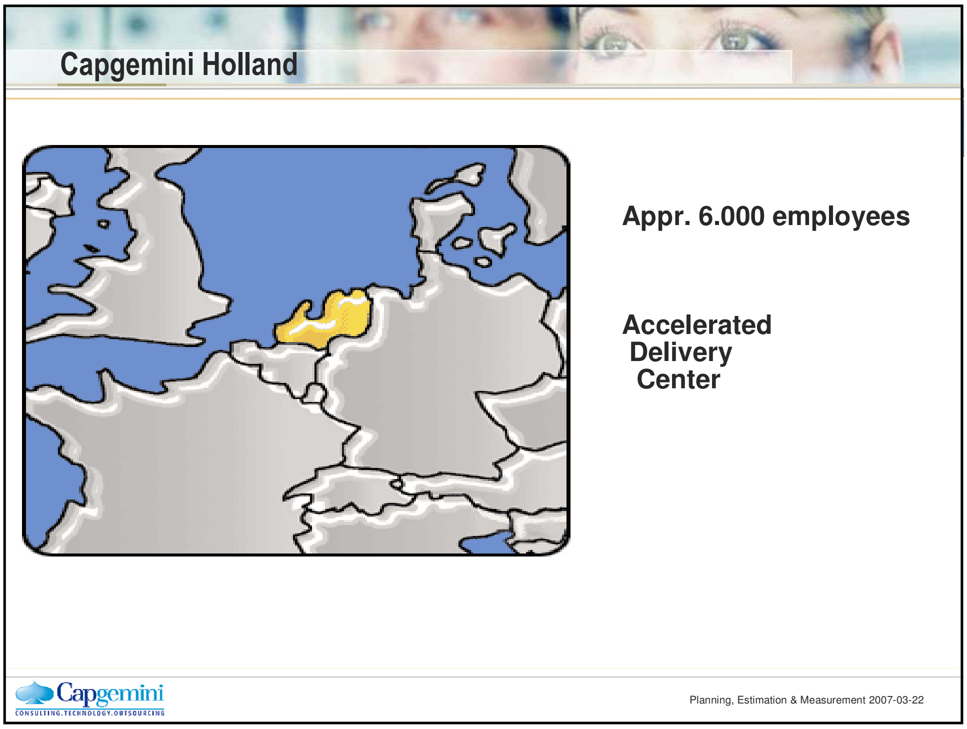
- •68.000 employees
- •More than 30 countries
- •Serve all possible markets
- •Approach: Collaborate

•Study: Chemistry, Environmental sciences and Laboratory Informatics

•12 years at Capgemini

•Roles: developer (pascal, C, C++), tester, process improver, project manger, recruiter, estimation & measurement officer

Capgemini
CONSULTING.TECHNOLOGY.OUTSOURCING



Capgemini Holland

Appr. 6.000 employees

Accelerated
Delivery
Center

ADC Objective

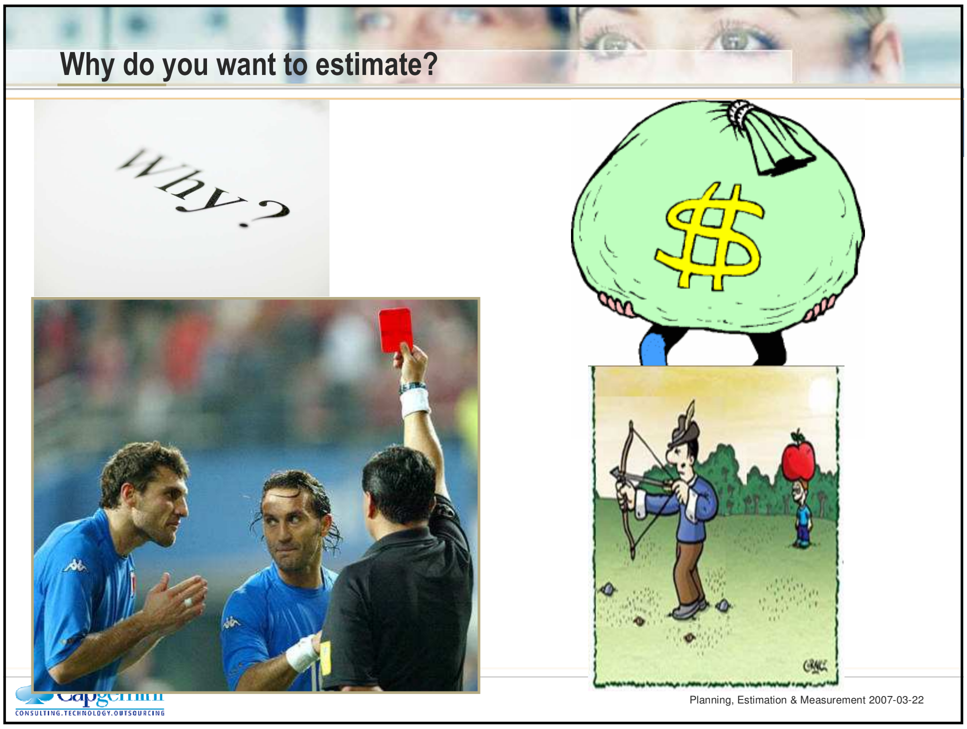
Productivity

Predictability

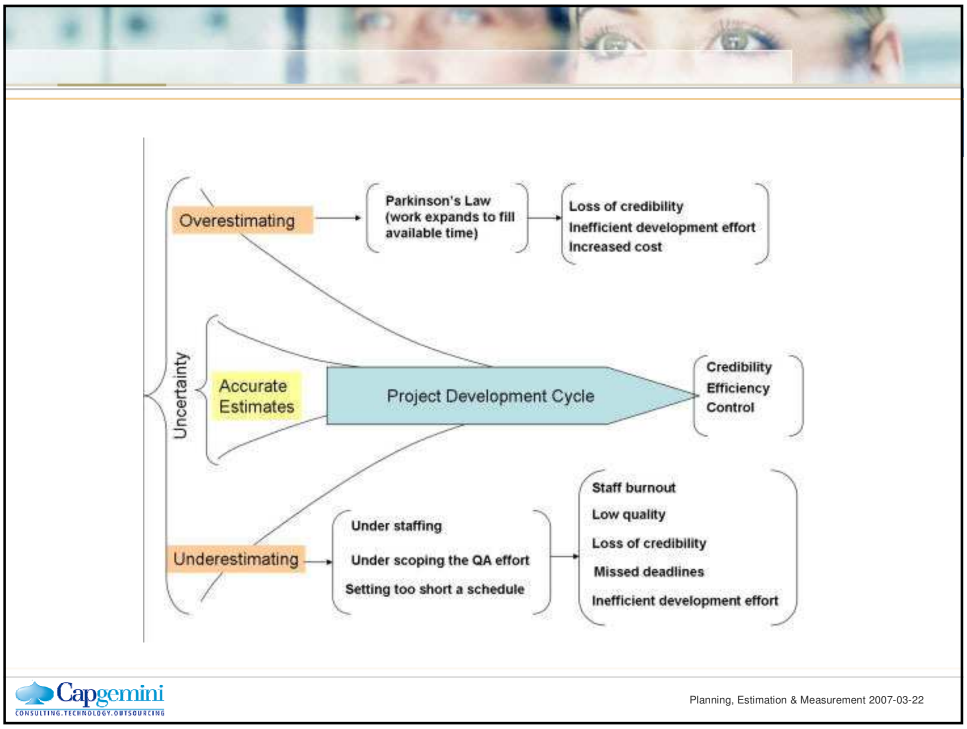
Project Risks

Development lead time

Transparency



Why do you want to estimate?

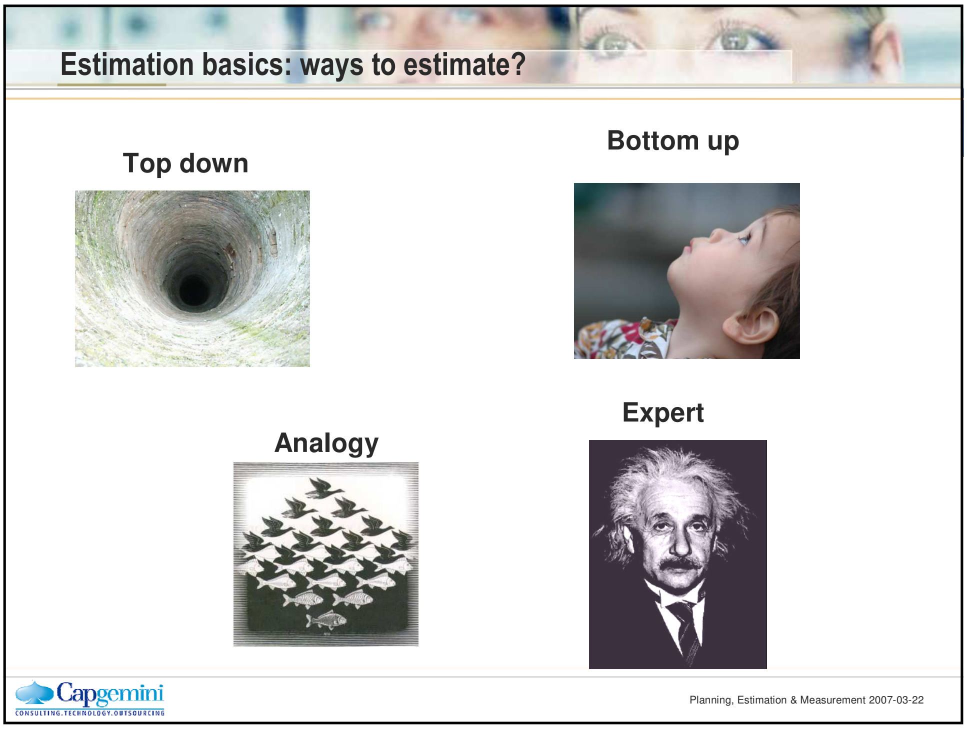


Uncertainty

Overestimating → Parkinson's Law (work expands to fill available time) → Loss of credibility, Inefficient development effort, Increased cost

Accurate Estimates — Project Development Cycle → Credibility, Efficiency, Control

Underestimating → Under staffing, Under scoping the QA effort, Setting too short a schedule → Staff burnout, Low quality, Loss of credibility, Missed deadlines, Inefficient development effort



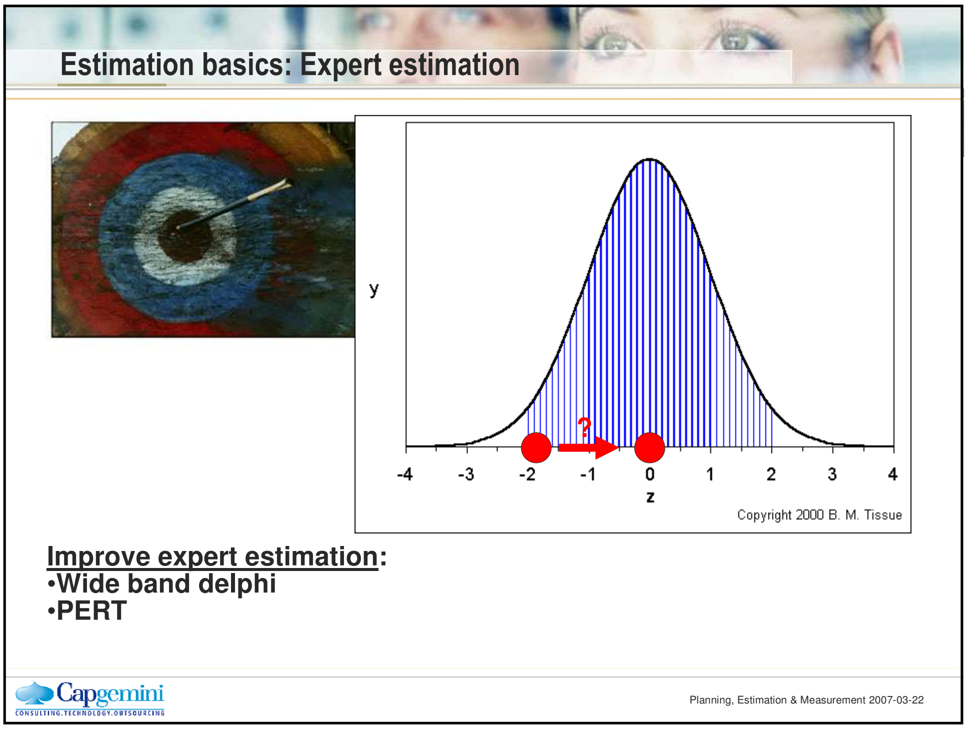
Estimation basics: ways to estimate?

Top down

Bottom up

Analogy

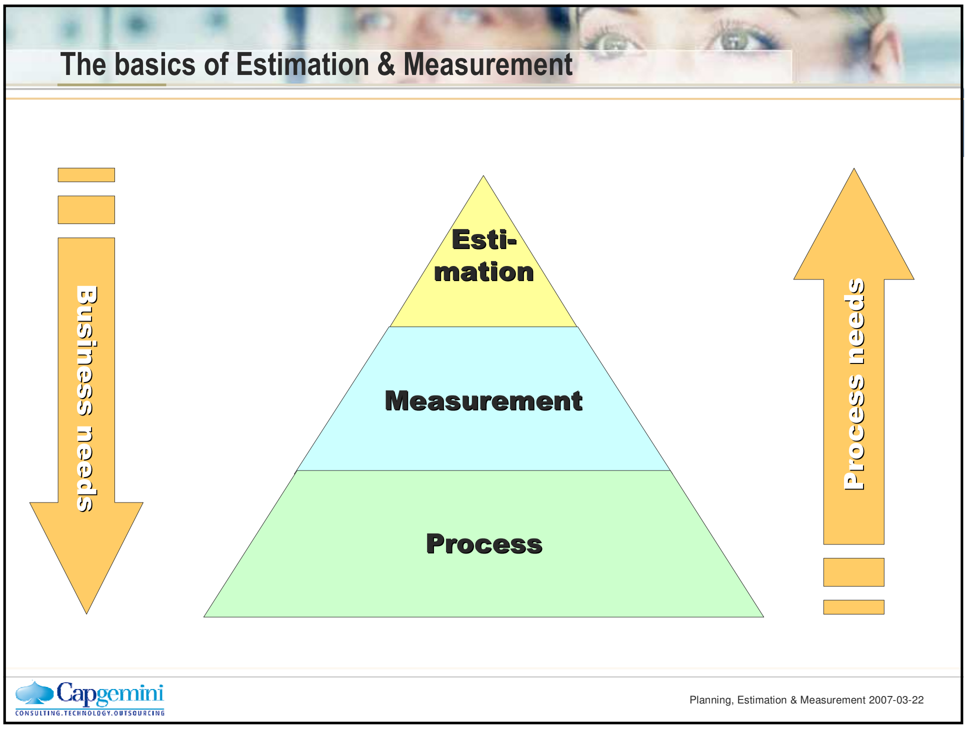
Expert



Estimation basics: Expert estimation

Copyright 2000 B. M. Tissue

Improve expert estimation:
- **Wide band delphi**
- **PERT**



The basics of Estimation & Measurement

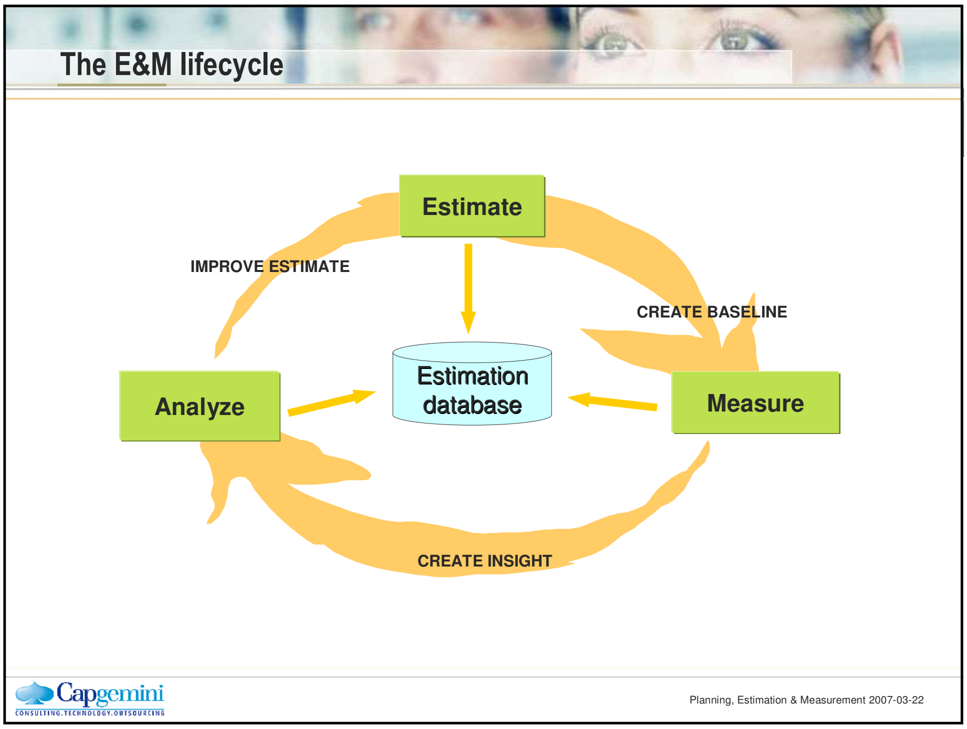
Business needs

Esti-
mation

Measurement

Process

Process needs



The E&M lifecycle

Estimate

CREATE BASELINE

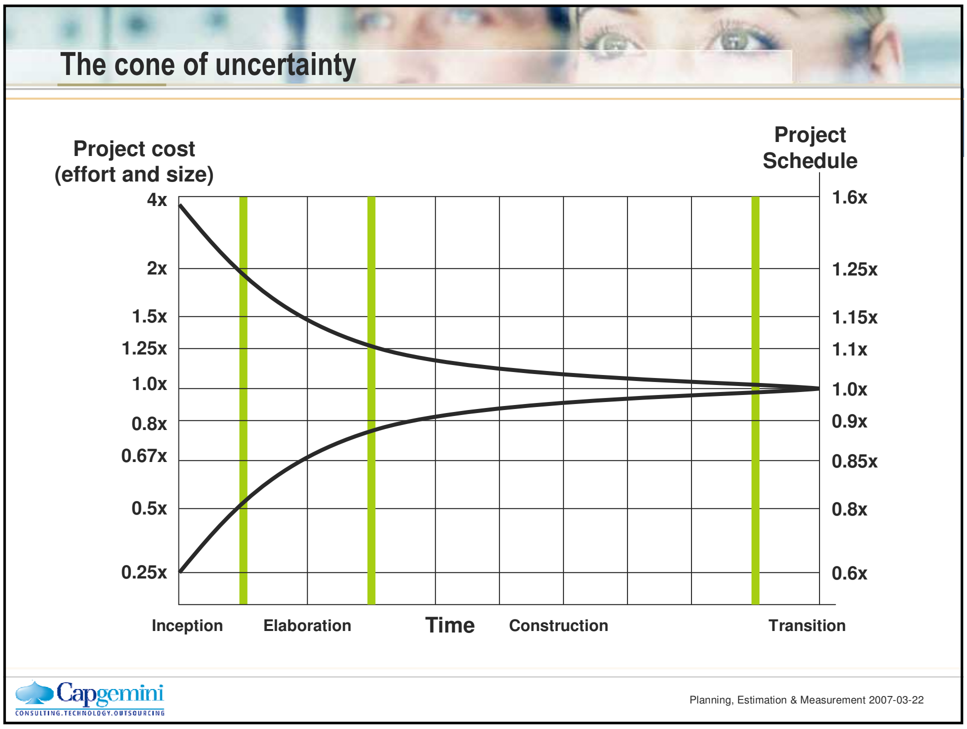
Measure

Estimation database

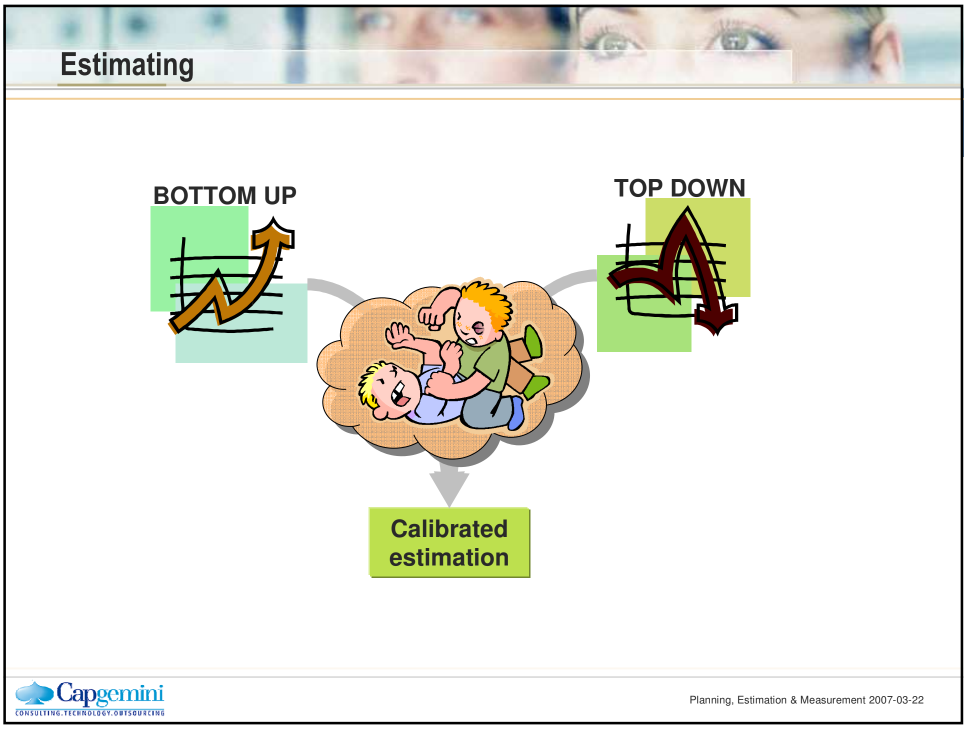
CREATE INSIGHT

Analyze

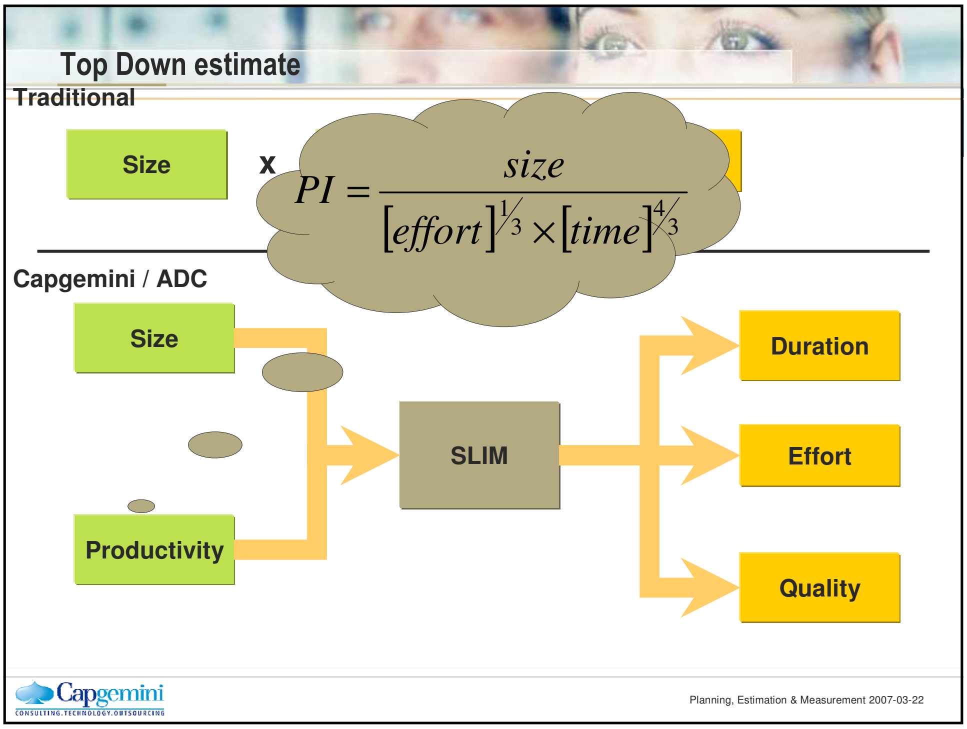
IMPROVE ESTIMATE



The cone of uncertainty



Estimating



Top Down estimate

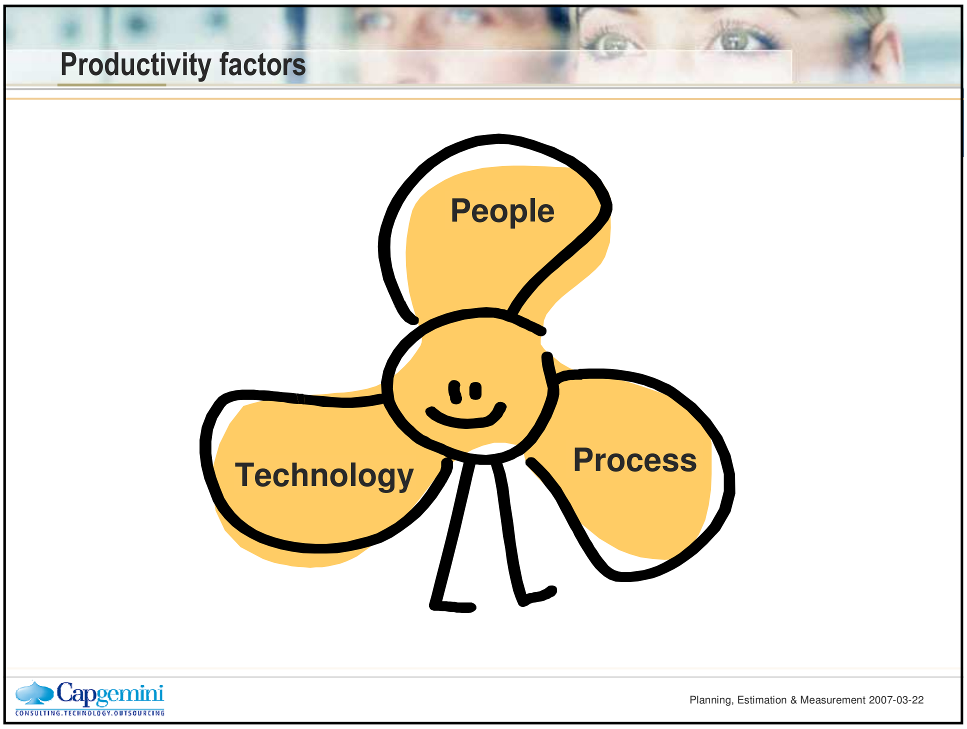
Traditional

Size x

$$PI = \frac{size}{[effort]^{1/3} \times [time]^{4/3}}$$

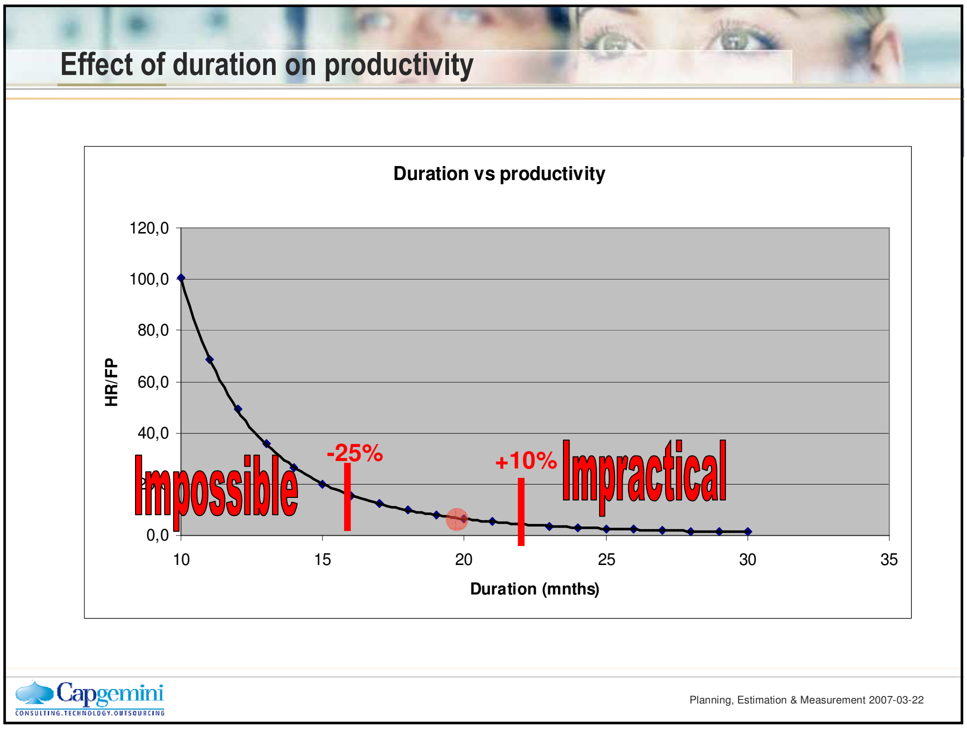
Capgemini / ADC

Planning, Estimation & Measurement 2007-03-22



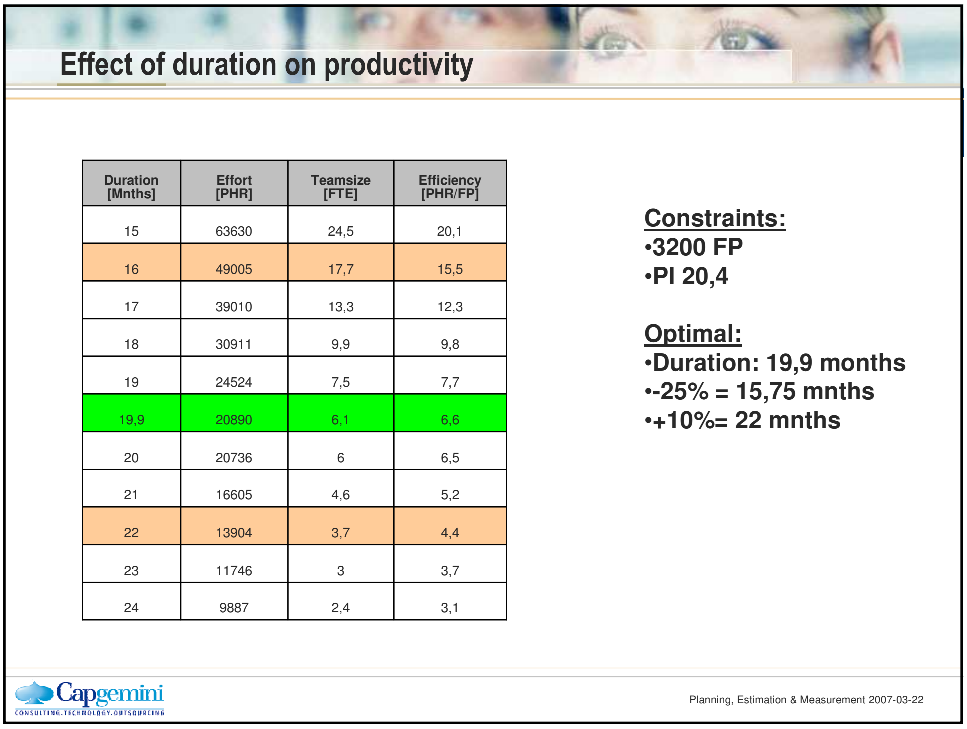
Productivity factors

Planning, Estimation & Measurement 2007-03-22



Effect of duration on productivity

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING



Effect of duration on productivity

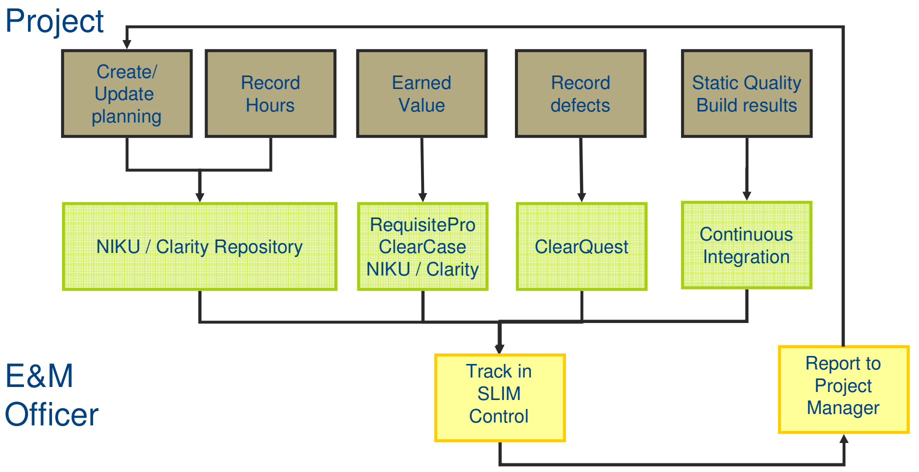
Duration [Mnths]	Effort [PHR]	Teamsize [FTE]	Efficiency [PHR/FP]
15	63630	24,5	20,1
16	49005	17,7	15,5
17	39010	13,3	12,3
18	30911	9,9	9,8
19	24524	7,5	7,7
19,9	20890	6,1	6,6
20	20736	6	6,5
21	16605	4,6	5,2
22	13904	3,7	4,4
23	11746	3	3,7
24	9887	2,4	3,1

Constraints:
•3200 FP
•PI 20,4

Optimal:
•Duration: 19,9 months
•-25% = 15,75 mnths
•+10%= 22 mnths

Capgemini CONSULTING.TECHNOLOGY.OUTSOURCING

Measurement: Tracking actuals



Project

Create/ Update planning

Record Hours

Earned Value

Record defects

Static Quality Build results

NIKU / Clarity Repository

RequisitePro ClearCase NIKU / Clarity

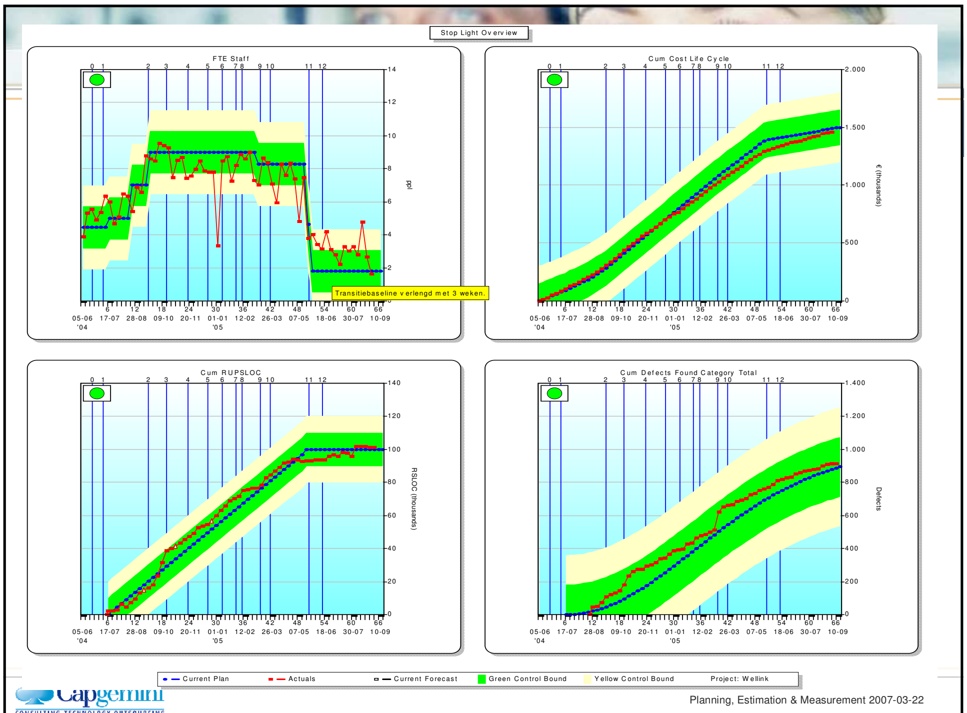
ClearQuest

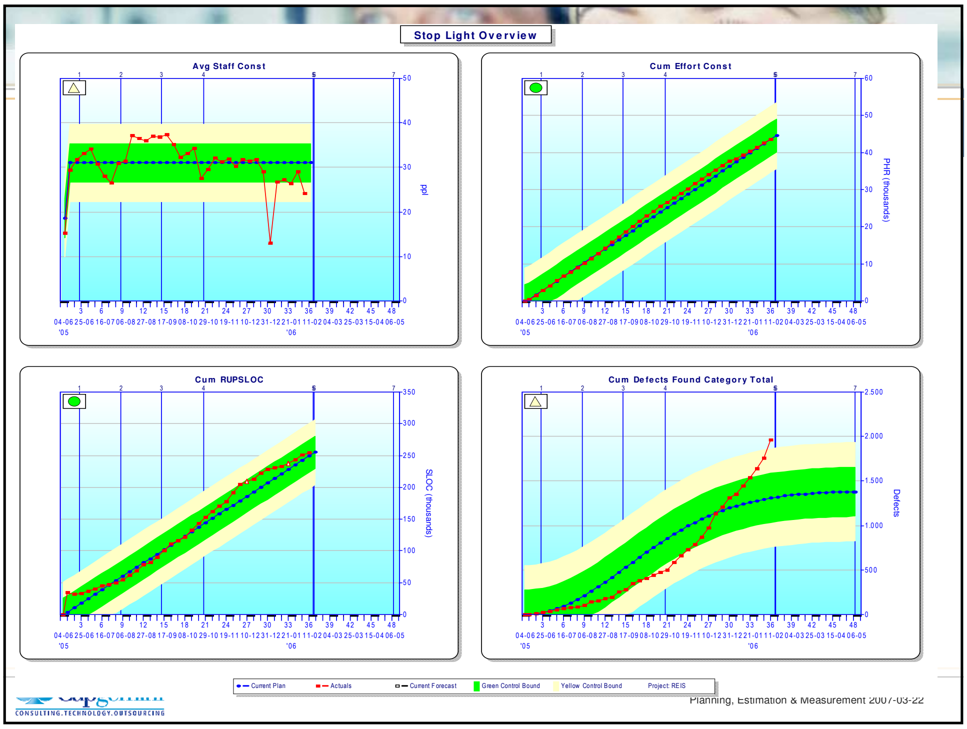
Continuous Integration

E&M Officer

Track in SLIM Control

Report to Project Manager



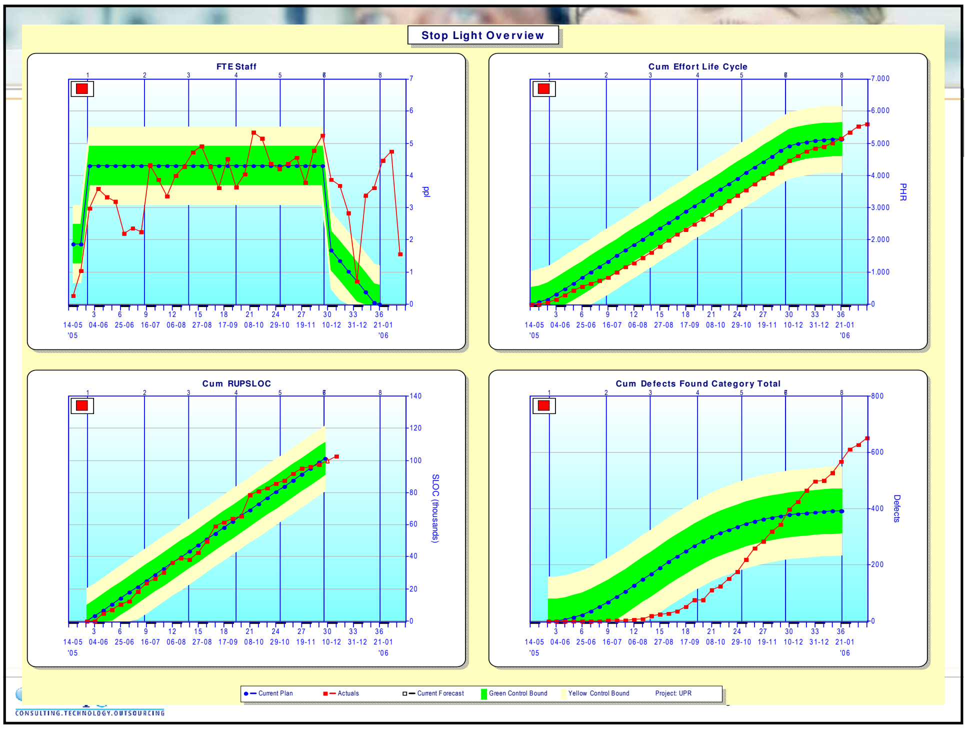


Stop Light Overview

●— Current Plan ■— Actuals □— Current Forecast 🟩 Green Control Bound Yellow Control Bound Project: REIS

CONSULTING.TECHNOLOGY.OUTSOURCING

Planning, Estimation & Measurement 2007-03-22



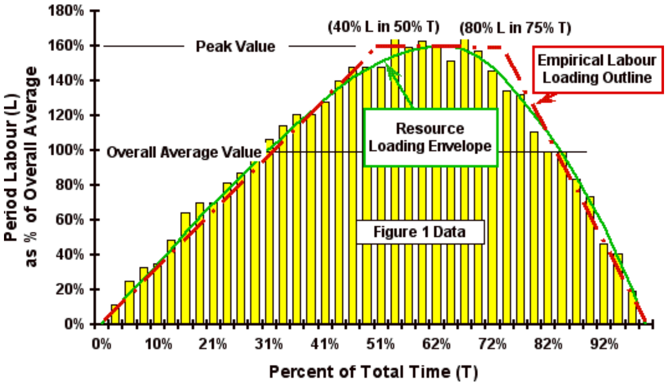
Stop Light Overview

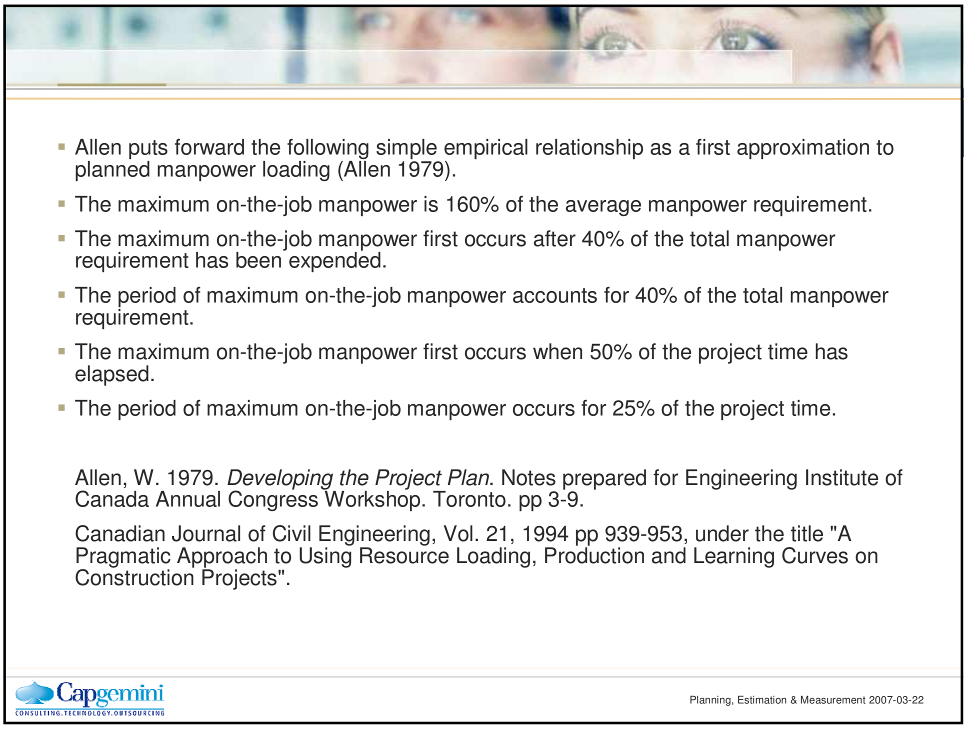
●— Current Plan ■— Actuals □— Current Forecast 🟩 Green Control Bound Yellow Control Bound Project: UPR

CONSULTING.TECHNOLOGY.OUTSOURCING

Your questions

Manpower build-up (from Construction Industry!)

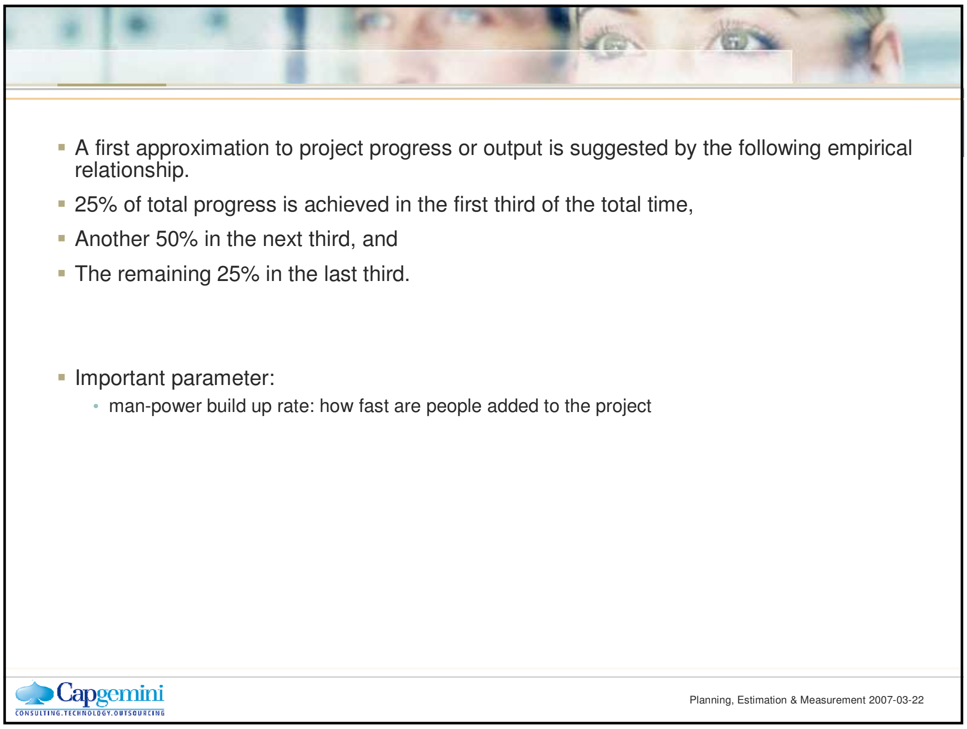




- Allen puts forward the following simple empirical relationship as a first approximation to planned manpower loading (Allen 1979).
- The maximum on-the-job manpower is 160% of the average manpower requirement.
- The maximum on-the-job manpower first occurs after 40% of the total manpower requirement has been expended.
- The period of maximum on-the-job manpower accounts for 40% of the total manpower requirement.
- The maximum on-the-job manpower first occurs when 50% of the project time has elapsed.
- The period of maximum on-the-job manpower occurs for 25% of the project time.

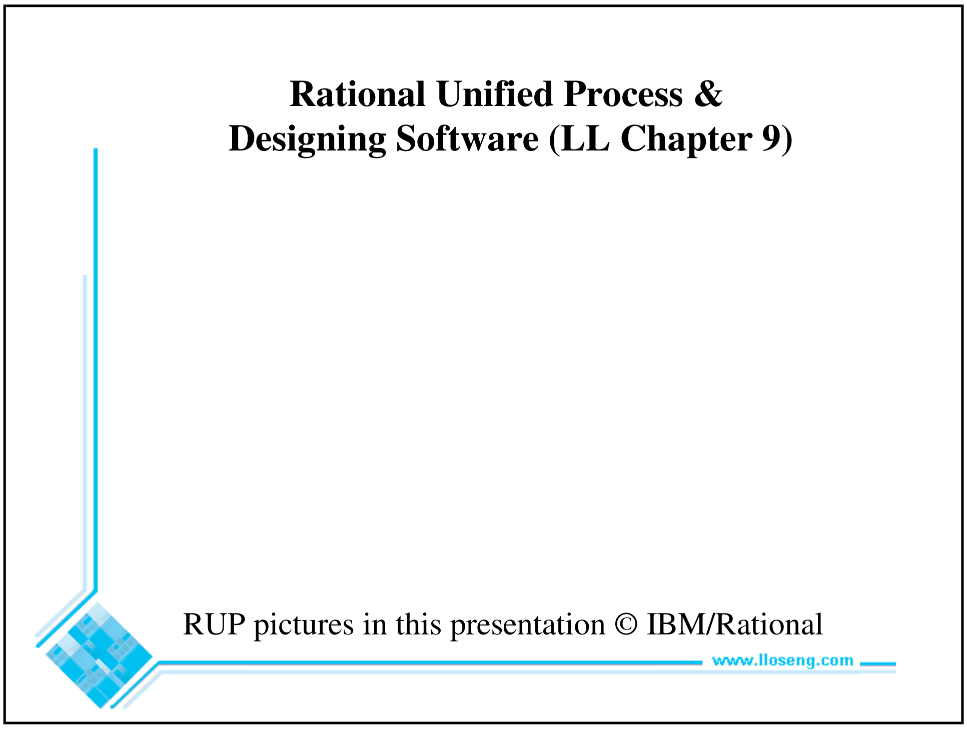
Allen, W. 1979. *Developing the Project Plan*. Notes prepared for Engineering Institute of Canada Annual Congress Workshop. Toronto. pp 3-9.

Canadian Journal of Civil Engineering, Vol. 21, 1994 pp 939-953, under the title "A Pragmatic Approach to Using Resource Loading, Production and Learning Curves on Construction Projects".



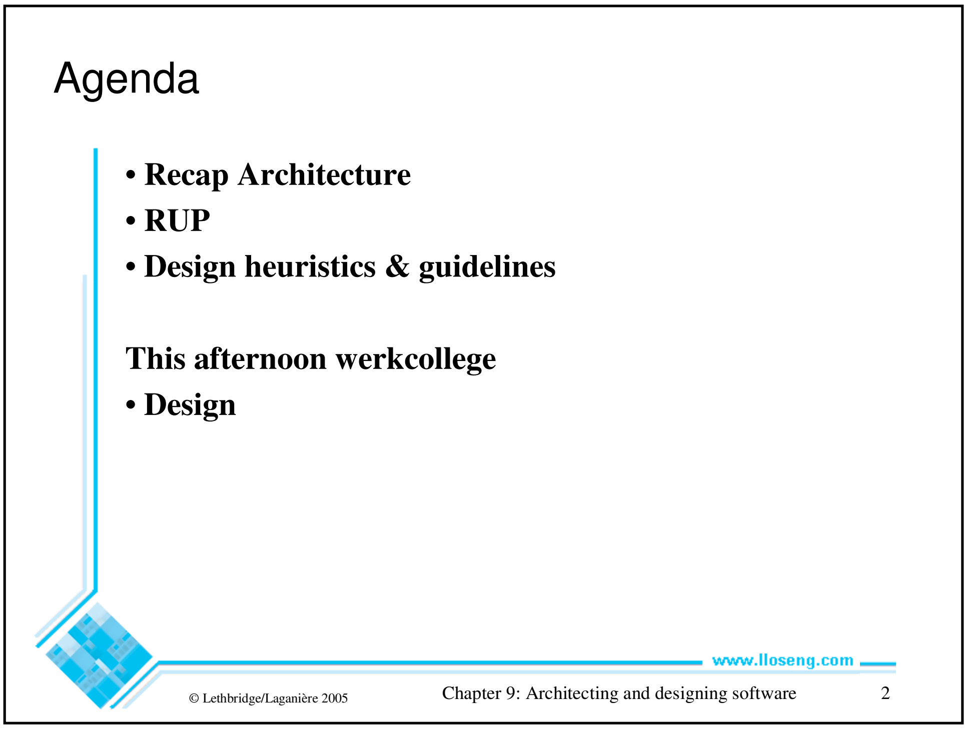
- A first approximation to project progress or output is suggested by the following empirical relationship.
- 25% of total progress is achieved in the first third of the total time,
- Another 50% in the next third, and
- The remaining 25% in the last third.

- Important parameter:
  - man-power build up rate: how fast are people added to the project



Rational Unified Process & Designing Software (LL Chapter 9)

RUP pictures in this presentation © IBM/Rational



Agenda

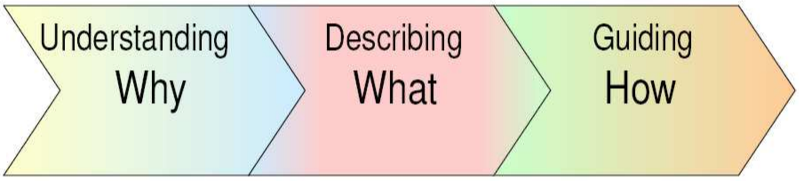
- Recap Architecture
- RUP
- Design heuristics & guidelines

This afternoon werkcollege

- Design

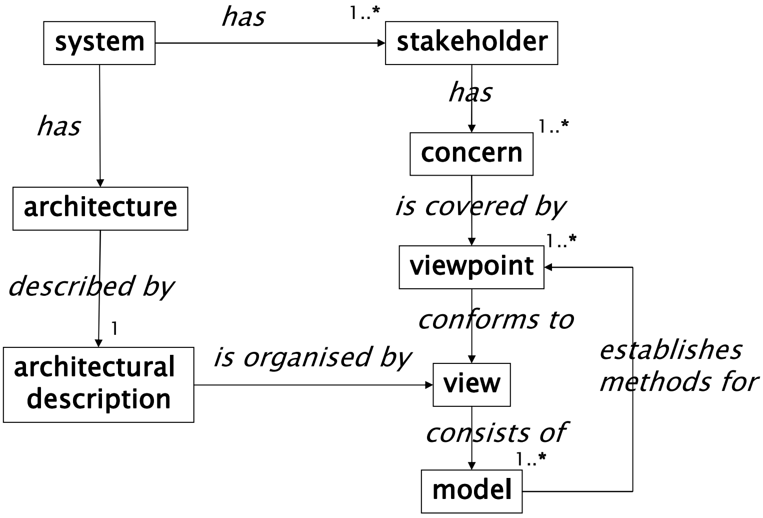
Multiple Purposes of Architecture

Understanding
+ Analyzing + Communicating + Constructing

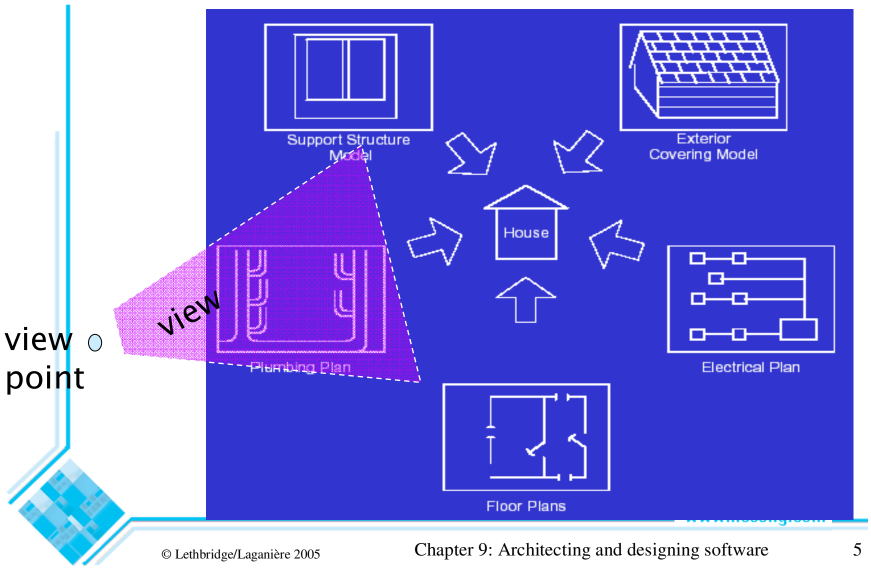


Picture from Gerrit Muller, How to Create a Managable Platform Architecture

Overview (According to IEEE 1471)



Viewpoints & views

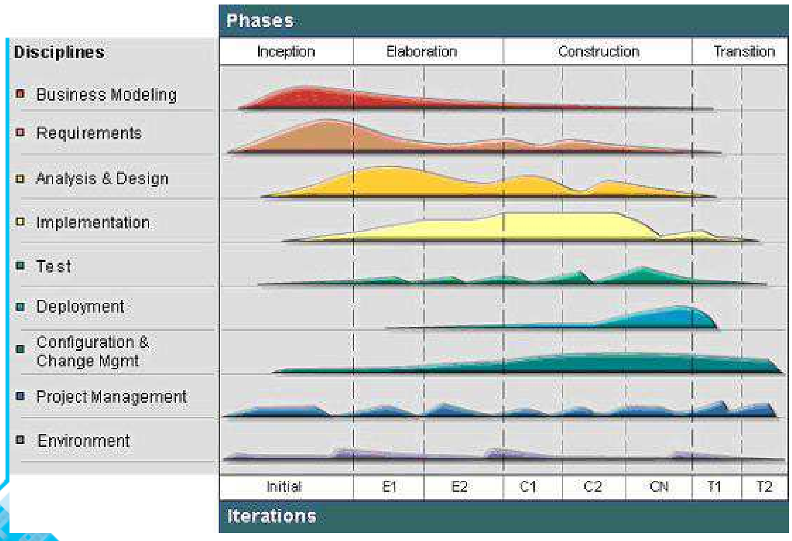


Recommendations for Architecture Description

- describe the system **goals** & the **assumptions on the environment**
- describe the design **principles, decisions, guidelines**
  - and their **rationale**
- describe **several views** that can be combined in a consistent model at least the following views should be given:
  - **functional/structural (decomposition) view**
    - include a description of the interfaces between (sub)systems
  - **process/dynamical/behaviour view**
  - **deployment view**
- prevent mixing of views
- address **non-functional** (*ilities) aspects
- use a well-defined notation and include its **key/legend**
    - this aids systematic use of notation/avoids inconsistent use
    - improves common understanding
    - prevents mixing of different levels of abstraction
- add explanation in **natural language**

Rational Unified Process (RUP)

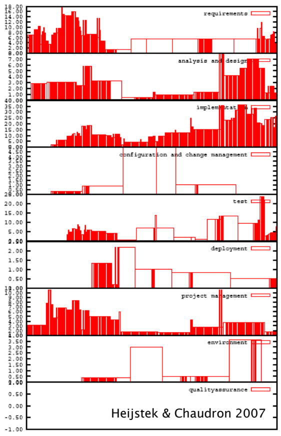
Rational Unified Process



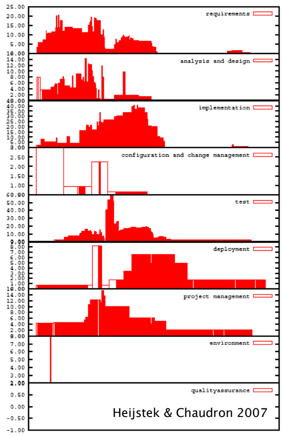
www.lloseng.com

Effort Distribution in Model-based Development – Heijstek & Chaudron

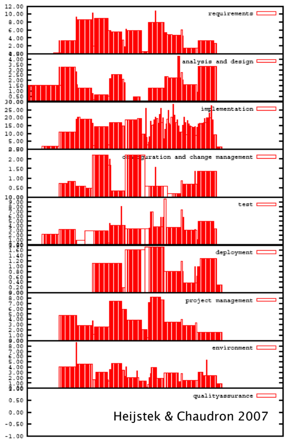
RUP Humps from 3 (largish) projects



Heijstek & Chaudron 2007

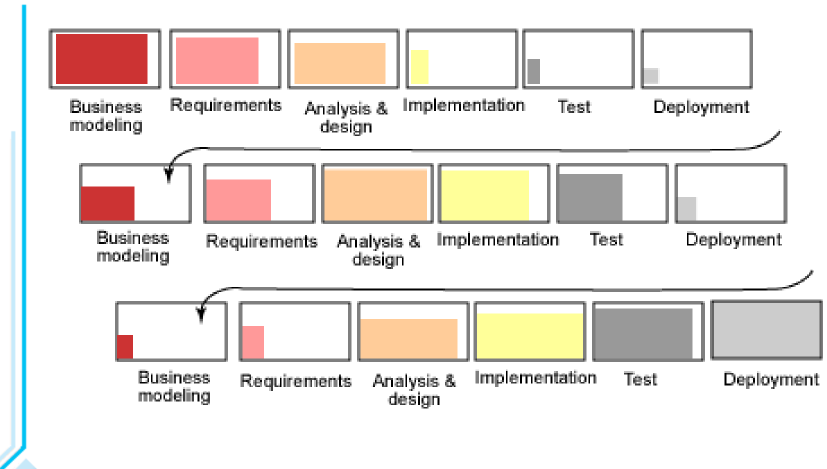


Heijstek & Chaudron 2007

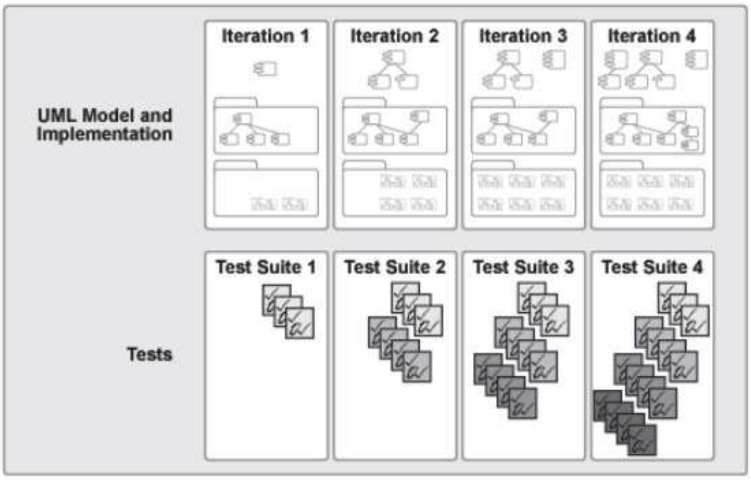


Heijstek & Chaudron 2007

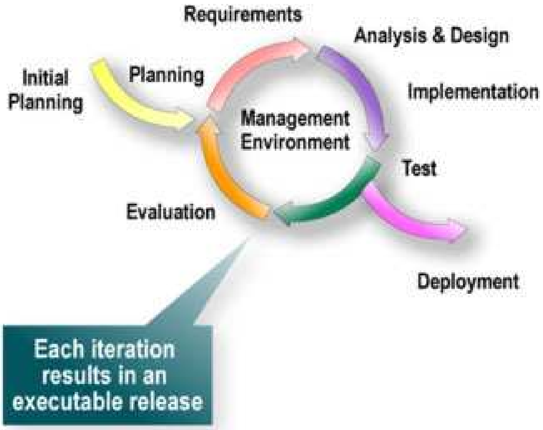
Progress perspective



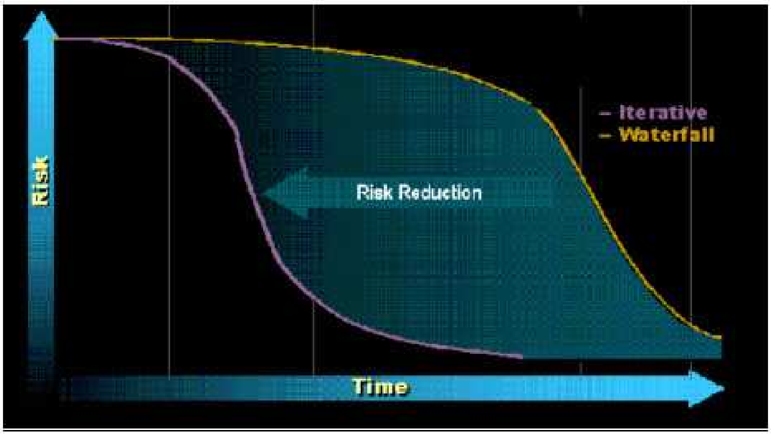
Progress perspective (alternative pic)



Iteration Perspective



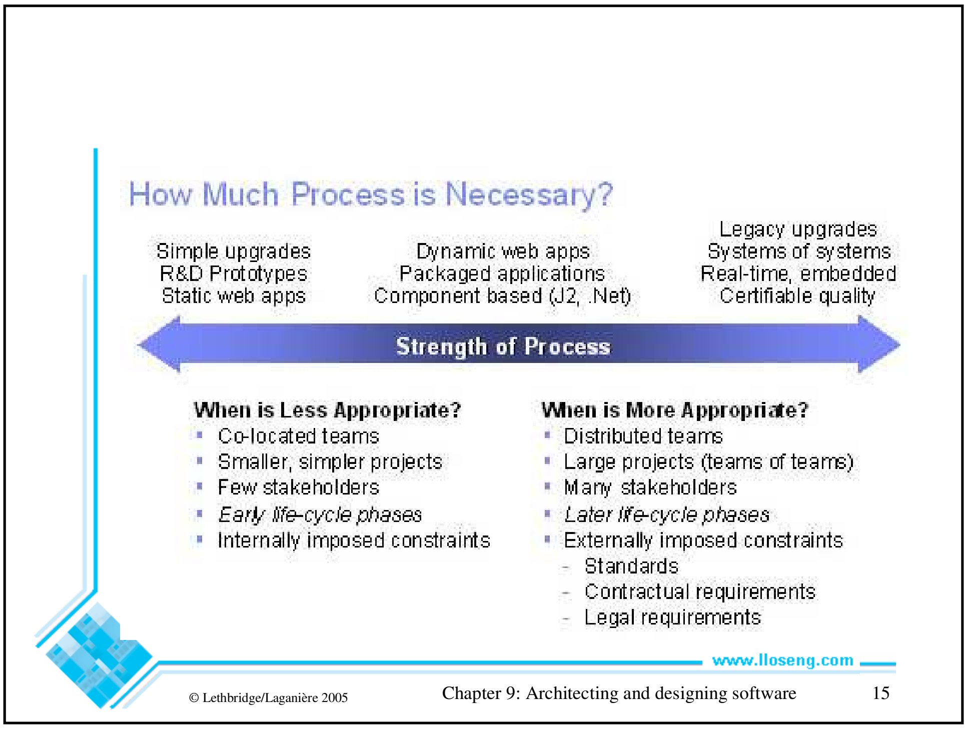
Incremental ➔ Risk reduction



Essentials of RUP

1. Develop software iteratively; Incrementally build and test
2. Manage requirements
3. Use component-based architectures
4. Visually model software
5. Verify software quality
6. Control changes to software

•Develop a Vision
•Manage to the Plan
•Identify and Mitigate Risks Early and regularly
•Examine the Business Case
•Provide User Support



How Much Process is Necessary?

Simple upgrades
R&D Prototypes
Static web apps

Dynamic web apps
Packaged applications
Component based (J2, .Net)

Legacy upgrades
Systems of systems
Real-time, embedded
Certifiable quality

Strength of Process

When is Less Appropriate?

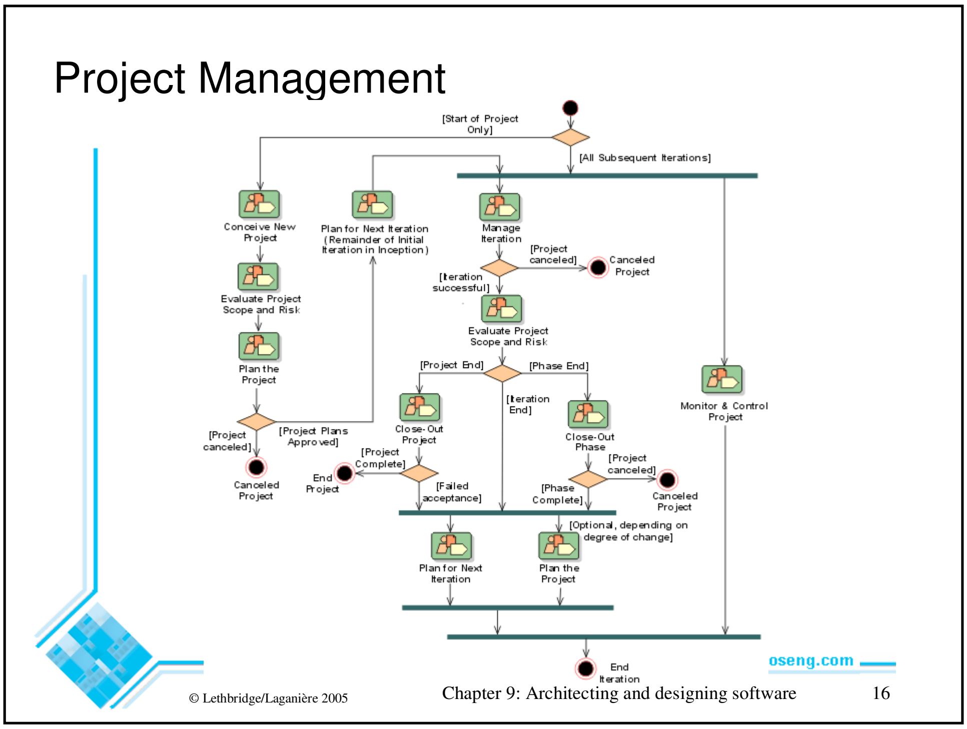
- Co-located teams
- Smaller, simpler projects
- Few stakeholders
- *Early life-cycle phases*
- Internally imposed constraints

When is More Appropriate?

- Distributed teams
- Large projects (teams of teams)
- Many stakeholders
- *Later life-cycle phases*
- Externally imposed constraints
  - Standards
  - Contractual requirements
  - Legal requirements

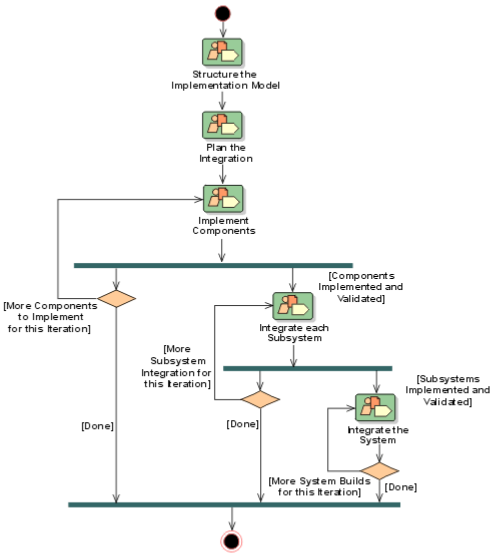
www.lloseng.com

Project Management



oseng.com

Implementation



Structure the Implementation Model

Plan the Integration

Implement Components

[Components Implemented and Validated]

[More Components to Implement for this Iteration]

Integrate each Subsystem

[More Subsystem Integration for this Iteration]

[Subsystems Implemented and Validated]

Integrate the System

[Done]

[Done]

[More System Builds for this Iteration]

[Done]

RUP Tooling

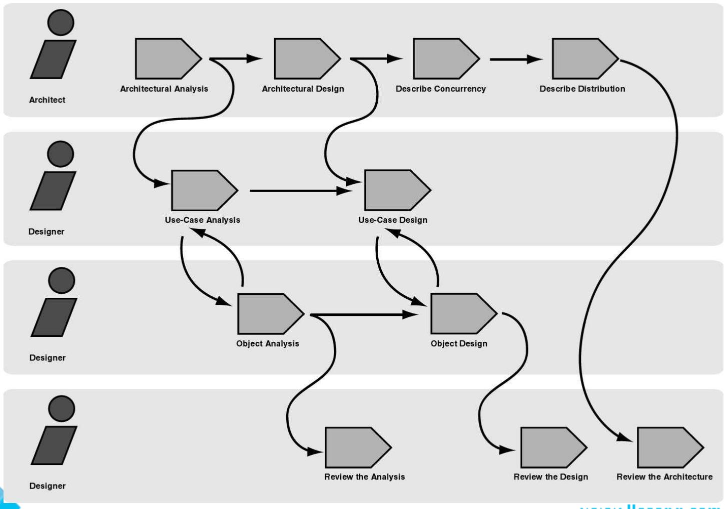
Describes processes in terms of:

- **workflows**
- **roles**
- **artifacts**

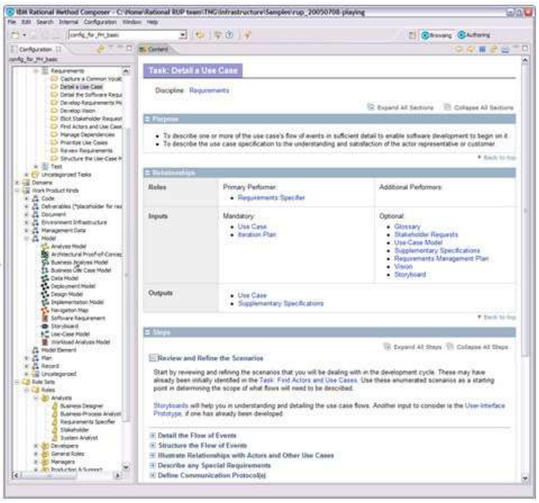
Provides

- **templates for deliverables**

RUP workflow



Tooling


IBM Rational Method Composer - C:\Home\Rational RUP team\TNG\infrastructure\Samples\rup_20050708-playing
File Edit Search Internal Configuration Window Help
config_for_rm_basic
Browsing
Authoring
Configuration
Content
Expand All Sections
Collapse All Sections
Back to top
Expand All Steps
Collapse All Steps

Task: Detail a Use Case

Discipline: Requirements

Purpose

- To describe one or more of the use case's flow of events in sufficient detail to enable software development to begin on it
- To describe the use case specification to the understanding and satisfaction of the actor representative or customer

Relationships

Roles	Primary Performer: • Requirements Specifier	Additional Performers:
Inputs	Mandatory: • Use Case • Iteration Plan	Optional: • Glossary • Stakeholder Requests • Use-Case Model • Supplementary Specifications • Requirements Management Plan • Vision • Storyboard
Outputs	• Use Case • Supplementary Specifications	

Steps

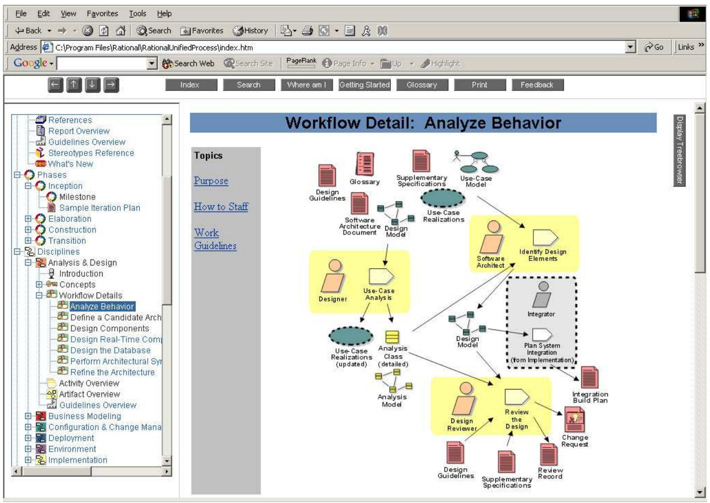
Review and Refine the Scenarios

Start by reviewing and refining the scenarios that you will be dealing with in the development cycle. These may have already been initially identified in the Task: Find Actors and Use Cases. Use these enumerated scenarios as a starting point in determining the scope of what flows will need to be described.

Storyboards will help you in understanding and detailing the use case flows. Another input to consider is the User-Interface Prototype, if one has already been developed.

- Detail the Flow of Events
- Structure the Flow of Events
- Illustrate Relationships with Actors and Other Use Cases
- Describe any Special Requirements
- Define Communication Protocol(s)

Tooling



Design

9.1 The Process of Design

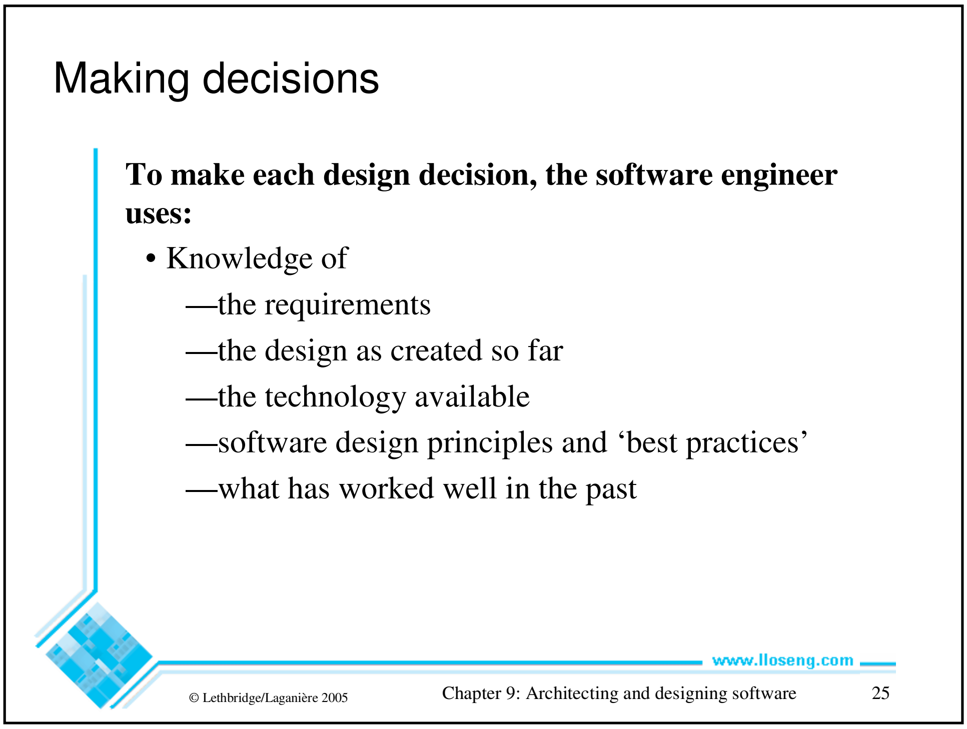
Definition:

- *Design* is a problem-solving process whose objective is to find and describe a way:
  - —To implement the system's *functional requirements*...
  - —While respecting the constraints imposed by the *quality, platform and process requirements...*
    - including the budget
  - —And while adhering to general principles of *good quality*

Design as a series of decisions

A designer is faced with a series of *design issues*

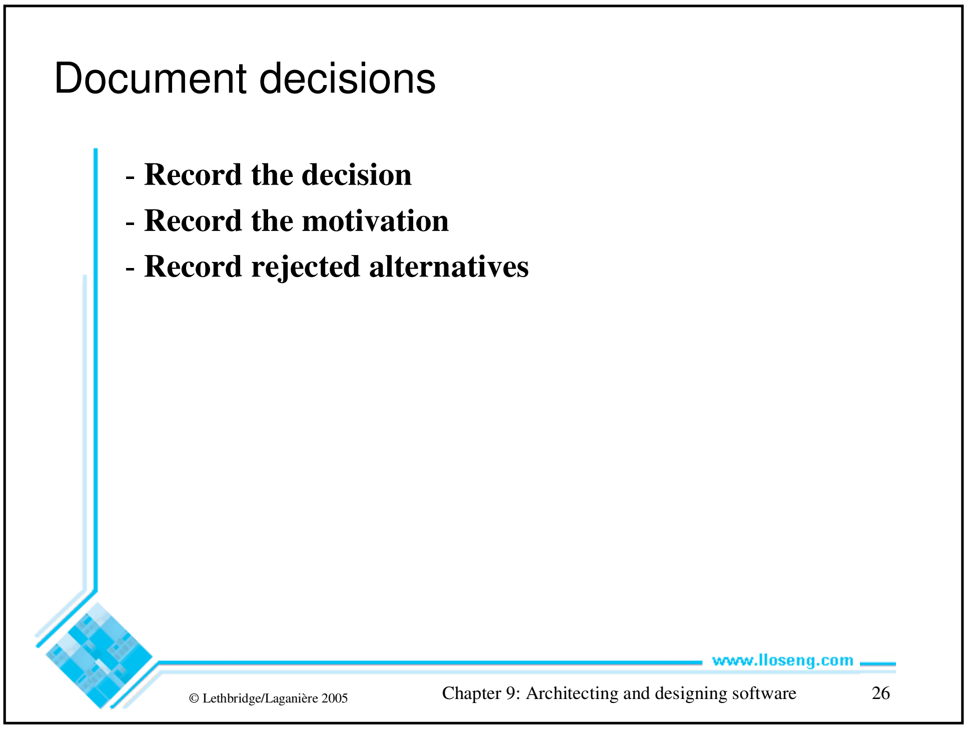
- These are sub-problems of the overall design problem.
- Each issue normally has several alternative solutions:
  - —design *options*.
- The designer makes a *design decision* to resolve each issue.
  - —This process involves choosing the best option from among the alternatives.



Making decisions

To make each design decision, the software engineer uses:

- Knowledge of
  - —the requirements
  - —the design as created so far
  - —the technology available
  - —software design principles and ‘best practices’
  - —what has worked well in the past



Document decisions

- Record the decision

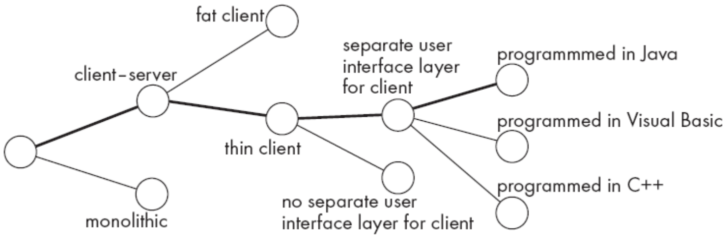
- Record the motivation

- Record rejected alternatives

Design space

The space of possible designs that could be achieved by choosing different sets of alternatives is often called the *design space*

- For example:


fat client
client-server
separate user interface layer for client
programmmed in Java
programmed in Visual Basic
thin client
programmed in C++
monolithic
no separate user interface layer for client

Features

According to

FODA: A prominent and user-visible aspect, quality or characteristic of a system.

ODM: A distinguishable characteristic of a system that is relevant to a stakeholder of the system

In mobile telephones:
- polyphonic ringtones
- SMS, MMS
- dual, tri-band,
- ...

In cars:
- airco
- power-steering
- remote key-lock
- ...

Feature models

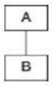
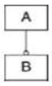
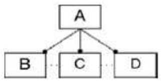
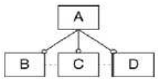
Types of features

Mandatory: All systems must have it
e.g. A car must have an engine

Alternative:
A system must have one out of multiple options
e.g. Transmission may be manual or automatic

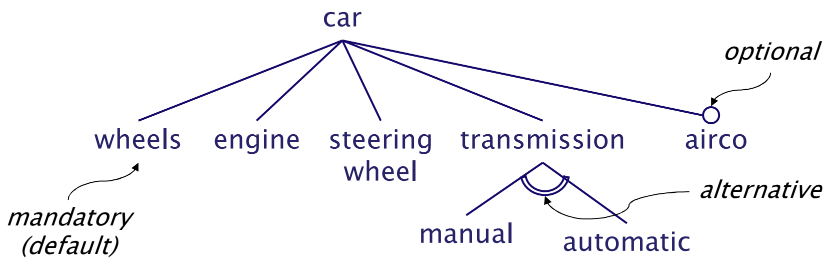
Optional: A system may have a feature
e.g. A car may have air-conditioning

Table 1
Explanation of feature diagram elements

Feature type	Graphical representation
Mandatory Mandatory feature B has to be included if its parent feature A is selected	
Optional Optional feature B may be included if its parent feature A is selected.	
Alternative Alternative features are organized in *alternative groups*. Exactly one feature of such a group B, C, D has to be selected if the group's parent feature A is selected.	
Or Or features are organized in *or groups*. At least one feature of such a group B, C, D has to be selected if the group's parent feature A is selected.	

Feature Diagram

A hierarchical decomposition of features.
A concept higher in the tree *consists of* its children



car

optional

wheels

engine

steering wheel

transmission

airco

alternative

mandatory (default)

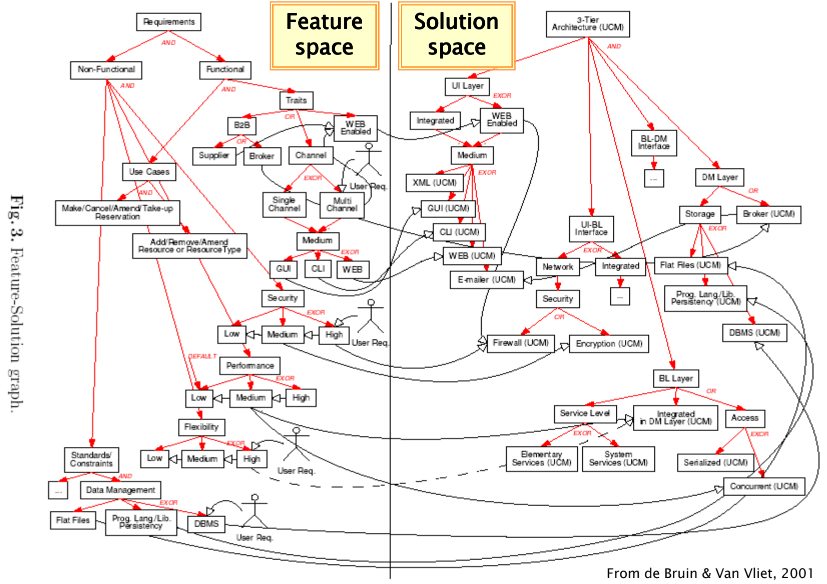
manual

automatic

Additional annotations that may be used in the feature diagram:

- mutually exclusive features
- rationale for chosing between alternatives
- composition rules: airco may be used if horsepower>100

Feature Solution Diagrams

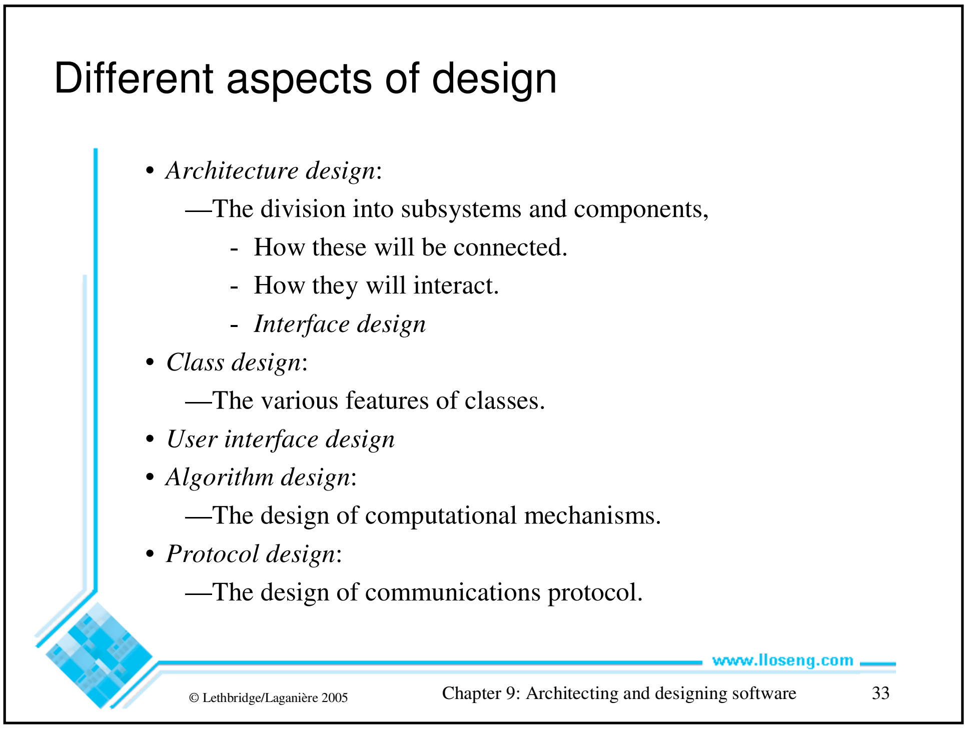


Feature space

Solution space

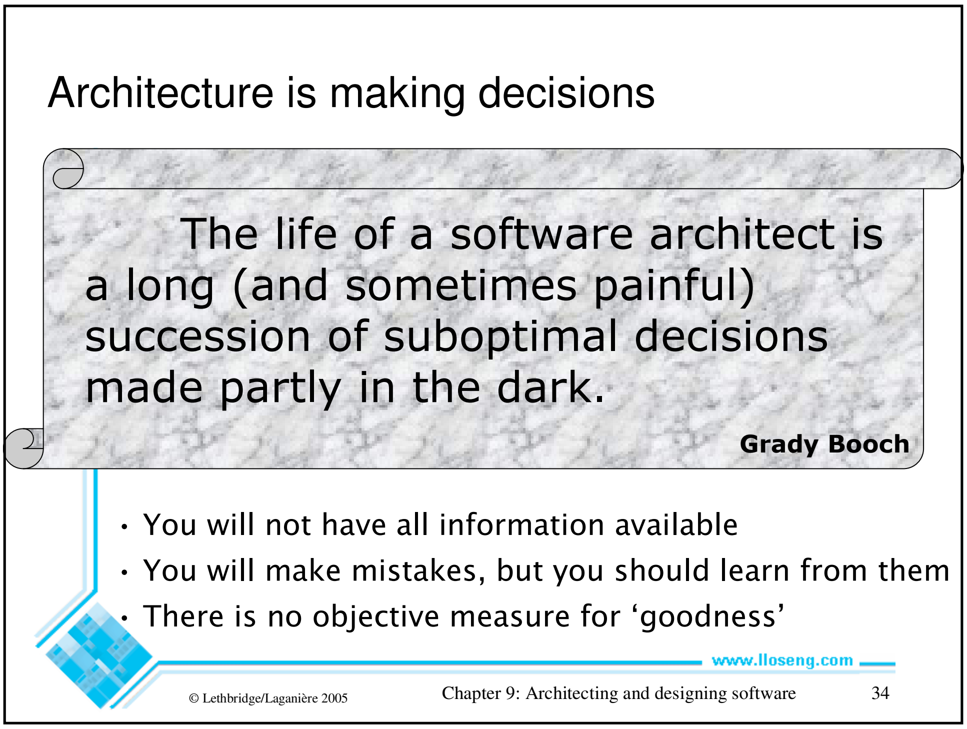
Fig. 3. Feature-Solution graph.

From de Bruin & Van Vliet, 2001



Different aspects of design

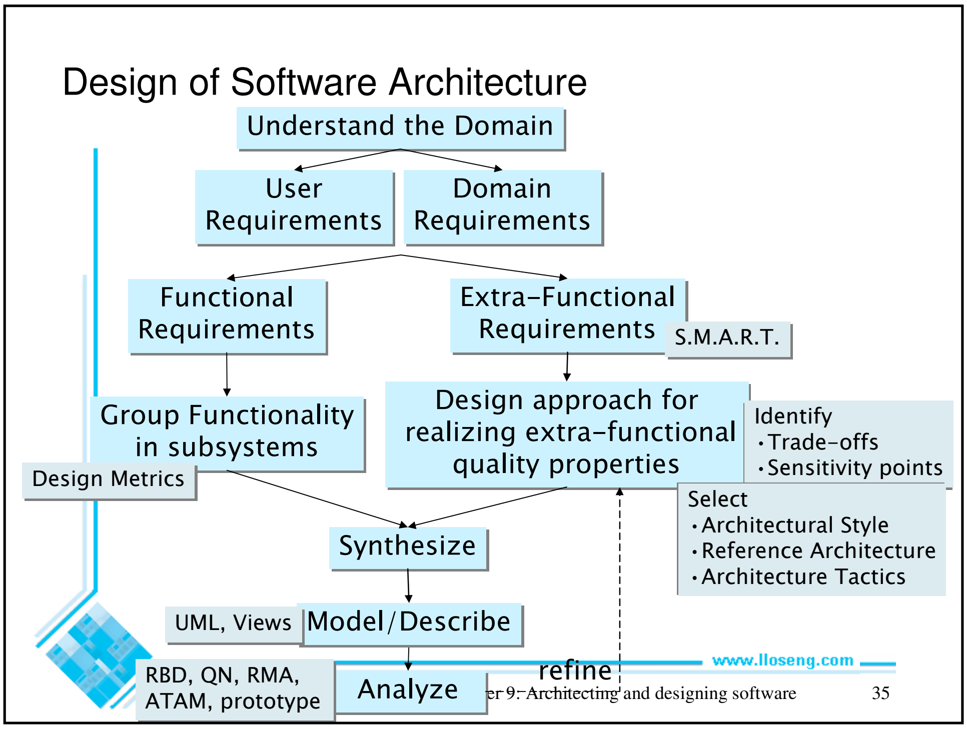
- *Architecture design*:
  - —The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - *Interface design*
- *Class design*:
  - —The various features of classes.
- *User interface design*
- *Algorithm design*:
  - —The design of computational mechanisms.
- *Protocol design*:
  - —The design of communications protocol.



Architecture is making decisions

> The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.
>
> **Grady Booch**

- You will not have all information available
- You will make mistakes, but you should learn from them
- There is no objective measure for ‘goodness’



Design of Software Architecture

Understand the Domain

User Requirements

Domain Requirements

Functional Requirements

Extra-Functional Requirements

S.M.A.R.T.

Group Functionality in subsystems

Design Metrics

Design approach for realizing extra-functional quality properties

Identify
- Trade-offs
- Sensitivity points

Select
- Architectural Style
- Reference Architecture
- Architecture Tactics

Synthesize

UML, Views

Model/Describe

RBD, QN, RMA, ATAM, prototype

Analyze

refine

Design Heuristics and Styles

(LL Chapter 9)

Michel Chaudron

Leiden Institute of Advanced Computer Science

Many slides based on Lethbridge and Laganiere

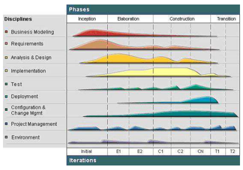
Agenda

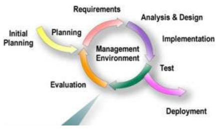
- Recap RUP
- Design heuristics & guidelines
- Architectural Styles

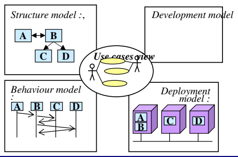
- This afternoon: geen werkcollege
- hand in assignments electronically

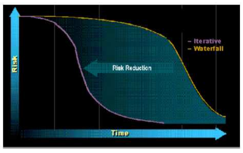
  chaudron@liacs.nl

Summary Rational Unified Process









Software Design Heuristics

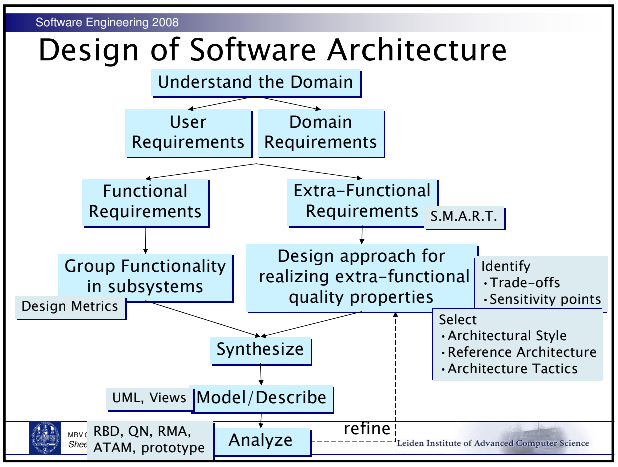
Different aspects of design

- □ *Architecture design*:
  - The division into subsystems and components,
    - □ How these will be connected:
    - □ How they will interact:
      - Interface design & architectural style
- □ *Class design*:
  - The various features of classes.
- □ *User interface design*
- □ *Algorithm design*:
  - The design of computational mechanisms.
- □ *Protocol design*:
  - The design of communications protocol.

Architecture is making decisions

> **The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.**
>
> **Grady Booch**

- You will not have all information available
- You will make mistakes, but you should learn from them
- There is no objective measure for ‘goodness’



Design of Software Architecture

Understand the Domain

User Requirements

Domain Requirements

Functional Requirements

Extra-Functional Requirements

S.M.A.R.T.

Group Functionality in subsystems

Design Metrics

Design approach for realizing extra-functional quality properties

Identify
- Trade-offs
- Sensitivity points

Select
- Architectural Style
- Reference Architecture
- Architecture Tactics

Synthesize

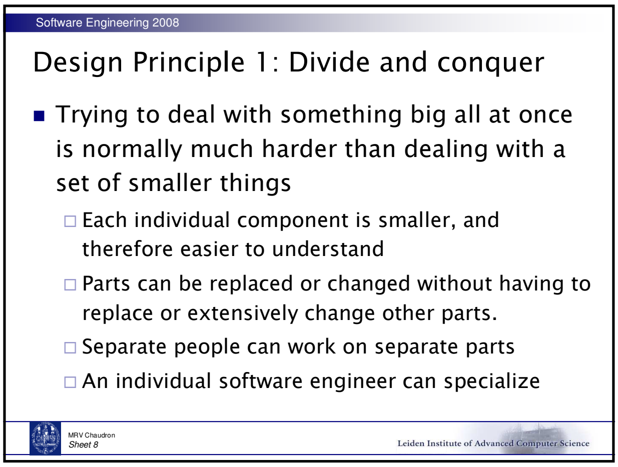
UML, Views

Model/Describe

RBD, QN, RMA, ATAM, prototype

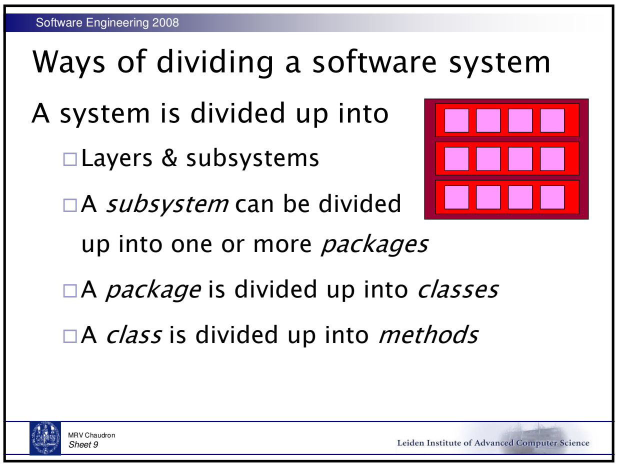
Analyze

refine



Design Principle 1: Divide and conquer

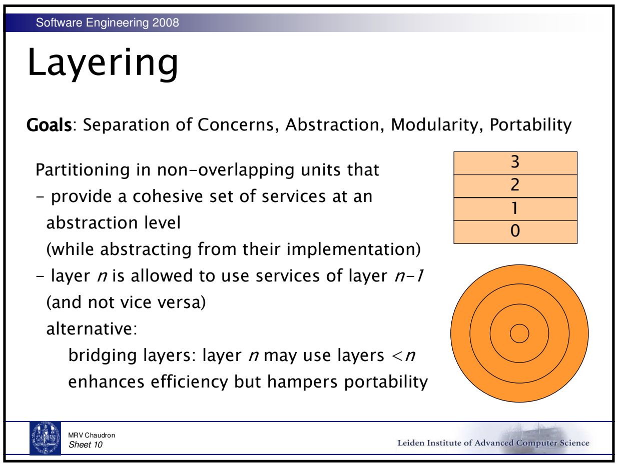
- Trying to deal with something big all at once is normally much harder than dealing with a set of smaller things
  - □ Each individual component is smaller, and therefore easier to understand
  - □ Parts can be replaced or changed without having to replace or extensively change other parts.
  - □ Separate people can work on separate parts
  - □ An individual software engineer can specialize



Ways of dividing a software system

A system is divided up into

- □ Layers & subsystems
- □ A *subsystem* can be divided up into one or more *packages*
- □ A *package* is divided up into *classes*
- □ A *class* is divided up into *methods*

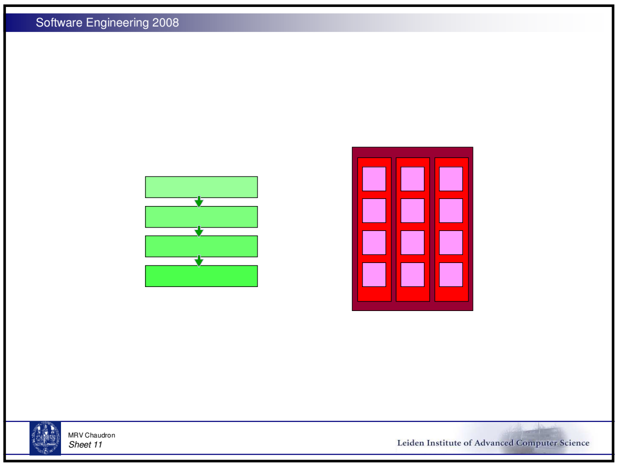


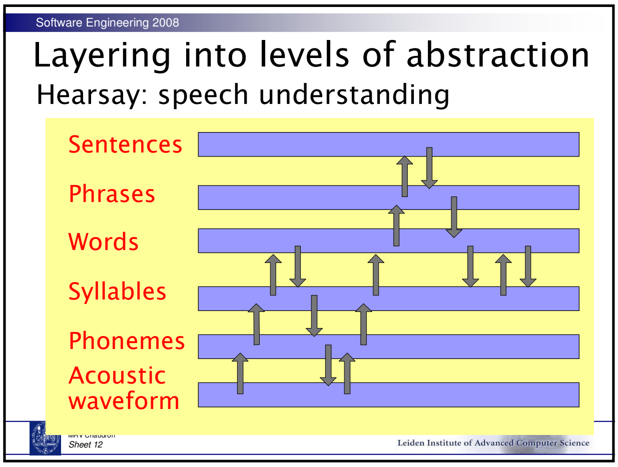
Layering

Goals: Separation of Concerns, Abstraction, Modularity, Portability

Partitioning in non-overlapping units that
- provide a cohesive set of services at an abstraction level
  (while abstracting from their implementation)
- layer *n* is allowed to use services of layer *n-1*
  (and not vice versa)

alternative:
bridging layers: layer *n* may use layers <*n*
enhances efficiency but hampers portability





Layering into levels of abstraction

Hearsay: speech understanding

Layering in Client / Server

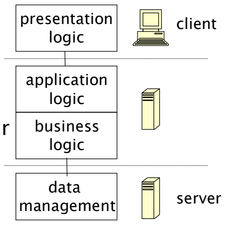
- **Presentation layer**

  Dialogue with users
- **Application logic**

  Application for individual user
- **Business logic**

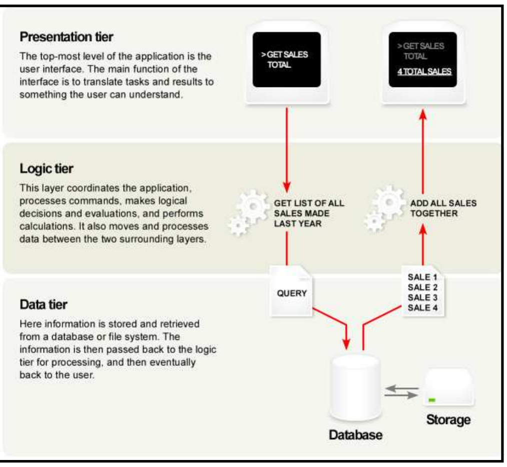
  Logic for processing across users, divisions
- **Data management**

  Storage of data



Unit of change
Unit of responsibility
Unit of deployment

Example 3-tier System



Presentation tier

The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

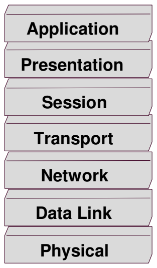
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

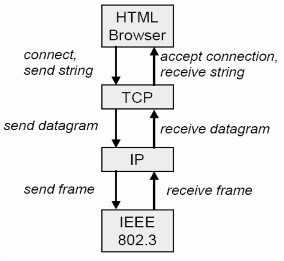
Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.

Diagram from Wikipedia, 2007

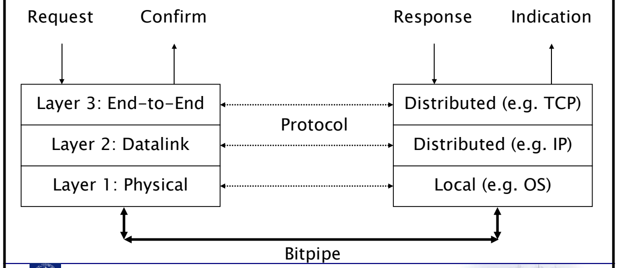
Layering in Computer Networks: OSI & Internet





Picture from Jeremy Bradbury, Queens Univ. Canada

Layering (2)
Example: Communication Stack



A Component-based Reference Architecture for Computer Games

(E. Folmer, 2007)

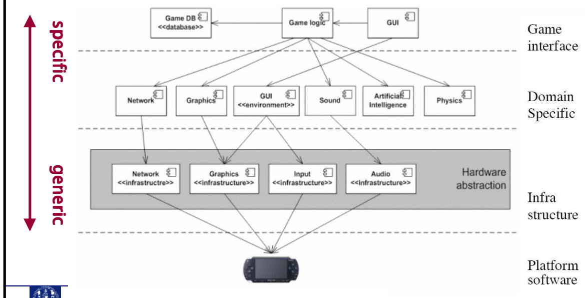
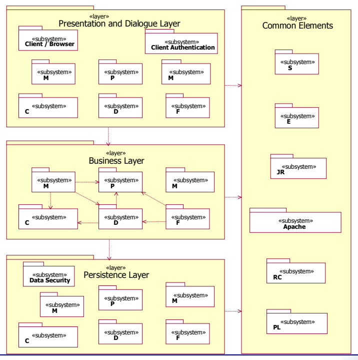
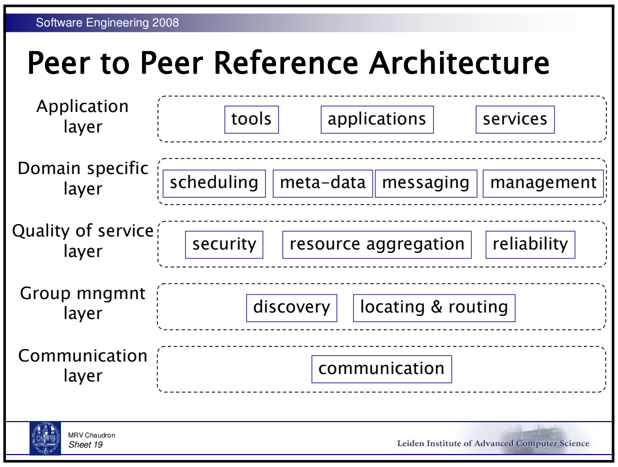


Fig. 1. A reference architecture for the games domain





Peer to Peer Reference Architecture

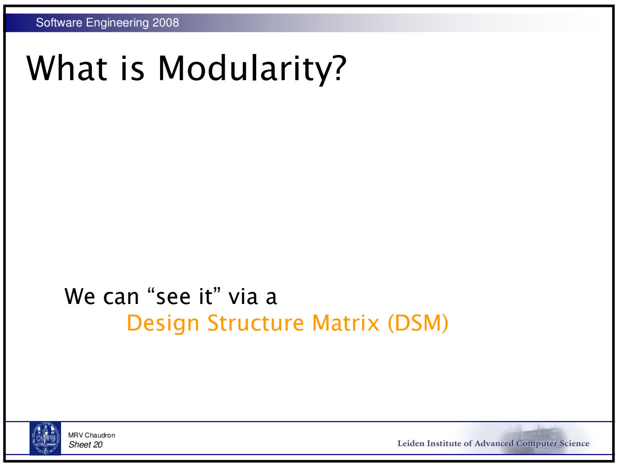
Application layer: tools | applications | services

Domain specific layer: scheduling | meta-data | messaging | management

Quality of service layer: security | resource aggregation | reliability

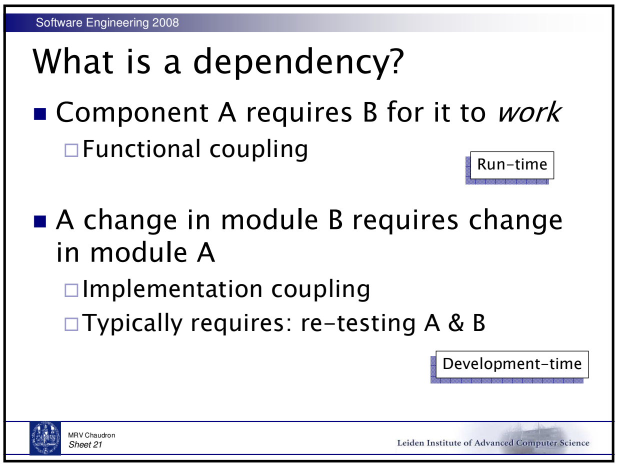
Group mngmnt layer: discovery | locating & routing

Communication layer: communication



What is Modularity?

We can “see it” via a
Design Structure Matrix (DSM)



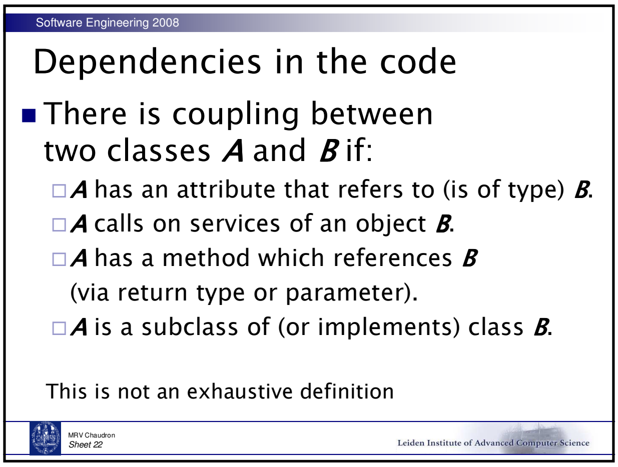
What is a dependency?

- Component A requires B for it to *work*
  - □Functional coupling

Run-time

- A change in module B requires change in module A
  - □Implementation coupling
  - □Typically requires: re-testing A & B

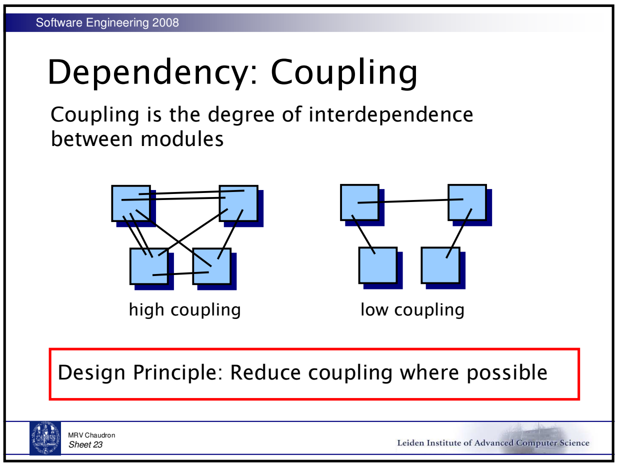
Development-time



Dependencies in the code

- There is coupling between two classes ***A*** and ***B*** if:
  - □***A*** has an attribute that refers to (is of type) ***B***.
  - □***A*** calls on services of an object ***B***.
  - □***A*** has a method which references ***B*** (via return type or parameter).
  - □***A*** is a subclass of (or implements) class ***B***.

This is not an exhaustive definition



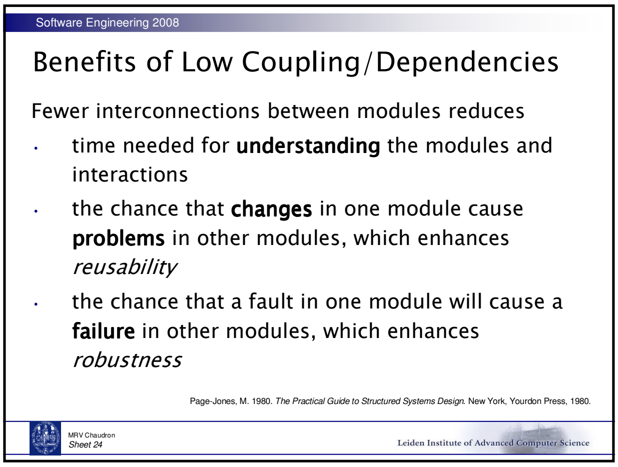
Dependency: Coupling

Coupling is the degree of interdependence between modules

high coupling

low coupling

Design Principle: Reduce coupling where possible



Benefits of Low Coupling/Dependencies

Fewer interconnections between modules reduces

- time needed for **understanding** the modules and interactions
- the chance that **changes** in one module cause **problems** in other modules, which enhances *reusability*
- the chance that a fault in one module will cause a **failure** in other modules, which enhances *robustness*

Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. New York, Yourdon Press, 1980.

Guideline: Minimize Dependency

Avoid dependencies where possible:

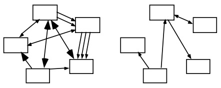
Design components so that

- □ they know about as few other components as possible
  - ▪ use as few parameters as possible
- □ for as short a time as possible
  - ▪ minimize number of calls between components

Ref: Component are from Mars – Chaudron & De Jong

Design Principle: Reduce coupling where possible

- ▪ *Coupling* occurs when there are *interdependencies* between one module and another
  - □ When interdependencies exist, changes in one place will require changes somewhere else.
  - □ A network of interdependencies makes it hard to see at a glance how some component works.
  - □ Type of coupling:
    - ▪ Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Separation of Concerns

- ▪ Zaken die niet bij elkaar horen moeten in verschillende eenheden (componenten / procedures / ..) worden geaddresseerd

Example Design Principles

Telecom Domain:

Separate the encoding/decoding of a message from the handling of a message, so

- ▪ **decode1 ; decode2 ; decode3 ; action1 ; action2**

handle

encode/ decode

And not

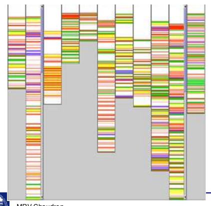
- ▪ **decode1 ; action1 ; decode2 ; action2 ; decode3**

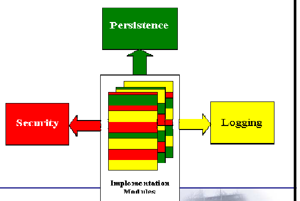
handle & encode/ decode

Aspect Orientation

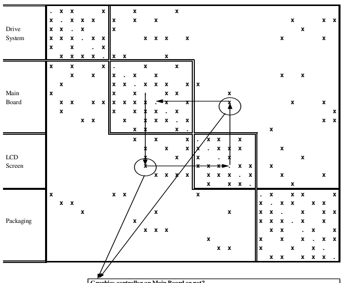
Design & maintain concerns in isolation

Automatically construct implementation by ‘weaving’ concerns



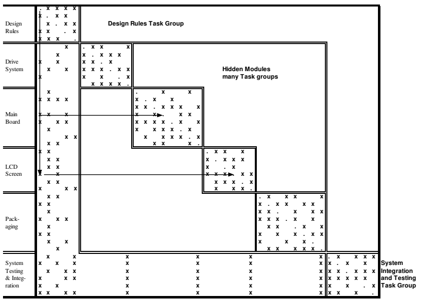


Design Structure Matrix Map of a Laptop Computer

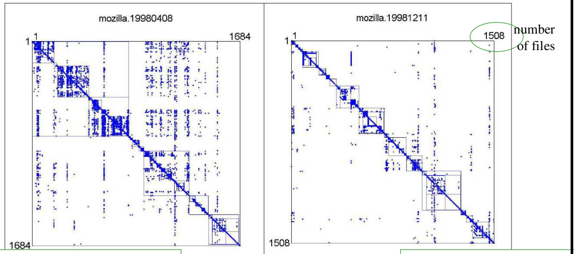


Graphics controller on Main Board or not?
If yes, screen specifications change;
If no, CPU must process more; adopt different interrupt protocols

Design Structure Matrix Map of a Modular System


Design Rules
Drive System
Main Board
LCD Screen
Pack-aging
System Testing & Integ-ration
Design Rules Task Group
Hidden Modules many Task groups
System Integration and Testing Task Group

DSM of Mozilla before and after redesign



2.4 dependencies per KSLOC

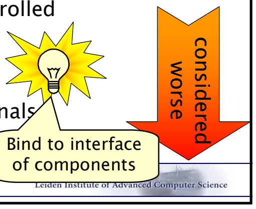
1.3 dependencies per KSLOC

Formerly Mozilla was the commercial Netscape Navigator, then released into open source.

From: Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, Alan MacCormack, John Rusnak, Carliss Baldwin, Harvard Business School, draft October 1st 2005

Types of Coupling

- Data coupling
  - data from one module is used in another
- Data type coupling
  - two modules use the same data type
- Control coupling
  - actions one module are controlled by another module (switch)
- Content coupling
  - a module refers to the internals of another module



Bind to interface of components

9.9 Difficulties and Risks in Design

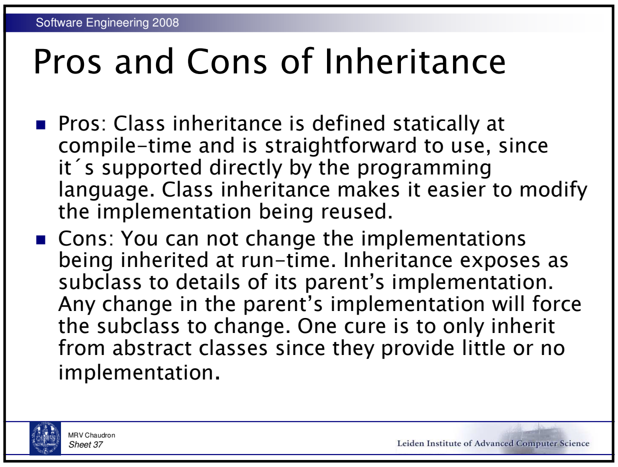
- Like modelling, design is a skill that requires considerable experience
  - *Individual software engineers should not attempt the design of large systems*
  - *Aspiring software architects should actively study designs of other systems*
- Poor designs can lead to expensive maintenance
  - *Ensure you follow the principles discussed in this chapter*

Difficulties and Risks in Design

- It requires constant effort to ensure a software system’s design remains good throughout its life
  - *Make the original design as flexible as possible so as to anticipate changes and extensions.*
  - *Ensure that the design documentation is usable and at the correct level of detail*
  - *Ensure that change is carefully managed*

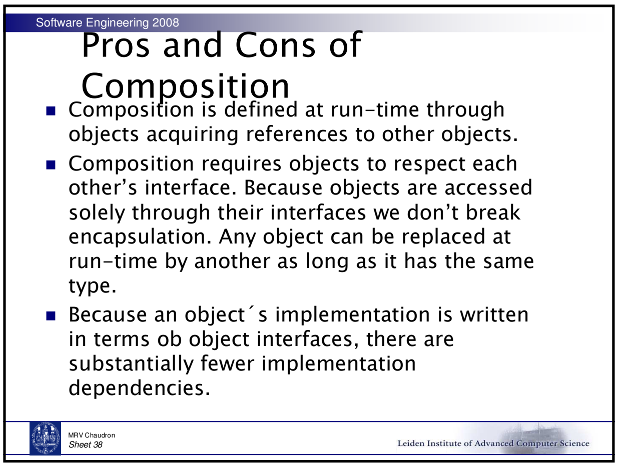
Inheritance vs. Composition

- The two most common techniques for reusing functionality in object-oriented systems are *class inheritance* and *object composition*
- Class inheritance defines the implementation of one class in terms of another’s implementation. With inheritance the internals of parent classes are often visible to sub-classes (*white box*).
- In object composition new functionality is obtained by assembling or composing objects to get more complex functionality. Internal details of objects are not visible, objects appear as *black boxes*.



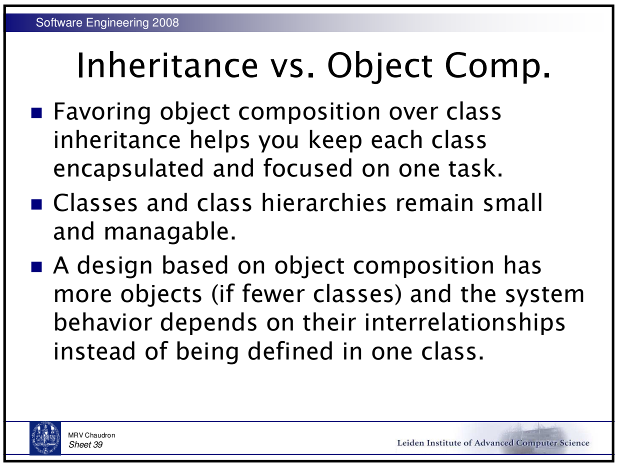
Pros and Cons of Inheritance

- Pros: Class inheritance is defined statically at compile-time and is straightforward to use, since it´s supported directly by the programming language. Class inheritance makes it easier to modify the implementation being reused.
- Cons: You can not change the implementations being inherited at run-time. Inheritance exposes as subclass to details of its parent's implementation. Any change in the parent's implementation will force the subclass to change. One cure is to only inherit from abstract classes since they provide little or no implementation.



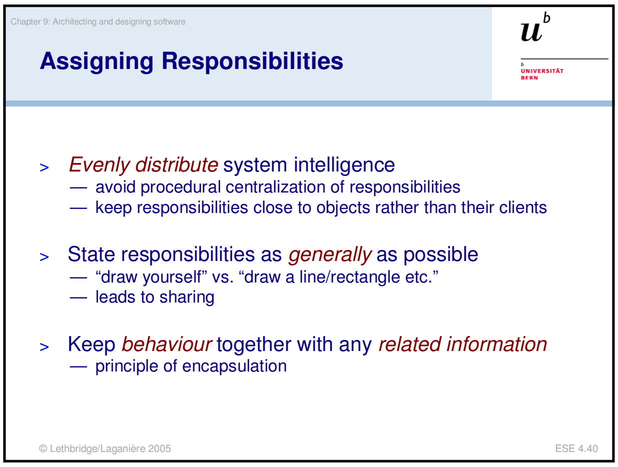
Pros and Cons of Composition

- Composition is defined at run-time through objects acquiring references to other objects.
- Composition requires objects to respect each other's interface. Because objects are accessed solely through their interfaces we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type.
- Because an object´s implementation is written in terms ob object interfaces, there are substantially fewer implementation dependencies.



Inheritance vs. Object Comp.

- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.
- Classes and class hierarchies remain small and managable.
- A design based on object composition has more objects (if fewer classes) and the system behavior depends on their interrelationships instead of being defined in one class.

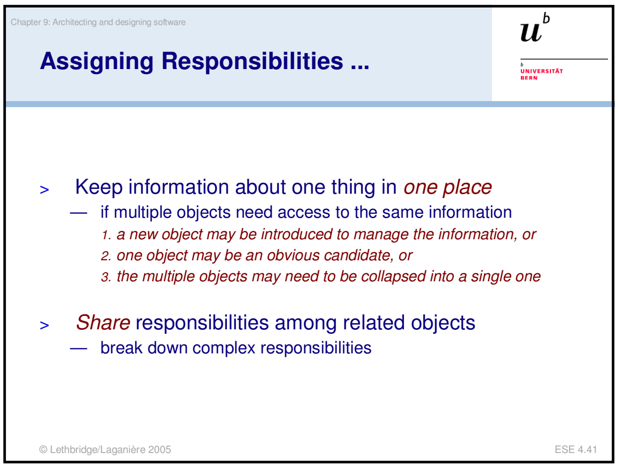


Assigning Responsibilities

> *Evenly distribute* system intelligence
>
> — avoid procedural centralization of responsibilities
>
> — keep responsibilities close to objects rather than their clients

> State responsibilities as *generally* as possible
>
> — "draw yourself" vs. "draw a line/rectangle etc."
>
> — leads to sharing

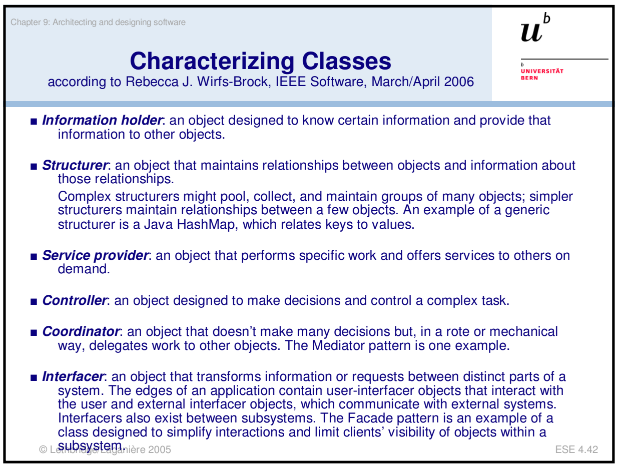
> Keep *behaviour* together with any *related information*
>
> — principle of encapsulation



Assigning Responsibilities ...

> Keep information about one thing in *one place*
>
> — if multiple objects need access to the same information
>
> 1. *a new object may be introduced to manage the information, or*
> 2. *one object may be an obvious candidate, or*
> 3. *the multiple objects may need to be collapsed into a single one*

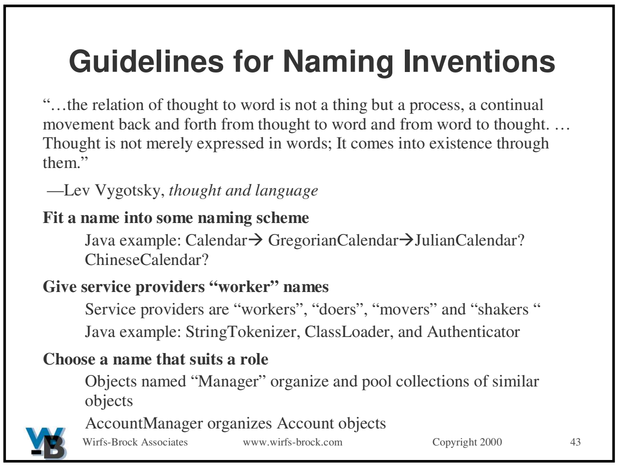
> *Share* responsibilities among related objects
>
> — break down complex responsibilities



Characterizing Classes

according to Rebecca J. Wirfs-Brock, IEEE Software, March/April 2006

- ***Information holder***: an object designed to know certain information and provide that information to other objects.
- ***Structurer***: an object that maintains relationships between objects and information about those relationships.
  Complex structurers might pool, collect, and maintain groups of many objects; simpler structurers maintain relationships between a few objects. An example of a generic structurer is a Java HashMap, which relates keys to values.
- ***Service provider***: an object that performs specific work and offers services to others on demand.
- ***Controller***: an object designed to make decisions and control a complex task.
- ***Coordinator***: an object that doesn't make many decisions but, in a rote or mechanical way, delegates work to other objects. The Mediator pattern is one example.
- ***Interfacer***: an object that transforms information or requests between distinct parts of a system. The edges of an application contain user-interfacer objects that interact with the user and external interfacer objects, which communicate with external systems. Interfacers also exist between subsystems. The Facade pattern is an example of a class designed to simplify interactions and limit clients' visibility of objects within a subsystem.



Guidelines for Naming Inventions

"…the relation of thought to word is not a thing but a process, a continual movement back and forth from thought to word and from word to thought. … Thought is not merely expressed in words; It comes into existence through them."

—Lev Vygotsky, *thought and language*

Fit a name into some naming scheme

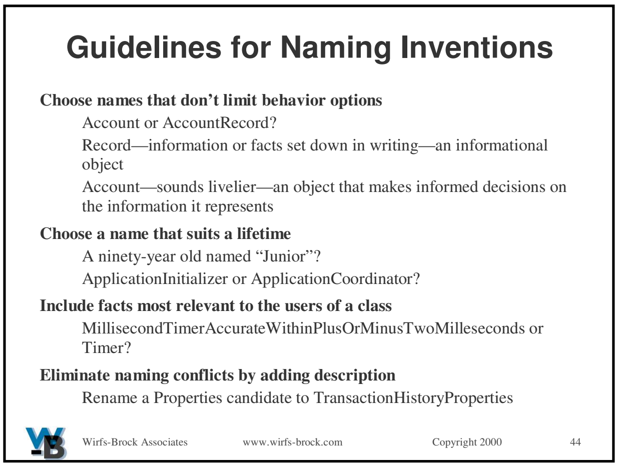
- Java example: Calendar→ GregorianCalendar→JulianCalendar? ChineseCalendar?

Give service providers "worker" names

- Service providers are "workers", "doers", "movers" and "shakers "
- Java example: StringTokenizer, ClassLoader, and Authenticator

Choose a name that suits a role

- Objects named "Manager" organize and pool collections of similar objects
- AccountManager organizes Account objects



Guidelines for Naming Inventions

Choose names that don't limit behavior options

- Account or AccountRecord?
- Record—information or facts set down in writing—an informational object
- Account—sounds livelier—an object that makes informed decisions on the information it represents

Choose a name that suits a lifetime

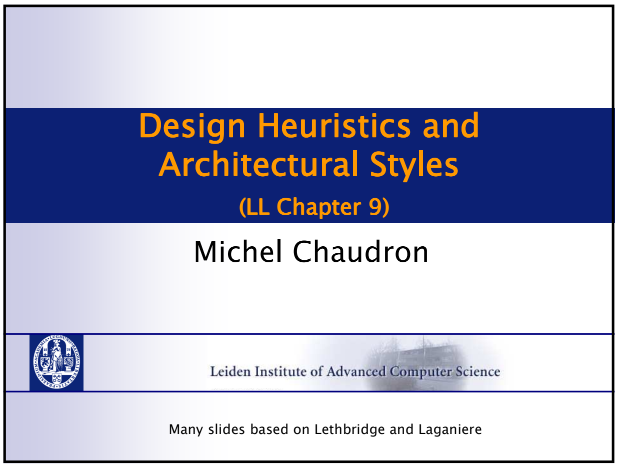
- A ninety-year old named "Junior"?
- ApplicationInitializer or ApplicationCoordinator?

Include facts most relevant to the users of a class

- MillisecondTimerAccurateWithinPlusOrMinusTwoMilleseconds or Timer?

Eliminate naming conflicts by adding description

- Rename a Properties candidate to TransactionHistoryProperties



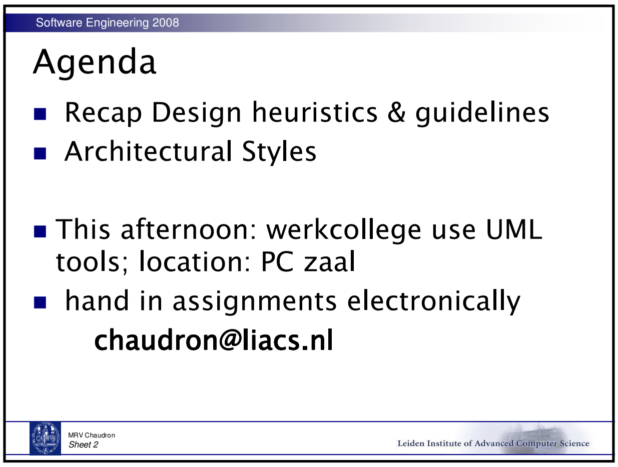
Design Heuristics and Architectural Styles

(LL Chapter 9)

Michel Chaudron

Leiden Institute of Advanced Computer Science

Many slides based on Lethbridge and Laganiere

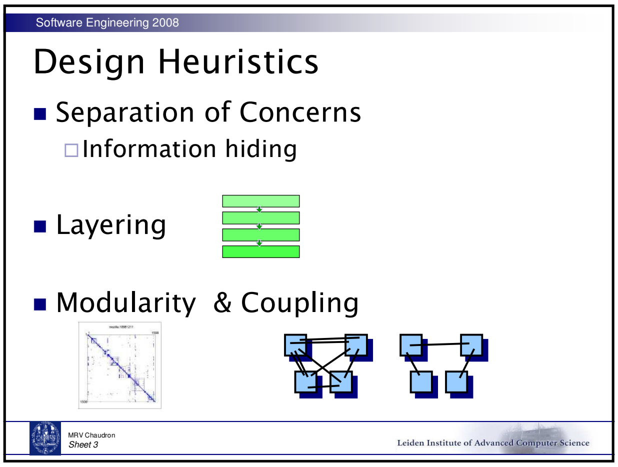


Agenda

- Recap Design heuristics & guidelines
- Architectural Styles

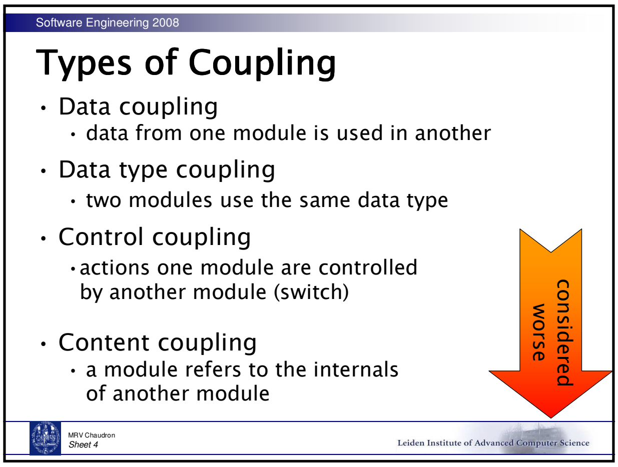
- This afternoon: werkcollege use UML tools; location: PC zaal
- hand in assignments electronically

  chaudron@liacs.nl



Design Heuristics

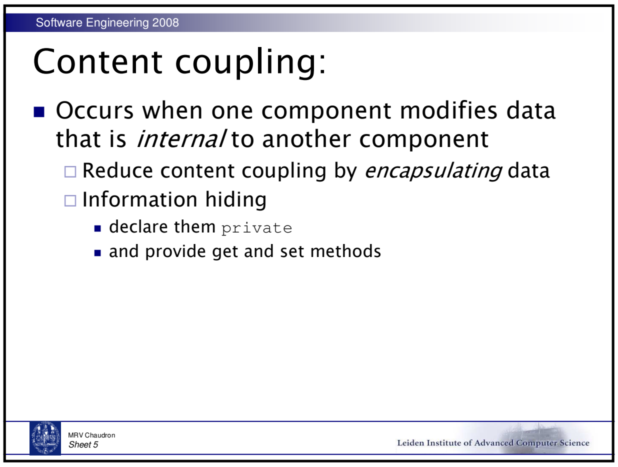
- Separation of Concerns
  - □ Information hiding
- Layering
- Modularity & Coupling



Types of Coupling

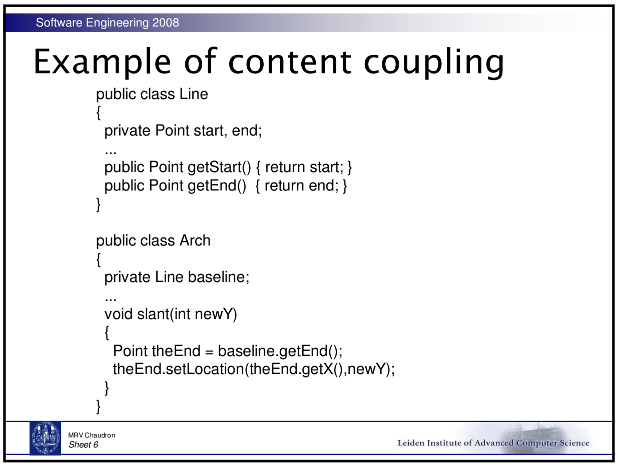
- Data coupling
  - data from one module is used in another
- Data type coupling
  - two modules use the same data type
- Control coupling
  - actions one module are controlled by another module (switch)
- Content coupling
  - a module refers to the internals of another module

considered worse



Content coupling:

- Occurs when one component modifies data that is *internal* to another component
  - □ Reduce content coupling by *encapsulating* data
  - □ Information hiding
    - declare them `private`
    - and provide get and set methods



Example of content coupling

```
public class Line
{
  private Point start, end;
  ...
  public Point getStart() { return start; }
  public Point getEnd()  { return end; }
}

public class Arch
{
  private Line baseline;
  ...
  void slant(int newY)
  {
    Point theEnd = baseline.getEnd();
    theEnd.setLocation(theEnd.getX(),newY);
  }
}
```

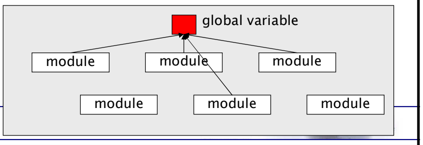
Information Hiding

- Usage of a module depends only on the information at the interface
- An interface should reveal as little as possible about the inner workings of the component
- An interface hides design decisions

D. L. Parnas, On the criteria to be used in decomposing systems into modules, *Communications of the ACM,* vol. 15, pp. 1053-1058, December 1972.

Common coupling

- Occurs whenever you use a *global variable*
  - □ All the components using the global variable become coupled to each other
  - □ A weaker form of common coupling is when a variable can be accessed by a *subset* of the system's classes
    - e.g. a Java package

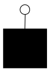


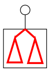
Control coupling

- Occurs when one procedure calls another using *a 'flag' or 'command'* that explicitly controls what the second procedure does
  - □ To make a change you have to change both the calling and called method
  - □ One way to reduce the control coupling could be to have a *look-up table*
    - commands are then mapped to a method that should be called when that command is issued

Example of control coupling

```
public routineX(String command)
{
  if (command.equals("drawCircle")
  {
    drawCircle();
  }
  else
  {
    drawRectangle();
  }
}
```





Caller needs to know:

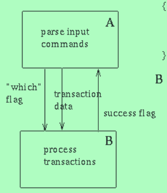
Not drawCircle => draw Rectangle

Control Coupling Example

control coupling

Two modules are control coupled if they communicate using at least one "control flag"

Example:



A code:

```
{...
  call B(t_info, "update",ok)
  if not(ok) then ...
  ...
}
```

B code:

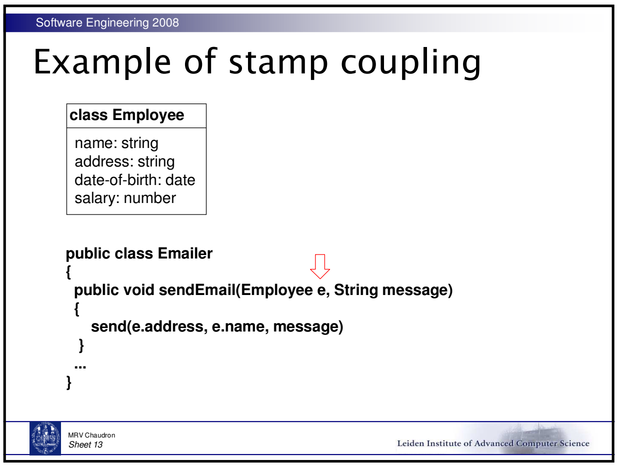
```
{receive parameters
    data, flag, outcome
 ...
  case (flag) of
    select: { ... }
    update: { ... }
    ...
    define: { ... }
    other: outcome := not ok
  end case
  outcome := ok
}
```

The behaviour of component B is controlled by component A through the parameter *flag*

Example from David Stotts
Dept. of Computer Science
University of North Carolina

Stamp coupling:

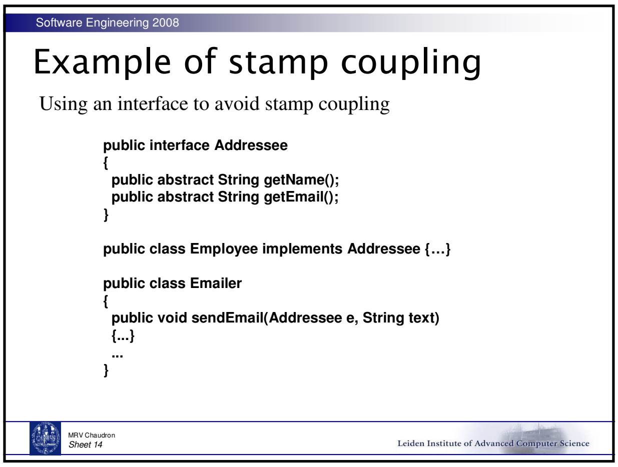
- Occurs whenever one of your application classes is declared as the *type* of a method argument
  - □ Since one class now uses the other, changing the system becomes harder
    - Reusing one class requires reusing the other
  - □ Two ways to reduce stamp coupling,
    - using an interface as the argument type
    - passing simple variables



Example of stamp coupling

class Employee
name: string address: string date-of-birth: date salary: number

```
public class Emailer
{
  public void sendEmail(Employee e, String message)
  {
     send(e.address, e.name, message)
   }
  ...
}
```



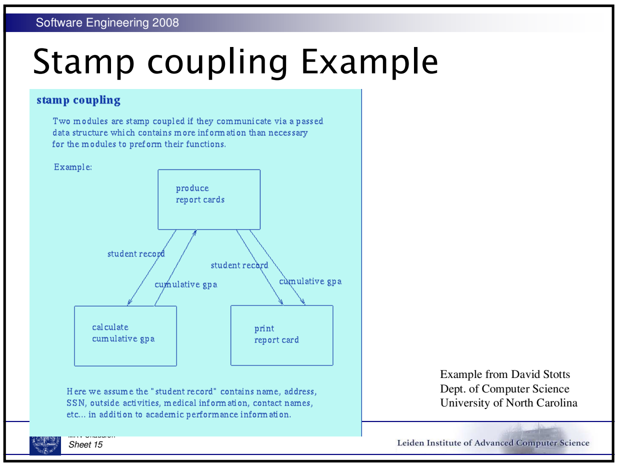
Example of stamp coupling

Using an interface to avoid stamp coupling

```
public interface Addressee
{
  public abstract String getName();
  public abstract String getEmail();
}

public class Employee implements Addressee {…}

public class Emailer
{
  public void sendEmail(Addressee e, String text)
  {…}
  ...
}
```



Stamp coupling Example

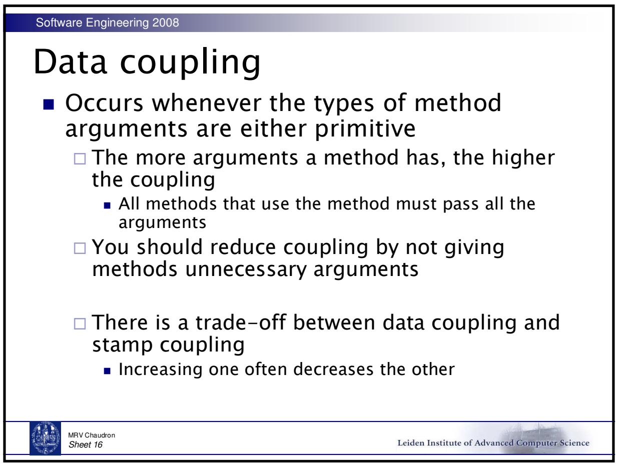
stamp coupling

Two modules are stamp coupled if they communicate via a passed data structure which contains more information than necessary for the modules to preform their functions.

Example:

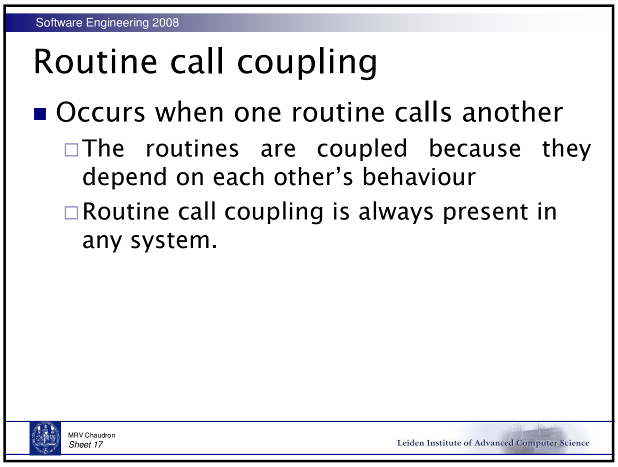
Here we assume the "student record" contains name, address, SSN, outside activities, medical information, contact names, etc... in addition to academic performance information.

Example from David Stotts
Dept. of Computer Science
University of North Carolina



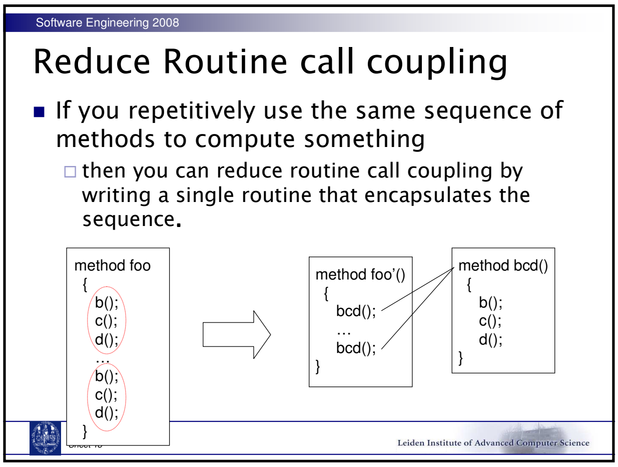
Data coupling

- Occurs whenever the types of method arguments are either primitive
  - □ The more arguments a method has, the higher the coupling
    - All methods that use the method must pass all the arguments
  - □ You should reduce coupling by not giving methods unnecessary arguments
  - □ There is a trade-off between data coupling and stamp coupling
    - Increasing one often decreases the other



Routine call coupling

- Occurs when one routine calls another
  - □ The routines are coupled because they depend on each other’s behaviour
  - □ Routine call coupling is always present in any system.



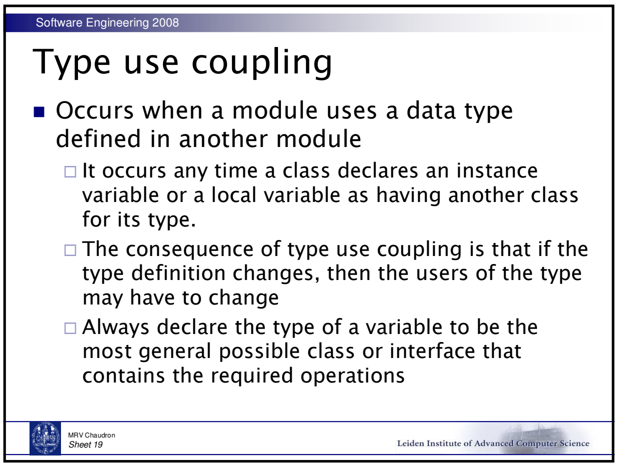
Reduce Routine call coupling

- If you repetitively use the same sequence of methods to compute something
  - □ then you can reduce routine call coupling by writing a single routine that encapsulates the sequence.

```
method foo
 {
   b();
   c();
   d();
   …
   b();
   c();
   d();
 }
```

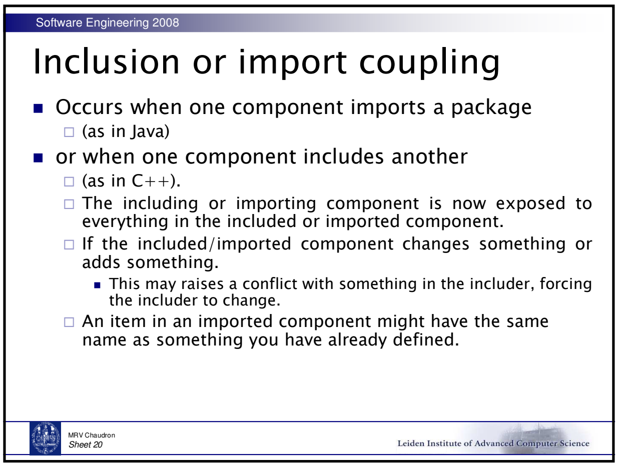
```
method foo’()
 {
   bcd();
   …
   bcd();
 }
```

```
method bcd()
 {
   b();
   c();
   d();
 }
```



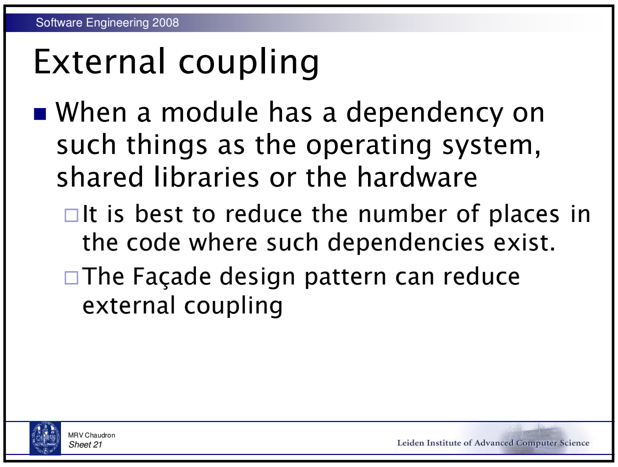
Type use coupling

- Occurs when a module uses a data type defined in another module
  - □ It occurs any time a class declares an instance variable or a local variable as having another class for its type.
  - □ The consequence of type use coupling is that if the type definition changes, then the users of the type may have to change
  - □ Always declare the type of a variable to be the most general possible class or interface that contains the required operations



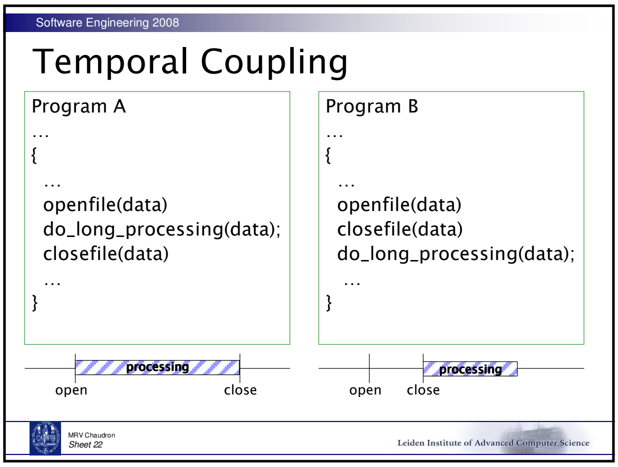
Inclusion or import coupling

- Occurs when one component imports a package
  - □ (as in Java)
- or when one component includes another
  - □ (as in C++).
  - □ The including or importing component is now exposed to everything in the included or imported component.
  - □ If the included/imported component changes something or adds something.
    - This may raises a conflict with something in the includer, forcing the includer to change.
  - □ An item in an imported component might have the same name as something you have already defined.



External coupling

- When a module has a dependency on such things as the operating system, shared libraries or the hardware
  - □ It is best to reduce the number of places in the code where such dependencies exist.
  - □ The Façade design pattern can reduce external coupling

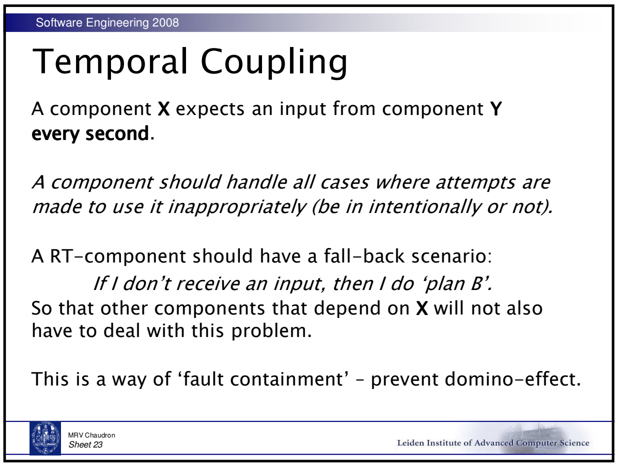


Temporal Coupling

```
Program A
…
{
  …
  openfile(data)
  do_long_processing(data);
  closefile(data)
  …
}
```

```
Program B
…
{
  …
  openfile(data)
  closefile(data)
  do_long_processing(data);
   …
}
```

processing
open
close
open
close
processing



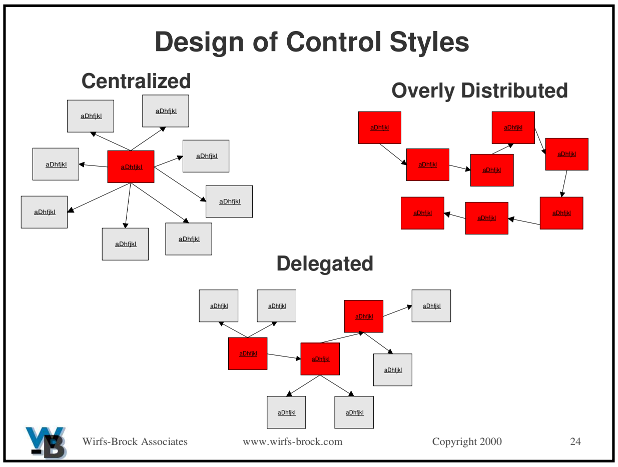
Temporal Coupling

A component **X** expects an input from component **Y** **every second**.

A component should handle all cases where attempts are made to use it inappropriately (be in intentionally or not).

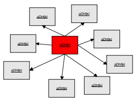
A RT-component should have a fall-back scenario:
If I don't receive an input, then I do 'plan B'.
So that other components that depend on **X** will not also have to deal with this problem.

This is a way of 'fault containment' – prevent domino-effect.



Design of Control Styles

Characteristics of Centralized Control



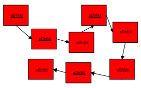
Centralized controllers can have extremely complex control logic

Controllers surrounded by simple information holders and service providers

These simple objects tend to have low-level, non-abstract interfaces

Drawback:
Changes can ripple among controlling and controlled objects

Characteristics of Overly Distributed Control

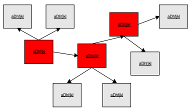


Long message chains to dig information out of information holders

Little or no value-added by those receiving a message and merely "delegating" request to next in chain

Drawback:
Hardwired dependencies between objects in call chain
May break encapsulation

Characteristics of Delegated Control



Coordinators know about fewer objects than dominating controllers

Higher level communications between objects

Benefits:
Changes typically localized and simpler
Easier to divide detailed design work

Interface Design

- An interface should reveal as little as possible about the inner workings of the component
- Users (callers) should depend only on the interface, not on the implementation

Recommended References:
- Effective Java: Programming Language Guide by Josh Bloch, Prentice Hall, 2001
  Check out video: http://www.infoq.com/presentations/effective-api-design
- Effective C++ by Scott Meyers, Addison-Wesley, 2005 (3rd ed).

Guidelines for Interface Design (1)

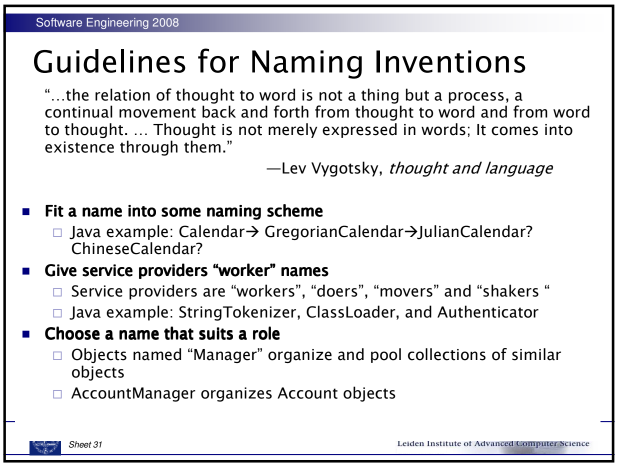
- *Completeness:*
  - include all functions
- *Essential/Minimal*:
  - omit needless features.
- *General:*
  - do not limit the applicability of an interface to its initial purpose as modules may be used in unexpected ways.
- *Consistency*
  - applies to many aspects of interface design such as naming conventions, parameter passing and exception handling.
- *Orthogonality*:
  - Keep independent features separately
  - Avoid offering the same service in multiple ways.
- *Open-ended:*
  - leave room for future expansion.
- *Opaqueness/Information-hiding:*
  - an interface should hide the details of the implementation.

Based on Hoffman [Hol90] based on o.a. Parnas

Guidelines for Interface Design (2)

1. Keep interfaces *cohesive* and *small* (in that order)
2. Use *different interfaces* for users of the interface that play *different roles* with respect to the functionality
3. Don't combine *generic* and *specific* functionality in the same interface
4. Group *optional* functionality in *separate* interfaces
5. Avoid the introduction of *convenience functions*
6. Use *strongly typed* interfaces
7. Use *systematic naming* conventions

From Henk Jonkers c.s., Philips Research 2002

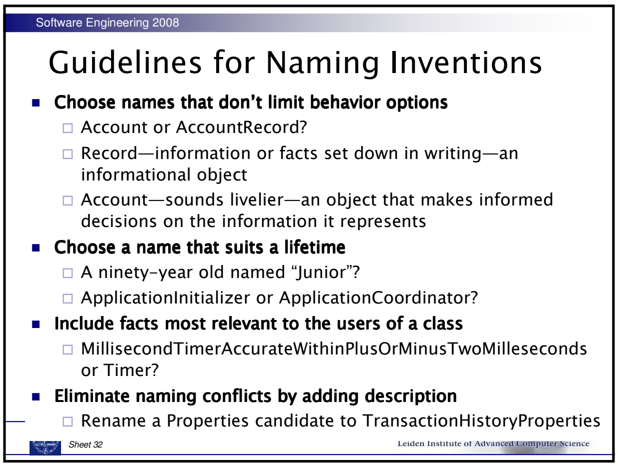


Guidelines for Naming Inventions

"…the relation of thought to word is not a thing but a process, a continual movement back and forth from thought to word and from word to thought. … Thought is not merely expressed in words; It comes into existence through them."

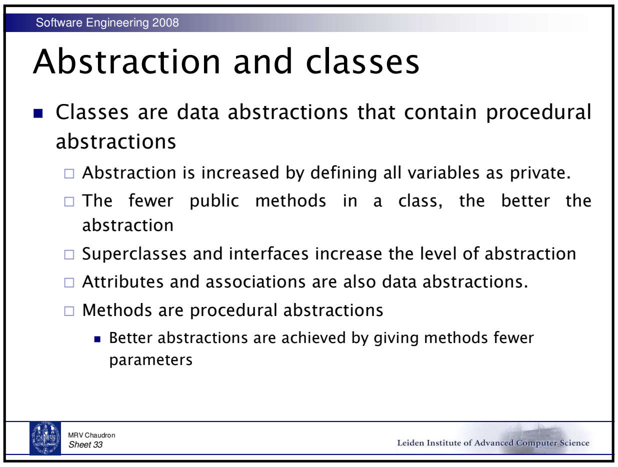
—Lev Vygotsky, *thought and language*

- **Fit a name into some naming scheme**
  - □ Java example: Calendar→ GregorianCalendar→JulianCalendar? ChineseCalendar?
- **Give service providers "worker" names**
  - □ Service providers are "workers", "doers", "movers" and "shakers "
  - □ Java example: StringTokenizer, ClassLoader, and Authenticator
- **Choose a name that suits a role**
  - □ Objects named "Manager" organize and pool collections of similar objects
  - □ AccountManager organizes Account objects



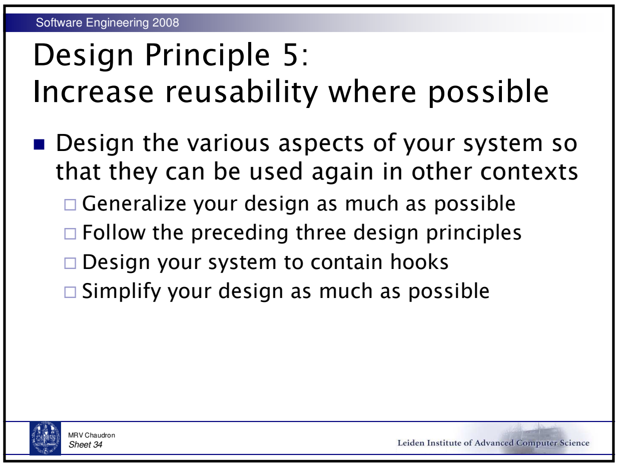
Guidelines for Naming Inventions

- **Choose names that don't limit behavior options**
  - □ Account or AccountRecord?
  - □ Record—information or facts set down in writing—an informational object
  - □ Account—sounds livelier—an object that makes informed decisions on the information it represents
- **Choose a name that suits a lifetime**
  - □ A ninety-year old named "Junior"?
  - □ ApplicationInitializer or ApplicationCoordinator?
- **Include facts most relevant to the users of a class**
  - □ MillisecondTimerAccurateWithinPlusOrMinusTwoMilleseconds or Timer?
- **Eliminate naming conflicts by adding description**
  - □ Rename a Properties candidate to TransactionHistoryProperties



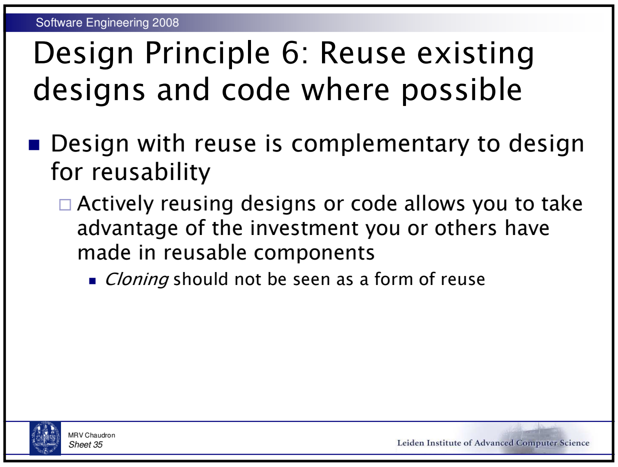
Abstraction and classes

- Classes are data abstractions that contain procedural abstractions
  - □ Abstraction is increased by defining all variables as private.
  - □ The fewer public methods in a class, the better the abstraction
  - □ Superclasses and interfaces increase the level of abstraction
  - □ Attributes and associations are also data abstractions.
  - □ Methods are procedural abstractions
    - Better abstractions are achieved by giving methods fewer parameters



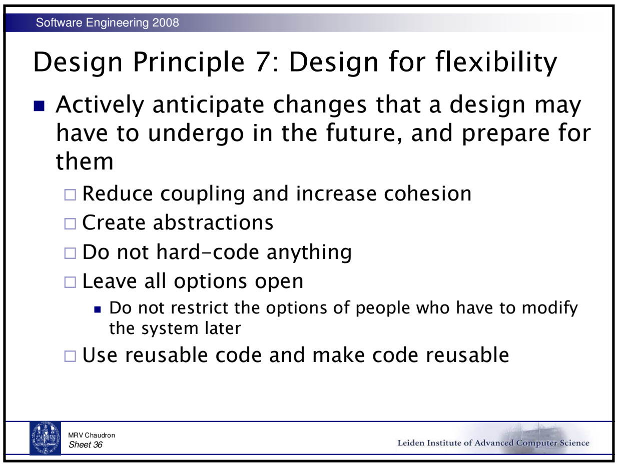
Design Principle 5: Increase reusability where possible

- Design the various aspects of your system so that they can be used again in other contexts
  - □ Generalize your design as much as possible
  - □ Follow the preceding three design principles
  - □ Design your system to contain hooks
  - □ Simplify your design as much as possible



Design Principle 6: Reuse existing designs and code where possible

- Design with reuse is complementary to design for reusability
  - □ Actively reusing designs or code allows you to take advantage of the investment you or others have made in reusable components
    - *Cloning* should not be seen as a form of reuse



Design Principle 7: Design for flexibility

- Actively anticipate changes that a design may have to undergo in the future, and prepare for them
  - □ Reduce coupling and increase cohesion
  - □ Create abstractions
  - □ Do not hard-code anything
  - □ Leave all options open
    - Do not restrict the options of people who have to modify the system later
  - □ Use reusable code and make code reusable

Design Principle 8: Anticipate obsolescence

- Plan for changes in the technology or environment so the software will continue to run or can be easily changed
  - □ Avoid using early releases of technology
  - □ Avoid using software libraries that are specific to particular environments
  - □ Avoid using undocumented features or little-used features of software libraries
  - □ Avoid using software or special hardware from companies that are less likely to provide long-term support
  - □ Use standard languages and technologies that are supported by multiple vendors

Design Principle 9: Design for Portability

- Have the software run on as many platforms as possible
  - □ Avoid the use of facilities that are specific to one particular environment
  - □ E.g. a library only available in Microsoft Windows

Design Principle 10: Design for Testability

- Take steps to make testing easier
  - □ Design a program to automatically test the software
    - Discussed more in Chapter 10
    - Ensure that all the functionality of the code can by driven by an external program, bypassing a graphical user interface
  - □ In Java, you can create a main() method in each class in order to exercise the other methods

Design Principle 11: Design defensively

- Never trust how others will try to use a component you are designing
  - □ Handle all cases where other code might attempt to use your component inappropriately
  - □ Check that all of the inputs to your component are valid: the *preconditions*
    - Unfortunately, over-zealous defensive design can result in unnecessarily repetitive checking

Design Heuristics

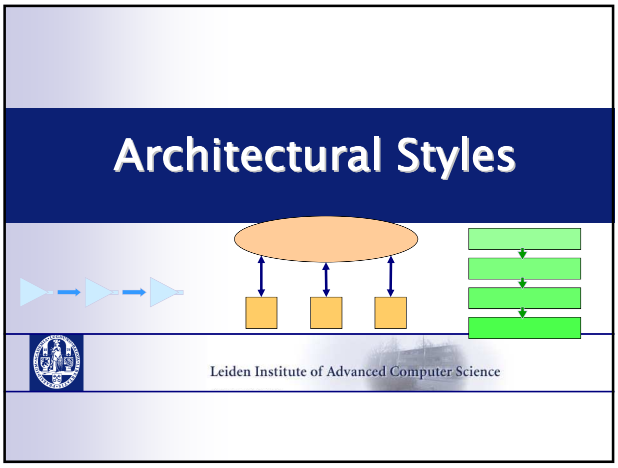
Design defensively:
Do not trust that others will use your component as specified – each component should ensure its own integrity

(from Lethbridge & Laganiere, p. 318)

A component should handle all cases where attempts are made to use it inappropriately:
- check whether all inputs are valid
- check preconditions

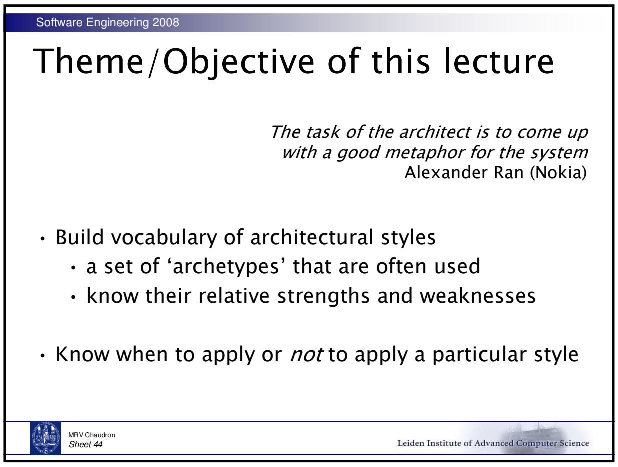
Using cost-benefit analysis to choose among alternatives

- To estimate the *costs*, add up:
  - □ The incremental cost of doing the *software engineering* work, including ongoing maintenance
  - □ The incremental costs of any *development technology* required
  - □ The incremental costs that *end-users and product support personnel* will experience
- To estimate the *benefits*, add up:
  - □ The incremental software engineering time saved
  - □ The incremental benefits measured in terms of either increased sales or else financial benefit to users



Architectural Styles

Leiden Institute of Advanced Computer Science



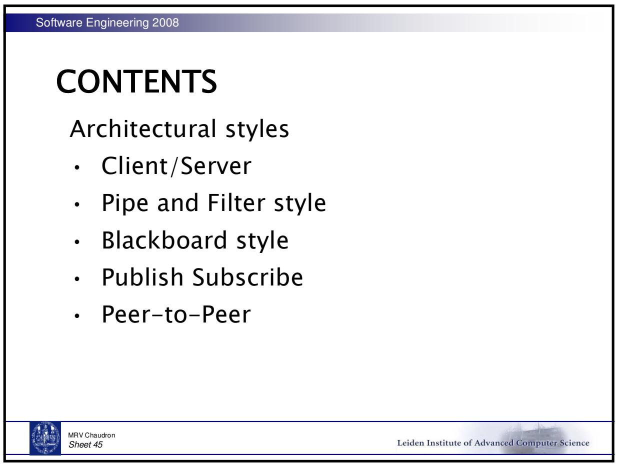
Software Engineering 2008

Theme/Objective of this lecture

The task of the architect is to come up with a good metaphor for the system
Alexander Ran (Nokia)

- Build vocabulary of architectural styles
  - a set of 'archetypes' that are often used
  - know their relative strengths and weaknesses
- Know when to apply or *not* to apply a particular style

MRV Chaudron
Sheet 44
Leiden Institute of Advanced Computer Science



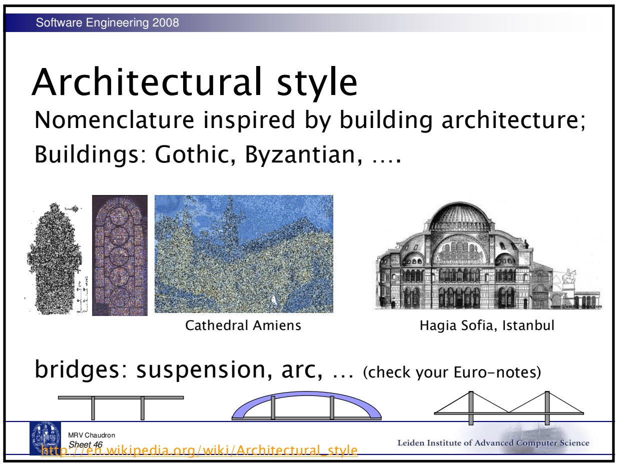
Software Engineering 2008

CONTENTS

Architectural styles

- Client/Server
- Pipe and Filter style
- Blackboard style
- Publish Subscribe
- Peer-to-Peer

MRV Chaudron
Sheet 45
Leiden Institute of Advanced Computer Science



Software Engineering 2008

Architectural style

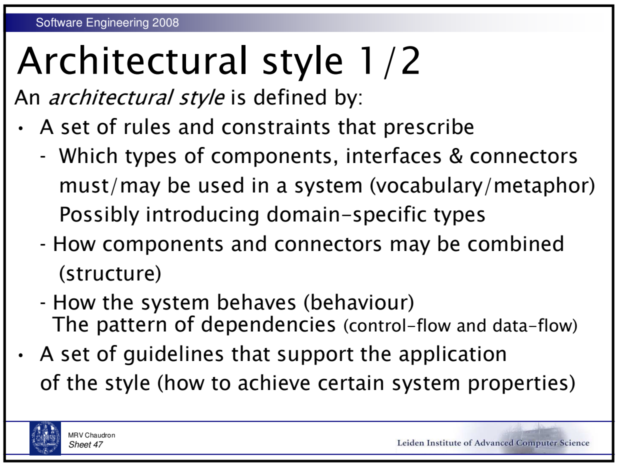
Nomenclature inspired by building architecture;
Buildings: Gothic, Byzantian, ….

Cathedral Amiens

Hagia Sofia, Istanbul

bridges: suspension, arc, … (check your Euro-notes)

MRV Chaudron
Sheet 46
http://en.wikipedia.org/wiki/Architectural_style
Leiden Institute of Advanced Computer Science



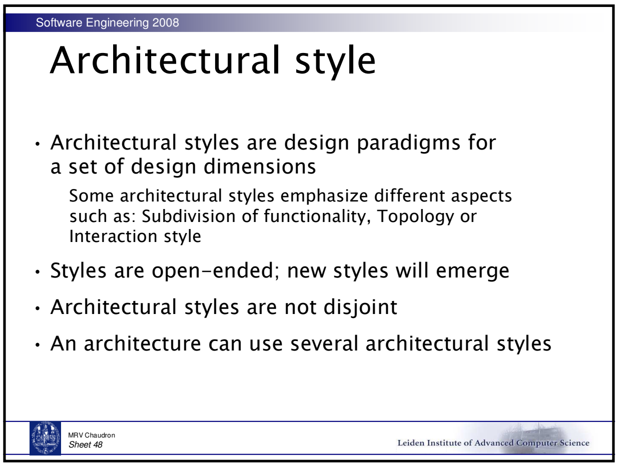
Software Engineering 2008

Architectural style 1/2

An *architectural style* is defined by:

- A set of rules and constraints that prescribe
  - Which types of components, interfaces & connectors must/may be used in a system (vocabulary/metaphor) Possibly introducing domain-specific types
  - How components and connectors may be combined (structure)
  - How the system behaves (behaviour) The pattern of dependencies (control-flow and data-flow)
- A set of guidelines that support the application of the style (how to achieve certain system properties)

MRV Chaudron
Sheet 47
Leiden Institute of Advanced Computer Science



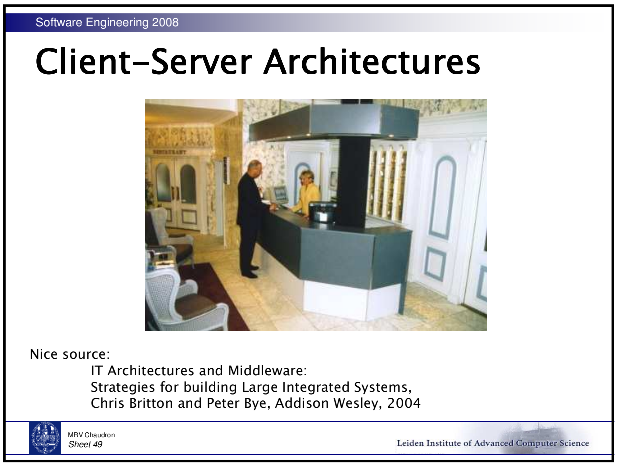
Software Engineering 2008

Architectural style

- Architectural styles are design paradigms for a set of design dimensions

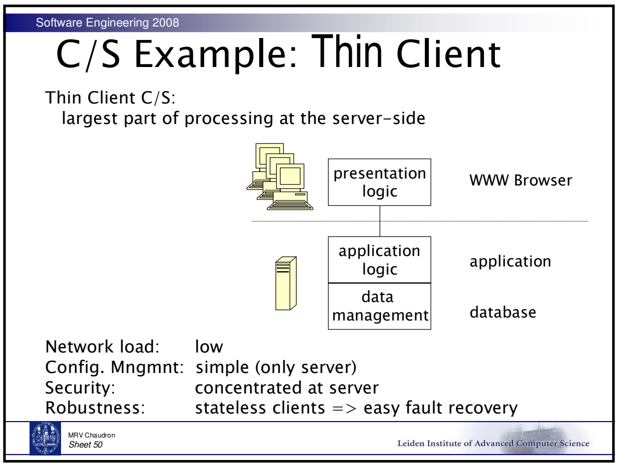
  Some architectural styles emphasize different aspects such as: Subdivision of functionality, Topology or Interaction style
- Styles are open-ended; new styles will emerge
- Architectural styles are not disjoint
- An architecture can use several architectural styles

MRV Chaudron
Sheet 48
Leiden Institute of Advanced Computer Science



Client-Server Architectures

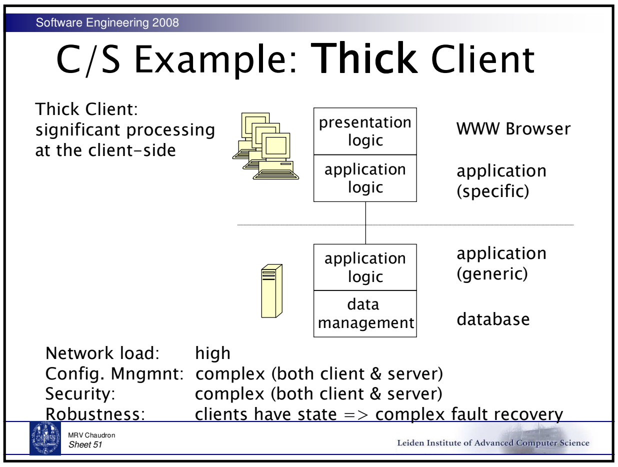
Nice source:
IT Architectures and Middleware:
Strategies for building Large Integrated Systems,
Chris Britton and Peter Bye, Addison Wesley, 2004



C/S Example: Thin Client

Thin Client C/S:
largest part of processing at the server-side

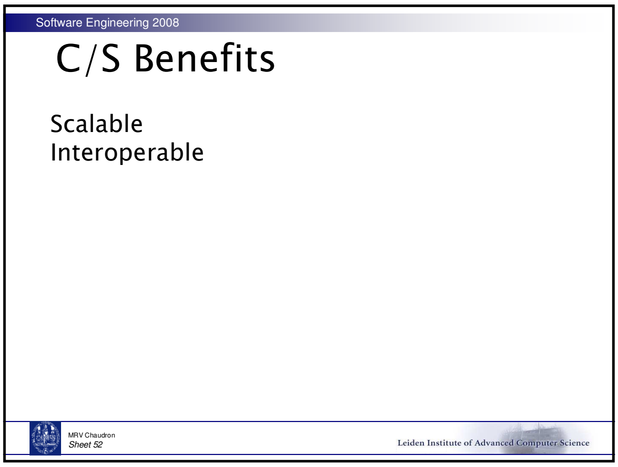
Network load: low
Config. Mngmnt: simple (only server)
Security: concentrated at server
Robustness: stateless clients => easy fault recovery



C/S Example: **Thick** Client

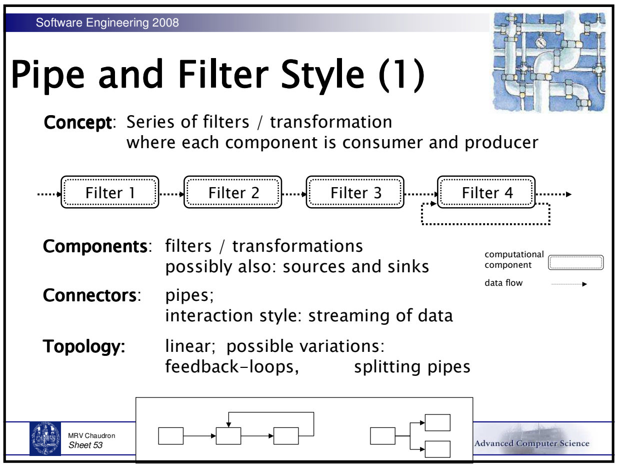
Thick Client:
significant processing
at the client-side

Network load: high
Config. Mngmnt: complex (both client & server)
Security: complex (both client & server)
Robustness: clients have state => complex fault recovery



C/S Benefits

Scalable
Interoperable



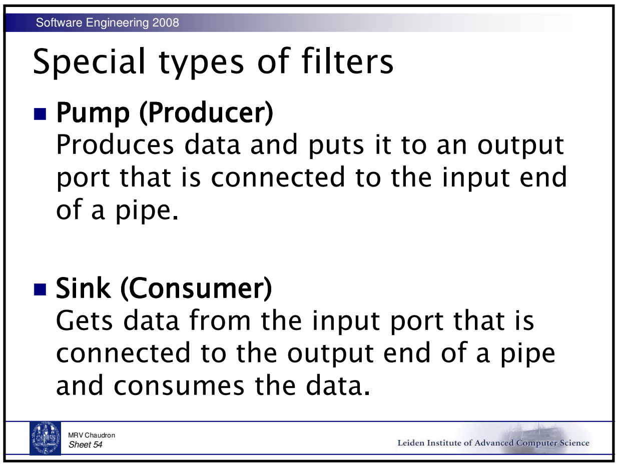
Pipe and Filter Style (1)

Concept: Series of filters / transformation
where each component is consumer and producer

Components: filters / transformations
possibly also: sources and sinks

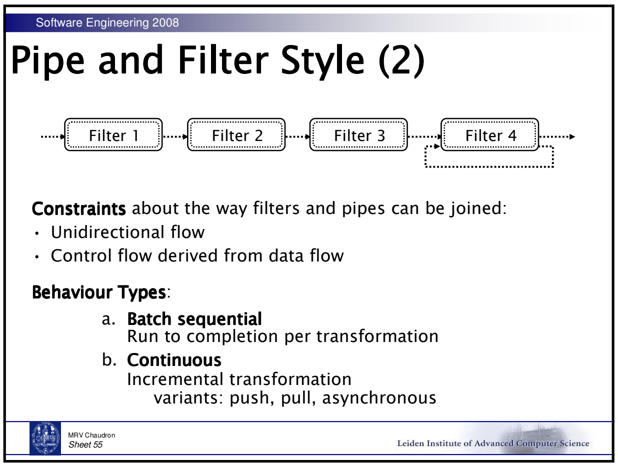
Connectors: pipes;
interaction style: streaming of data

Topology: linear; possible variations:
feedback-loops, splitting pipes



Special types of filters

- **Pump (Producer)**
  Produces data and puts it to an output port that is connected to the input end of a pipe.
- **Sink (Consumer)**
  Gets data from the input port that is connected to the output end of a pipe and consumes the data.



Software Engineering 2008

Pipe and Filter Style (2)

Filter 1
Filter 2
Filter 3
Filter 4

Constraints about the way filters and pipes can be joined:

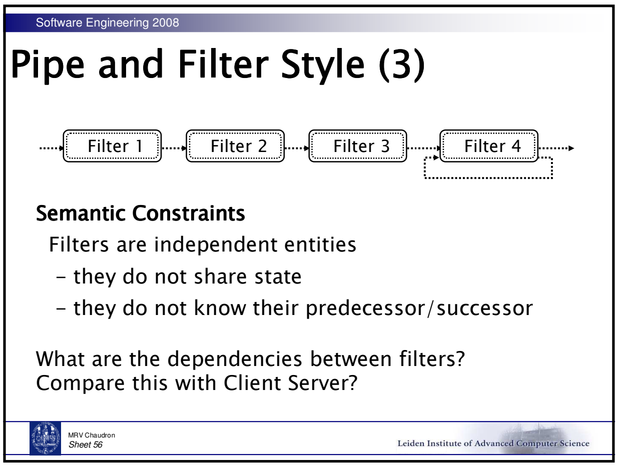
- Unidirectional flow
- Control flow derived from data flow

Behaviour Types:

a. **Batch sequential**
Run to completion per transformation

b. **Continuous**
Incremental transformation
variants: push, pull, asynchronous

MRV Chaudron
Sheet 55
Leiden Institute of Advanced Computer Science



Software Engineering 2008

Pipe and Filter Style (3)

Filter 1
Filter 2
Filter 3
Filter 4

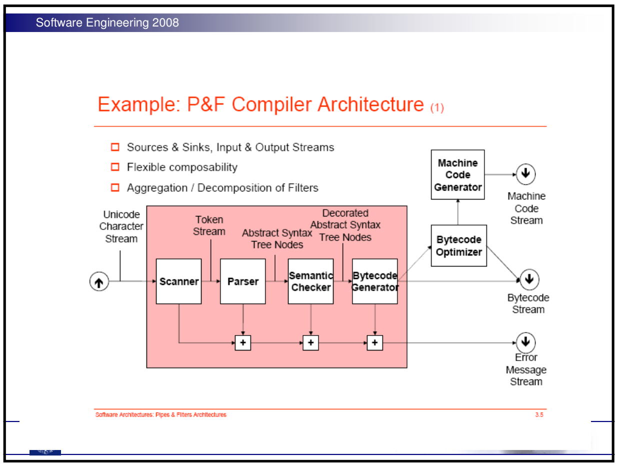
Semantic Constraints

Filters are independent entities

- they do not share state
- they do not know their predecessor/successor

What are the dependencies between filters?
Compare this with Client Server?

MRV Chaudron
Sheet 56
Leiden Institute of Advanced Computer Science

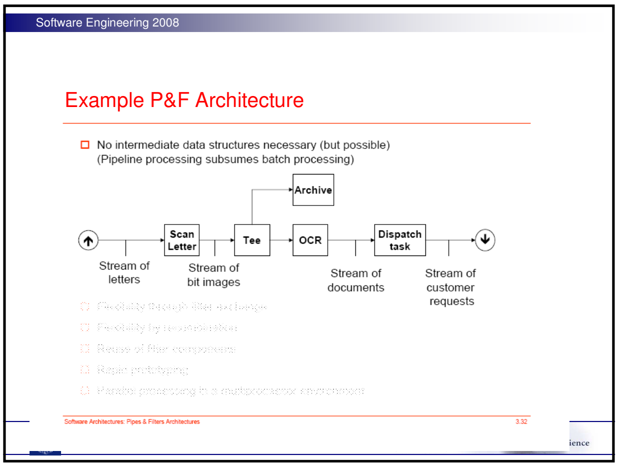


Software Engineering 2008

Example: P&F Compiler Architecture (1)

□ Sources & Sinks, Input & Output Streams
□ Flexible composability
□ Aggregation / Decomposition of Filters

Software Architectures: Pipes & Filters Architectures 3.5



Software Engineering 2008

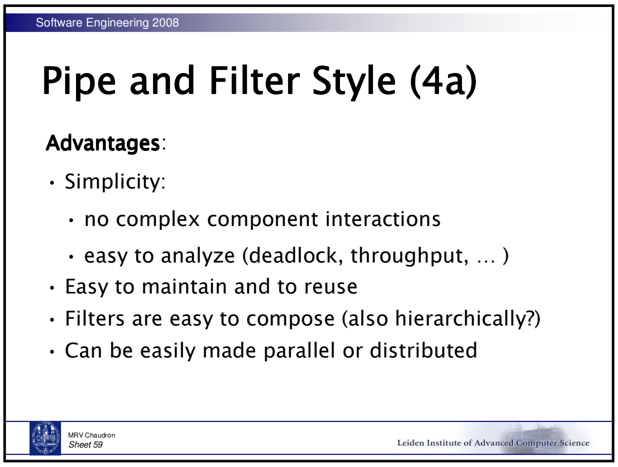
Example P&F Architecture

□ No intermediate data structures necessary (but possible)
(Pipeline processing subsumes batch processing)

□ Flexibility through filter exchange
□ Flexibility by recombination
□ Reuse of filter components
□ Rapid prototyping
□ Parallel processing in a multiprocessor environment

Software Architectures: Pipes & Filters Architectures 3.32

ience



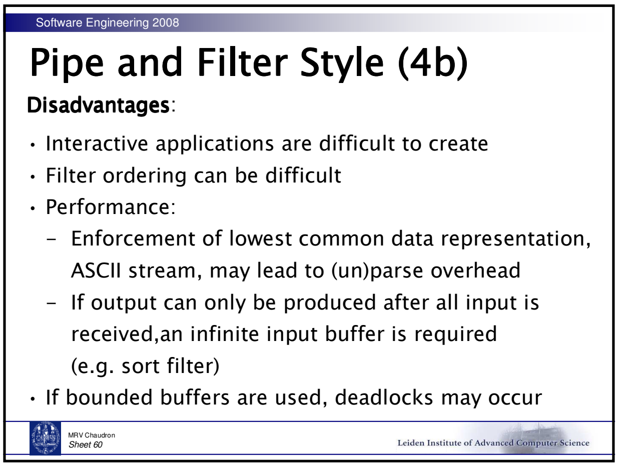
Software Engineering 2008

Pipe and Filter Style (4a)

Advantages:

- Simplicity:
  - no complex component interactions
  - easy to analyze (deadlock, throughput, …)
- Easy to maintain and to reuse
- Filters are easy to compose (also hierarchically?)
- Can be easily made parallel or distributed

MRV Chaudron
Sheet 59
Leiden Institute of Advanced Computer Science



Software Engineering 2008

Pipe and Filter Style (4b)

Disadvantages:

- Interactive applications are difficult to create
- Filter ordering can be difficult
- Performance:
  - Enforcement of lowest common data representation, ASCII stream, may lead to (un)parse overhead
  - If output can only be produced after all input is received,an infinite input buffer is required (e.g. sort filter)
- If bounded buffers are used, deadlocks may occur

MRV Chaudron
Sheet 60
Leiden Institute of Advanced Computer Science

Pipe and Filter Style (5) Quality Factors

Extendibility: extends easily with new filters

Flexibility: - functionality of filters can be easily redefined,
- control can be re-routed
(both at design-time, run-time is difficult)

Robustness: 'weakest link' is limitation

Security: -

Performance: allows straightforward parallelisation

Pipe and Filter Style (6) Application Context

Rules of thumb for choosing pipe-and-filter (o.a. from Shaw/Buschman):
- if a system can be described by a **regular interaction pattern** of a collection of processing units at the same level of abstraction; e.g. a series of incremental stages (horizontal composition of functionality);
- if the computation involves the **transformation of streams of data** (processes with limited fan-in/fan-out)

Hint: use a looped-pipe-and-filter if the system does continuous controlling of a physical system

Typical application domain: signal processing

Quality Improvement Methods and Empirical Research in SE

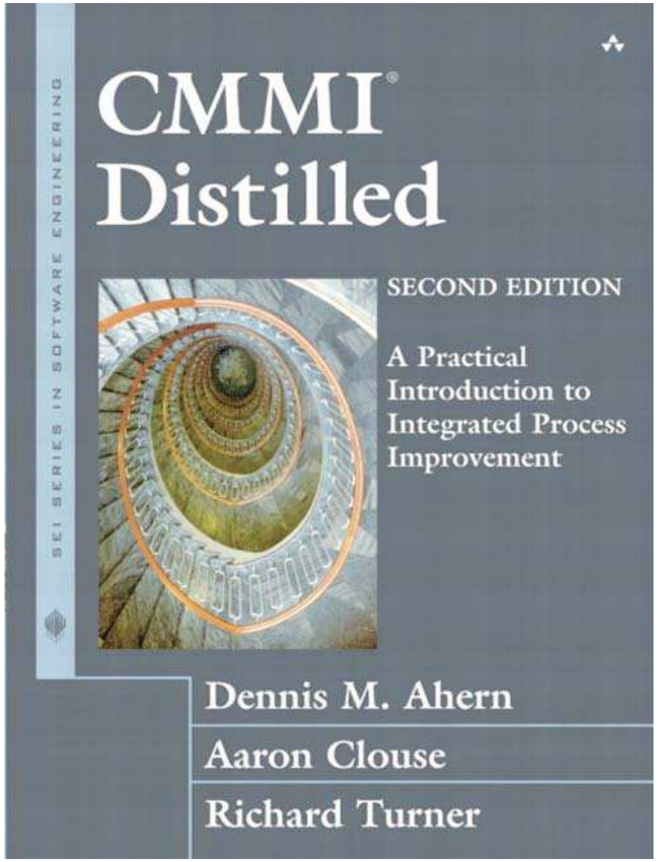
Michel Chaudron

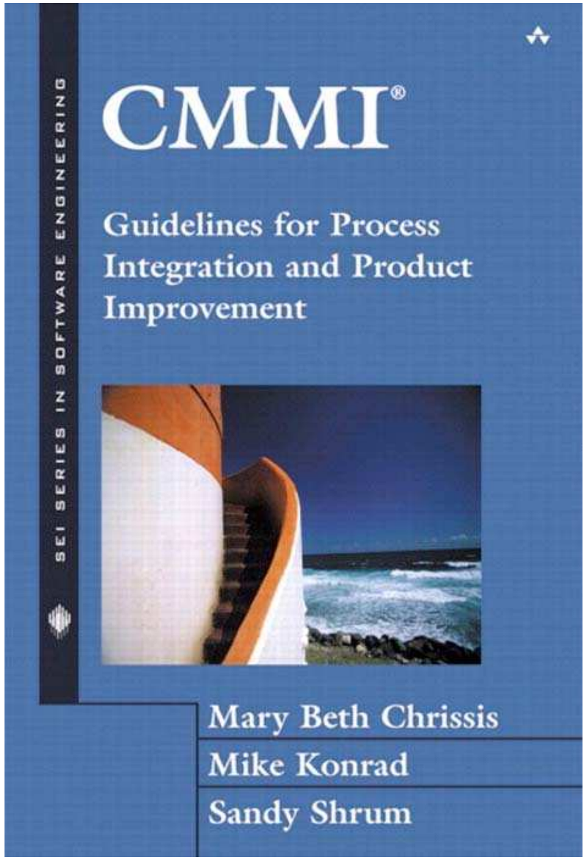
Leiden Institute of Advanced Computer Science

Agenda

- Quality Improvement
  - Software Process Improvement (CMMI)
  - Review and Inspection
  - Formal Methods
- Risk Management
- Empirical Research in Software Engineering
- Summary

CMM - Capability Maturity Model*

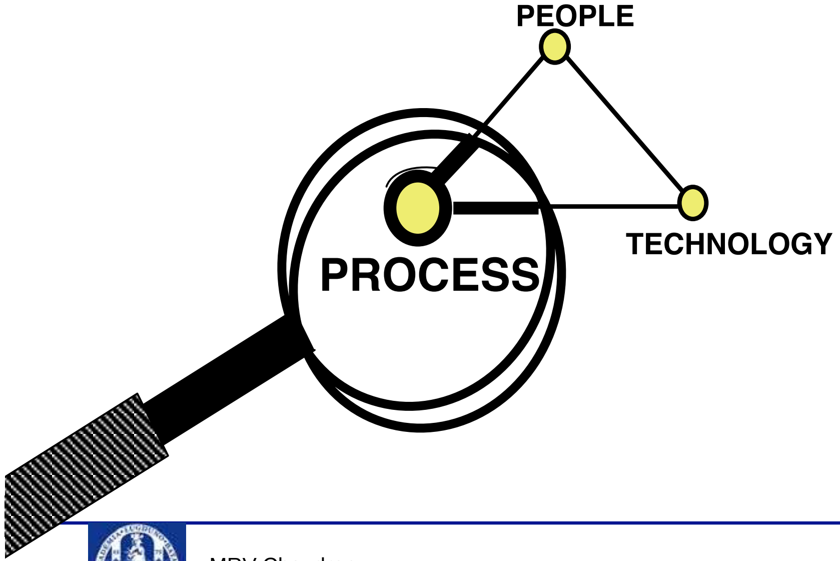

SEI SERIES IN SOFTWARE ENGINEERING
CMMI®
Distilled
SECOND EDITION
A Practical Introduction to Integrated Process Improvement
Dennis M. Ahern
Aaron Clouse
Richard Turner


SEI SERIES IN SOFTWARE ENGINEERING
CMMI®
Guidelines for Process Integration and Product Improvement
Mary Beth Chrissis
Mike Konrad
Sandy Shrum

* a.k.a. Consultant Money Making Initiative

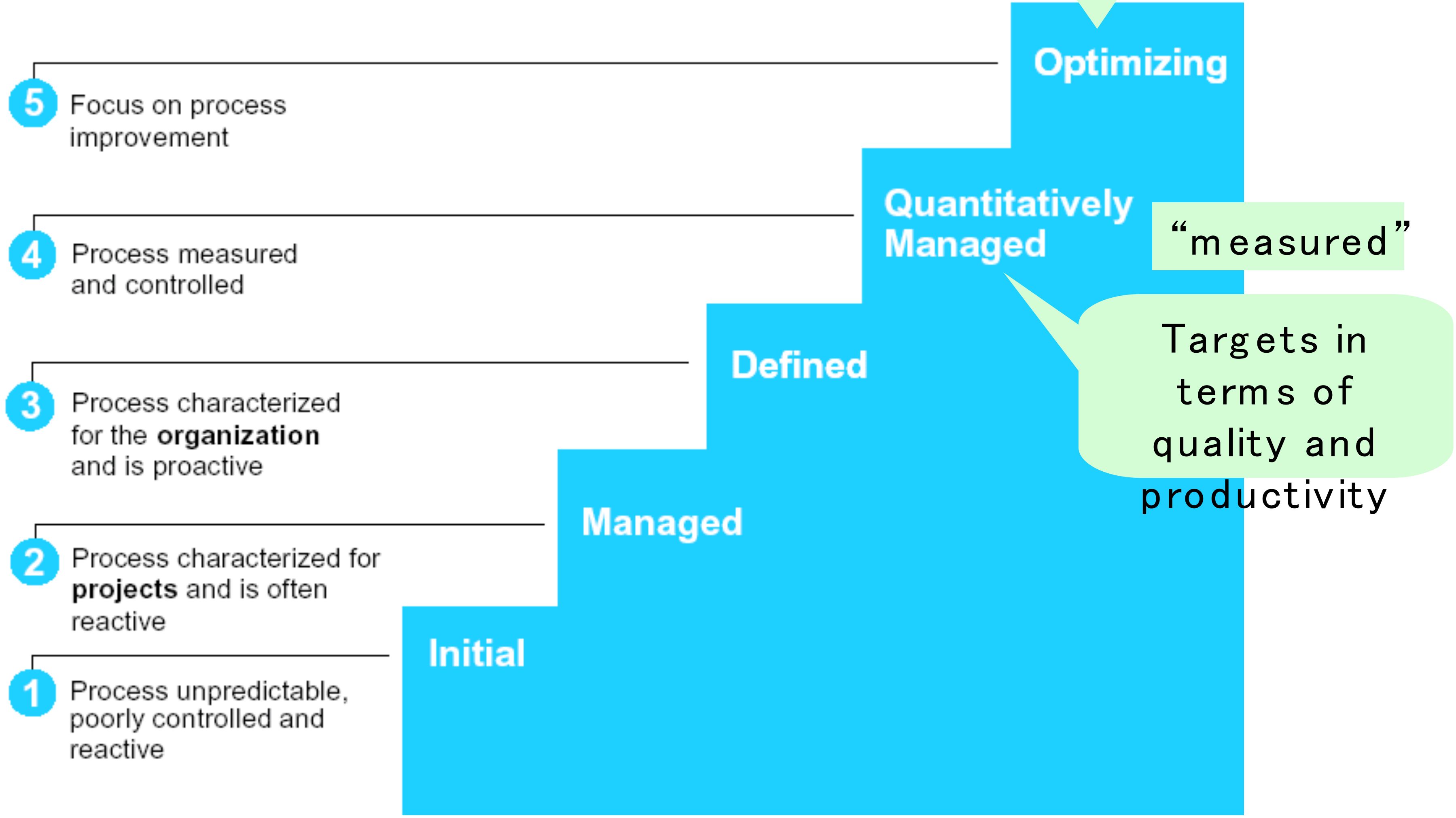
Premise of Software Process Improvement (SPI)

"The quality of a product is largely determined by the quality of the process that is used to develop and maintain it."



CMMI Maturity Levels

Identify opportunities for improvement



5 Focus on process improvement

4 Process measured and controlled

3 Process characterized for the **organization** and is proactive

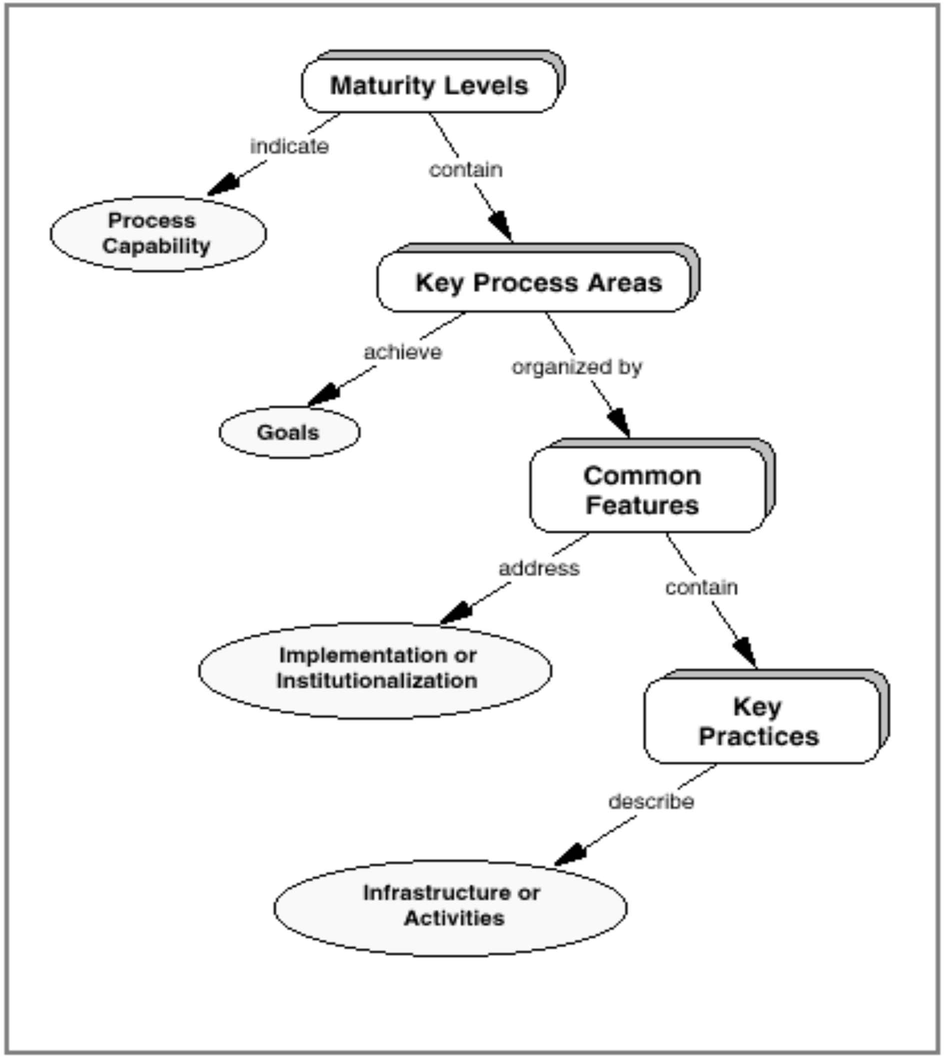
2 Process characterized for **projects** and is often reactive

1 Process unpredictable, poorly controlled and reactive

“measured”

Targets in terms of quality and productivity

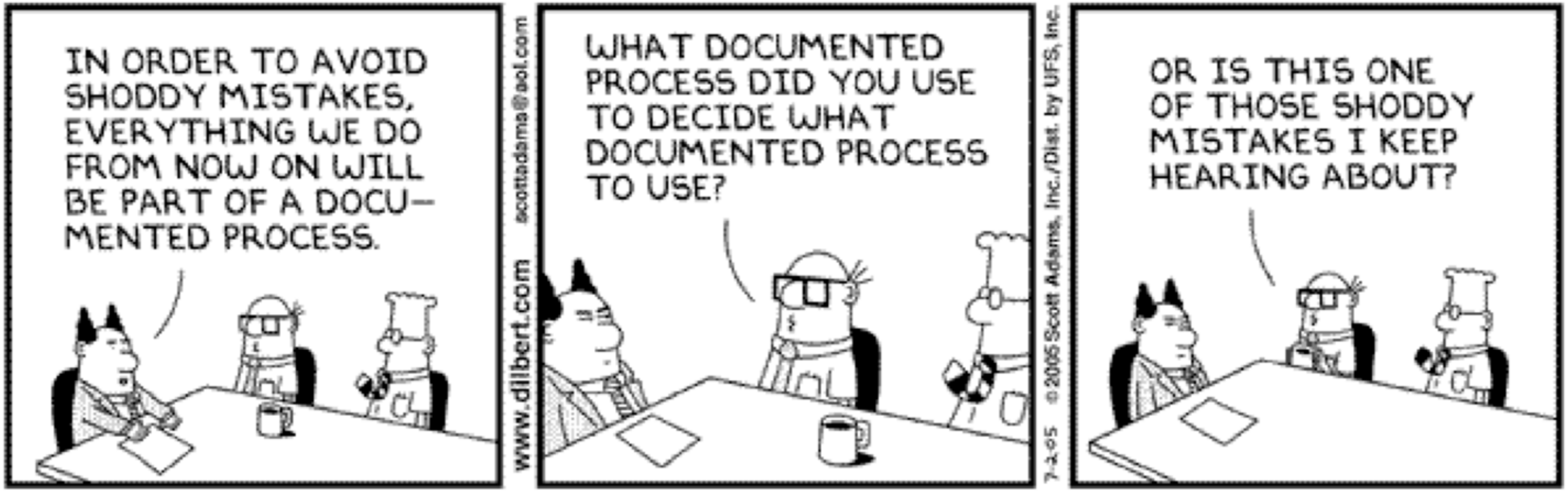
The CMM Structure



Process Areas by Maturity Level

Level	Focus	Process Areas
5 Optimizing	*Continuous process improvement*	Organizational Innovation and Deployment Causal Analysis and Resolution
4 Quantitatively Managed	*Quantitative management*	Organizational Process Performance Quantitative Project Management
3 Defined	*Process standardization* (SS) *(IPPD)* *(IPPD)*	Requirements Development Technical Solution Product Integration Verification Validation Organizational Process Focus Organizational Process Definition Organizational Training Integrated Project Management Integrated Supplier Management Risk Management Decision Analysis and Resolution Organizational Environment for Integration Integrated Teaming
2 Managed	*Basic project management*	Requirements Management Project Planning Project Monitoring and Control Supplier Agreement Management Measurement and Analysis Process and Product Quality Assurance Configuration Management
1 Initial		

Is the premise true?


IN ORDER TO AVOID SHODDY MISTAKES, EVERYTHING WE DO FROM NOW ON WILL BE PART OF A DOCUMENTED PROCESS.
www.dilbert.com scottadams@aol.com
WHAT DOCUMENTED PROCESS DID YOU USE TO DECIDE WHAT DOCUMENTED PROCESS TO USE?
7-2-05 © 2005 Scott Adams, Inc./Dist. by UFS, Inc.
OR IS THIS ONE OF THOSE SHODDY MISTAKES I KEEP HEARING ABOUT?

© Scott Adams, Inc./Dist. by UFS, Inc.

Process improvement should be done to help the business— not for its own sake.

Software Review and Inspection

Leiden Institute of Advanced Computer Science

Review

- A **Review** is a reading technique in which a software artifact is checked for defects by one or more persons other than the creator(s) of the document.
- Review can be applied to any type of document: code, design documents, test plans and requirements
- There are a number of types of review ranging in formality and effect.

Types of Review

- Buddy Checking
  - having a person other than the author **informally** review a piece of work.
  - generally does not involve the use of checklists to guide inspection and is therefore **not repeatable**.
  - generally does not require collection of data
  - **difficult to put under managerial control**

Types of Review

- Walkthrough
  - □ the author of an artifact **presents** his document or program to an audience of peers
  - □ The audience asks **questions** and makes comments on the artifact being presented in an attempt to identify defects
  - □ often break down into arguments about an issue
  - □ usually involve **no prior preparation** on behalf of the audience
  - □ usually involve minimal documentation of the process and of the issues found
  - □ process improvement and defect tracking are therefore not easy

Types of Review

- Review by Circulation
  - similar in concept to a walkthrough
  - artifact to be reviewed is **circulated** to a group of the author(s) peers for comment
  - avoids potential arguments over issues, however it also **avoids the benefits of discussion**
  - reviewer may be able to spend **longer reviewing** the artifact
  - there is documentation of the issues found, enabling defect tracking
  - usually minimal data collection

Types of Review

- Inspection (Fagan 76)
  - □ formally structured and managed peer review processes
  - □ involve a review team with clearly defined roles
  - □ **specific data is collected** during inspections
  - □ inspections have quantitative goals set
  - □ reviewers check an artifact against an unambiguous set of **inspection criteria** for that type of artifact
  - □ The required data collection promotes process improvement, and subsequent improvements in quality.

Software Inspection

- The inspection process comprises three broad stages:
  - □ preparation
  - □ collection
  - □ repair

- Gilb and Graham [GilbGraham93] expand this three stage process into the inspection steps; Entry, Planning, Kickoff Meeting, Individual Checking, Logging Meeting, Root Cause Analysis Edit, Follow Up, Exit.

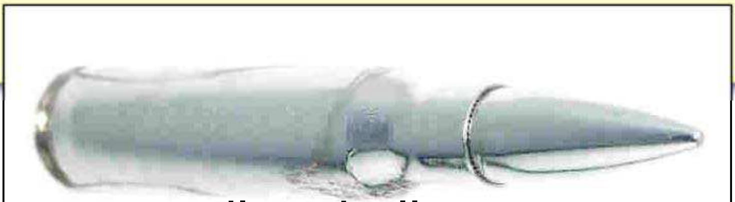
Principles of inspecting

- Choose an effective and efficient inspection team
  - between two and five people
  - Including **experienced** software engineers
- Require that participants **prepare** for inspections
  - They should **study** the documents **prior to the meeting** and come prepared with a list of defects
- Only inspect documents that are ready
  - Attempting to inspect a very poor document will result in defects being missed

Benefits of Inspection

believers edition

- 30% to 100% net productivity increases;
- Overall project time saving of 10% to 30%;
- 5 to 10 times reduction in test execution costs and time;
- Reduction in maintenance costs of up to one order of magnitude;
- Improvement in consequent product quality;
- Minimal defect correction backlash at systems integration time.
- In addition to these tangible benefits, less tangible benefits such as a training effect for inspectors are also evident.



Silver bullet?

Benefits of Inspection

Chaudron edition

- Helps creating common understanding and shared vision of the system
- Small investment in effort can have large benefits
  - □ Becomes better when staff is trained and checklists and reading guidelines are available
- Subjective
- Does not solve all problems
  - □ ➔ should be used in combination with other QA techniques

A peer-review

- Managers are normally not involved
  - This allows the participants to express their criticisms more openly, not fearing repercussions
  - The members of an inspection team should feel they are all working together to create a better document
  - Nobody should be blamed

Egoless-ness

- You are not your document/code
- Being open to improvement
- Seeing feedback as a learning opportunity

- Nobody is perfect

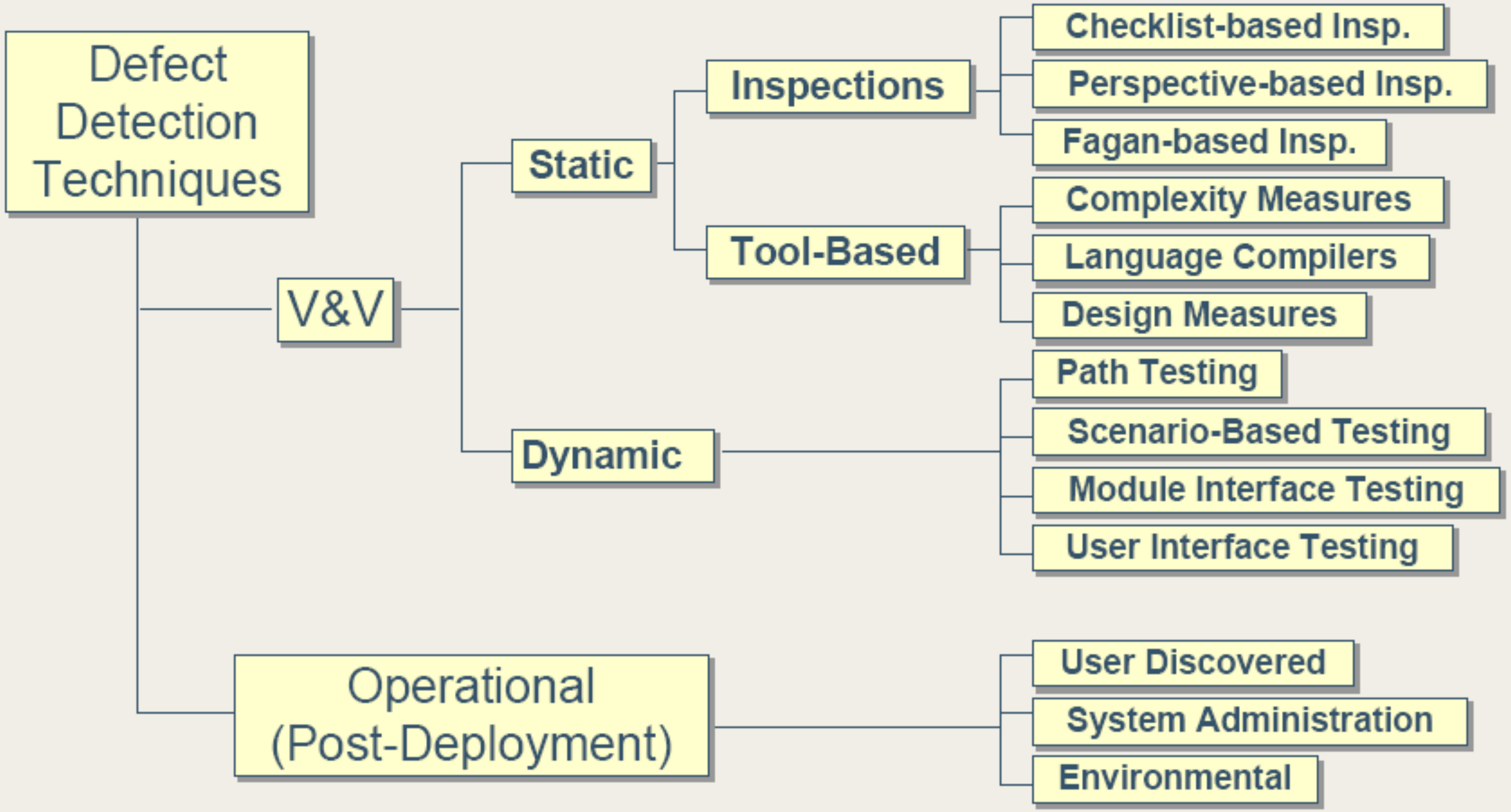
Quality Improvement Methods

- Structured processes
- Reviews and Inspections
- Metrics
- Testing
- Prototyping
- Mathematical proof of correctness / formal specification

Carnegie Mellon
Software Engineering Institute

Defect Discovery Sources

(how are the data generated)



Defect Detection Techniques

- V&V
  - Static
    - Inspections
      - Checklist-based Insp.
      - Perspective-based Insp.
      - Fagan-based Insp.
    - Tool-Based
      - Complexity Measures
      - Language Compilers
      - Design Measures
  - Dynamic
    - Path Testing
    - Scenario-Based Testing
    - Module Interface Testing
    - User Interface Testing
- Operational (Post-Deployment)
  - User Discovered
  - System Administration
  - Environmental

© 2001 by Carnegie Mellon University

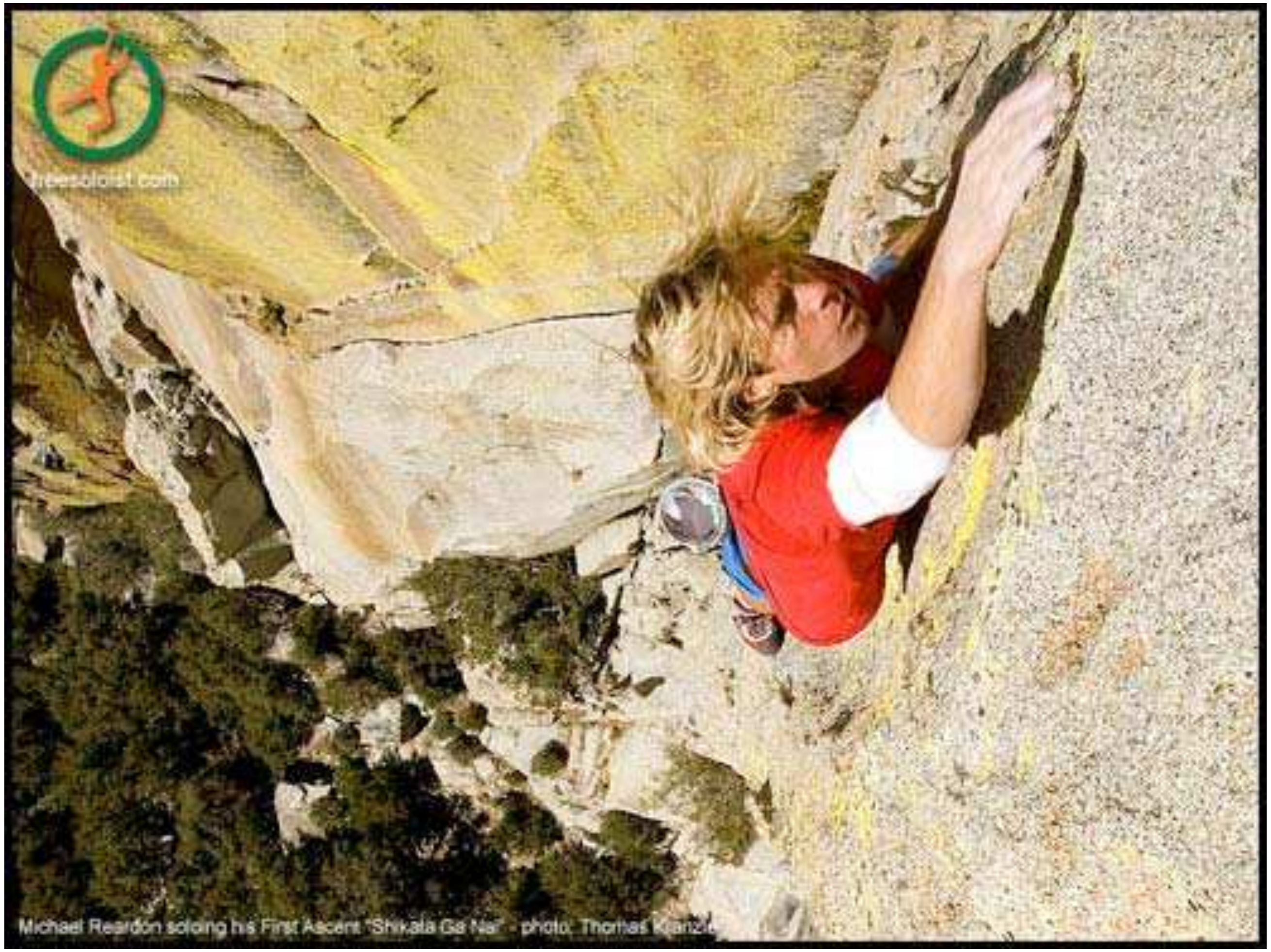
Risk Management

Leiden Institute of Advanced Computer Science

Risk Management

- Risk
  - Risk refers to uncertainty about the structure, outcomes or consequences of a decision or plan.
- Risk Management?
  - A Method for Dealing with Project Risks
    - Identification and Handling of Risks
  - On-Going Activity

Risk?

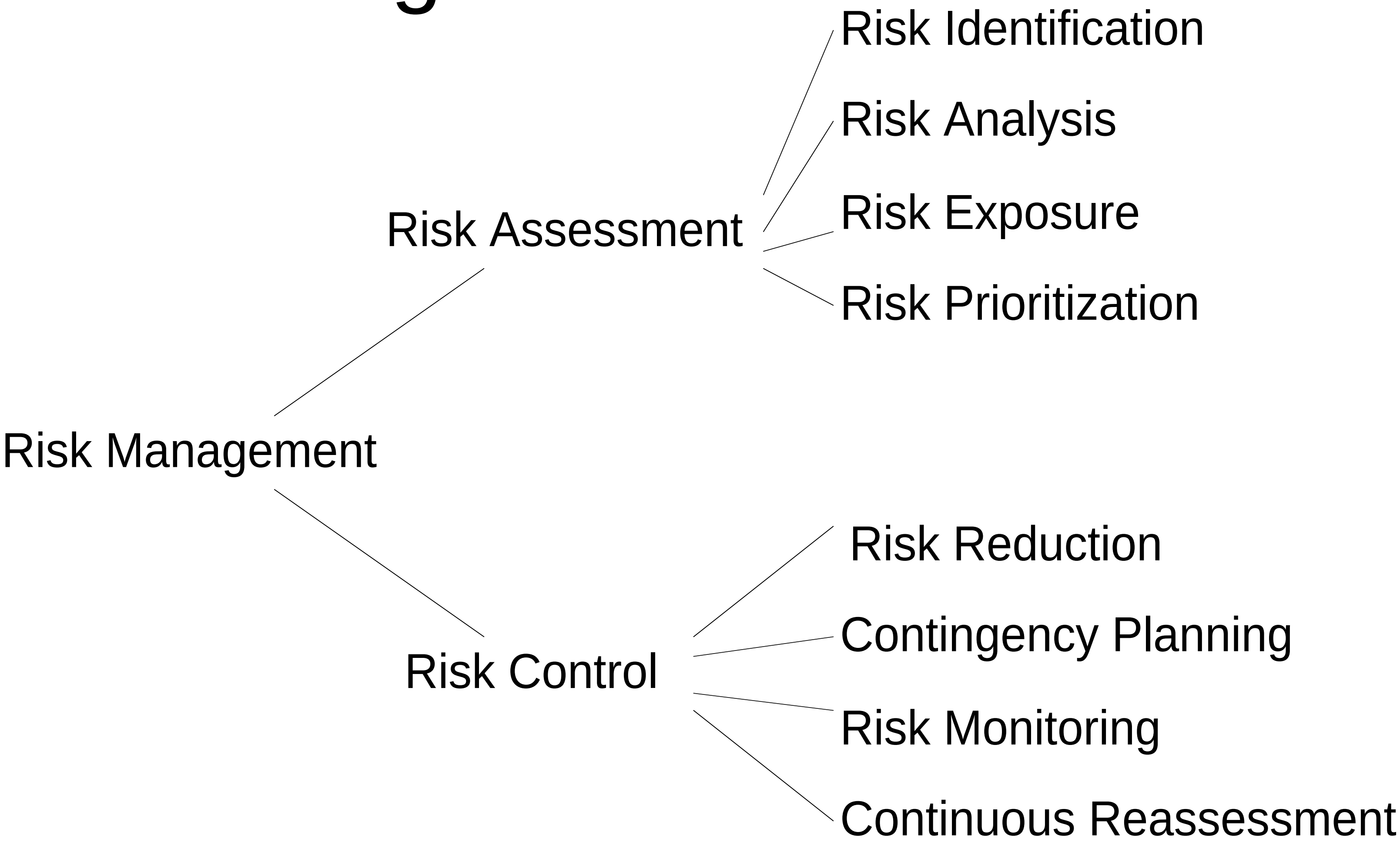


Michael Reardon soloing his First Ascent "Shikata Ga Nai" - photo: Thomas Kranzle

Risk Management: Basic Approach

- Analysis of Project
  - Identification of Risks
- For Each Risk:
  - Impact and Probability Analysis
    - What is the Nature of the Risk?
  - Avoidance/Mitigation Plans
    - How Can We Minimize the Risk?
  - Contingency Plans
    - What Do We Do if it Occurs?

Risk Management



Risk Management

Risk Assessment

Risk Identification

Risk Analysis

Risk Exposure

Risk Prioritization

Risk Control

Risk Reduction

Contingency Planning

Risk Monitoring

Continuous Reassessment

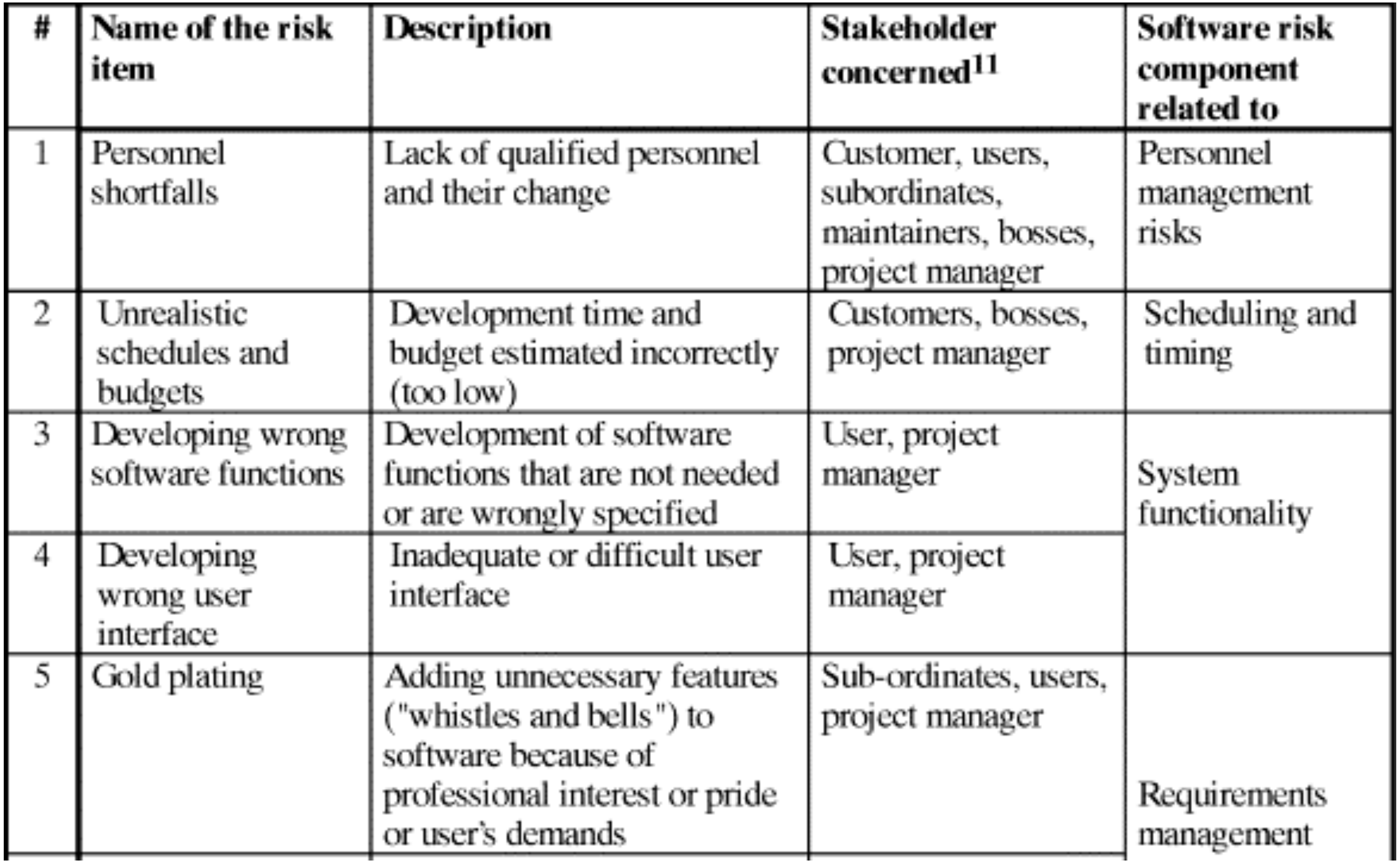
Risk Management: How to Identify Risks

- Start with a typical list of software risks
- Review development plan
  - Critical Paths
  - Critical Staff Members
  - Critical Vendor Deliveries
  - Critical Milestones
  - Training Requirements
- Review Requirements
- Review Technical Design
- Review Past Projects

Risk Management:
How to Identify Risks (Continued)

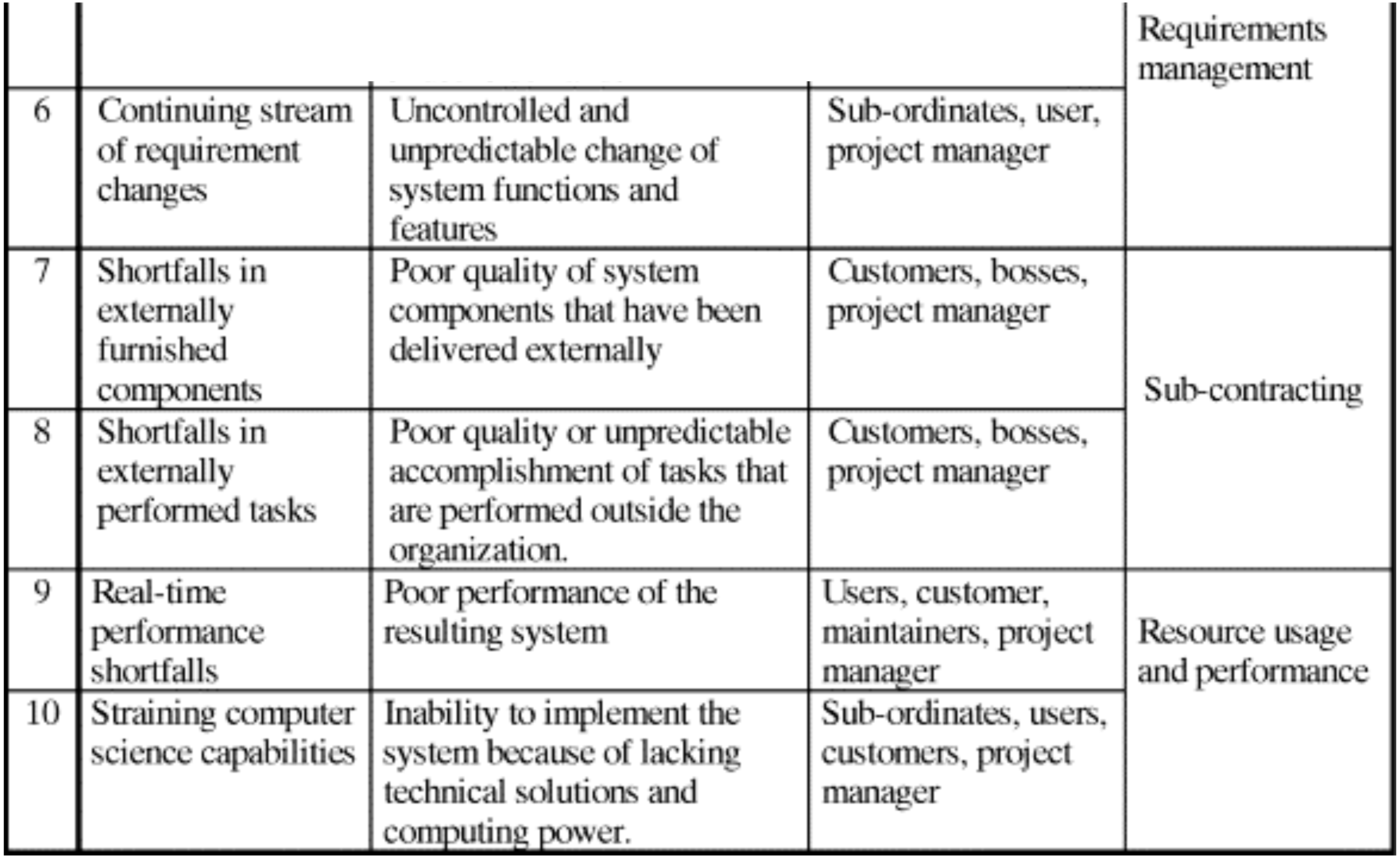
- Conduct Risk Brainstorming Sessions with Staff, Users, Vendors, Customers, and Management
  - □ Try to assess the direction of thinking by third parties as they may give an indication of future requirements, expectations, or vendor changes.
  - □ If you are dependent on vendors, try to understand their business situation.
- Get as much input as possible!

Common Risks in IT Development



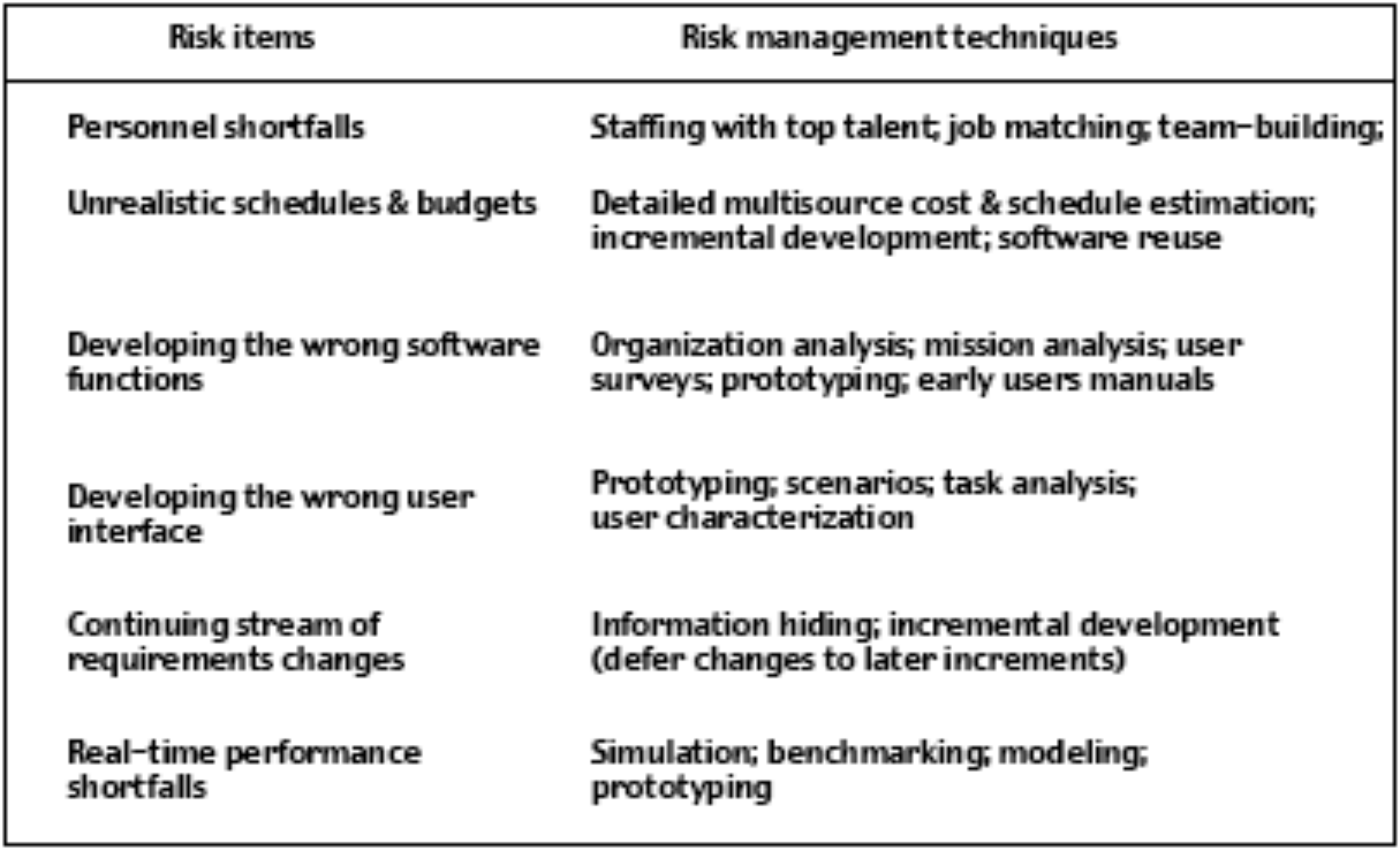
#	Name of the risk item	Description	Stakeholder concerned[11]	Software risk component related to
1	Personnel shortfalls	Lack of qualified personnel and their change	Customer, users, subordinates, maintainers, bosses, project manager	Personnel management risks
2	Unrealistic schedules and budgets	Development time and budget estimated incorrectly (too low)	Customers, bosses, project manager	Scheduling and timing
3	Developing wrong software functions	Development of software functions that are not needed or are wrongly specified	User, project manager	System functionality
4	Developing wrong user interface	Inadequate or difficult user interface	User, project manager	
5	Gold plating	Adding unnecessary features ("whistles and bells") to software because of professional interest or pride or user's demands	Sub-ordinates, users, project manager	Requirements management

Common Risks in IT Development



				Requirements management
6	Continuing stream of requirement changes	Uncontrolled and unpredictable change of system functions and features	Sub-ordinates, user, project manager	
7	Shortfalls in externally furnished components	Poor quality of system components that have been delivered externally	Customers, bosses, project manager	Sub-contracting
8	Shortfalls in externally performed tasks	Poor quality or unpredictable accomplishment of tasks that are performed outside the organization.	Customers, bosses, project manager	
9	Real-time performance shortfalls	Poor performance of the resulting system	Users, customer, maintainers, project manager	Resource usage and performance
10	Straining computer science capabilities	Inability to implement the system because of lacking technical solutions and computing power.	Sub-ordinates, users, customers, project manager	

Software Risk Management Techniques



Risk items	Risk management techniques
Personnel shortfalls	Staffing with top talent; job matching; team-building;
Unrealistic schedules & budgets	Detailed multisource cost & schedule estimation; incremental development; software reuse
Developing the wrong software functions	Organization analysis; mission analysis; user surveys; prototyping; early users manuals
Developing the wrong user interface	Prototyping; scenarios; task analysis; user characterization
Continuing stream of requirements changes	Information hiding; incremental development (defer changes to later increments)
Real-time performance shortfalls	Simulation; benchmarking; modeling; prototyping

Risk analysis

- estimating size of loss
  - how long it takes to “fix” the risk
- estimating probability of loss
  - most experienced estimates risks
  - delphi method vs. group consensus
  - betting on topic
  - adjective calibration
- risk exposure
  - probability of unexpected loss multiplied by the size of loss

Analysis, Exposure, & Prioritization

- For Each Risk:
  - Determine Probability of Occurrence
    - What is the likelyhood of occurrence?
  - Determine Impact
    - What is the impact if it occurres?
  - Determine Exposure
    - What will we lose if the risk occurs?

- For All Risks:
  - Prioritize
    - Where should we put our limited resources?

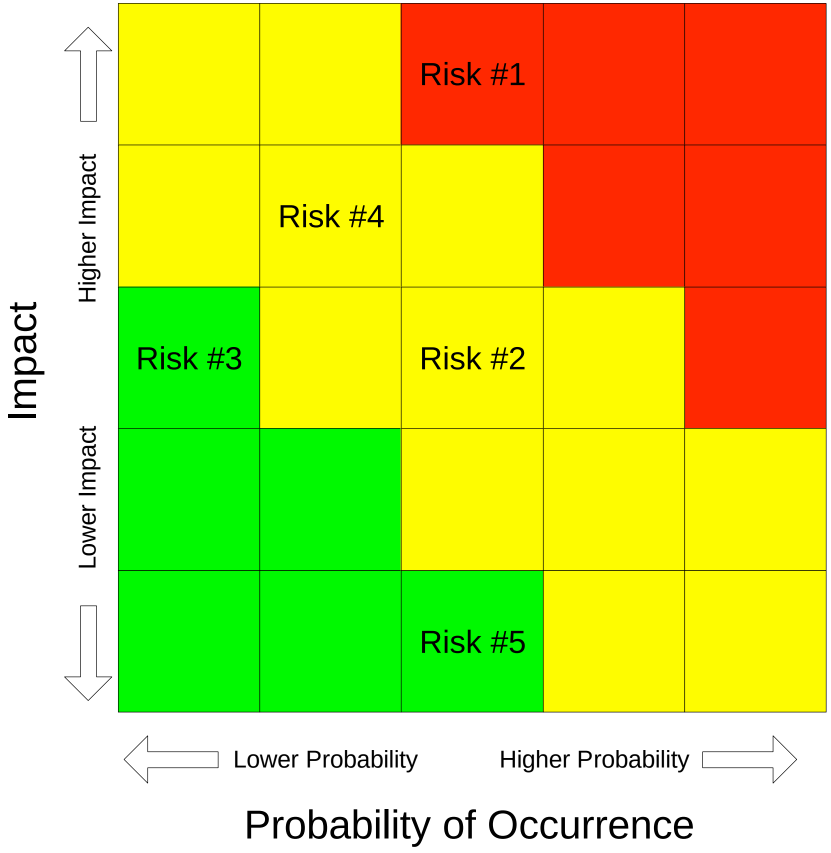
Analysis, Exposure, Prioritization: How?

- Various Techniques Available But Key is Experience
  - □ Individual
  - □ Organizational

- Don’t Rely on Just Yourself - Get lots of Inputs

Risk Assessment: A Simple Classification & Tracking Method

- Probability of Occurrence vs Impact
  - □ 1 to 5 Scale
- Priorities
  - □ Red - High
  - □ Yellow - Med
  - □ Green - Low
- Review/Present Chart Periodically

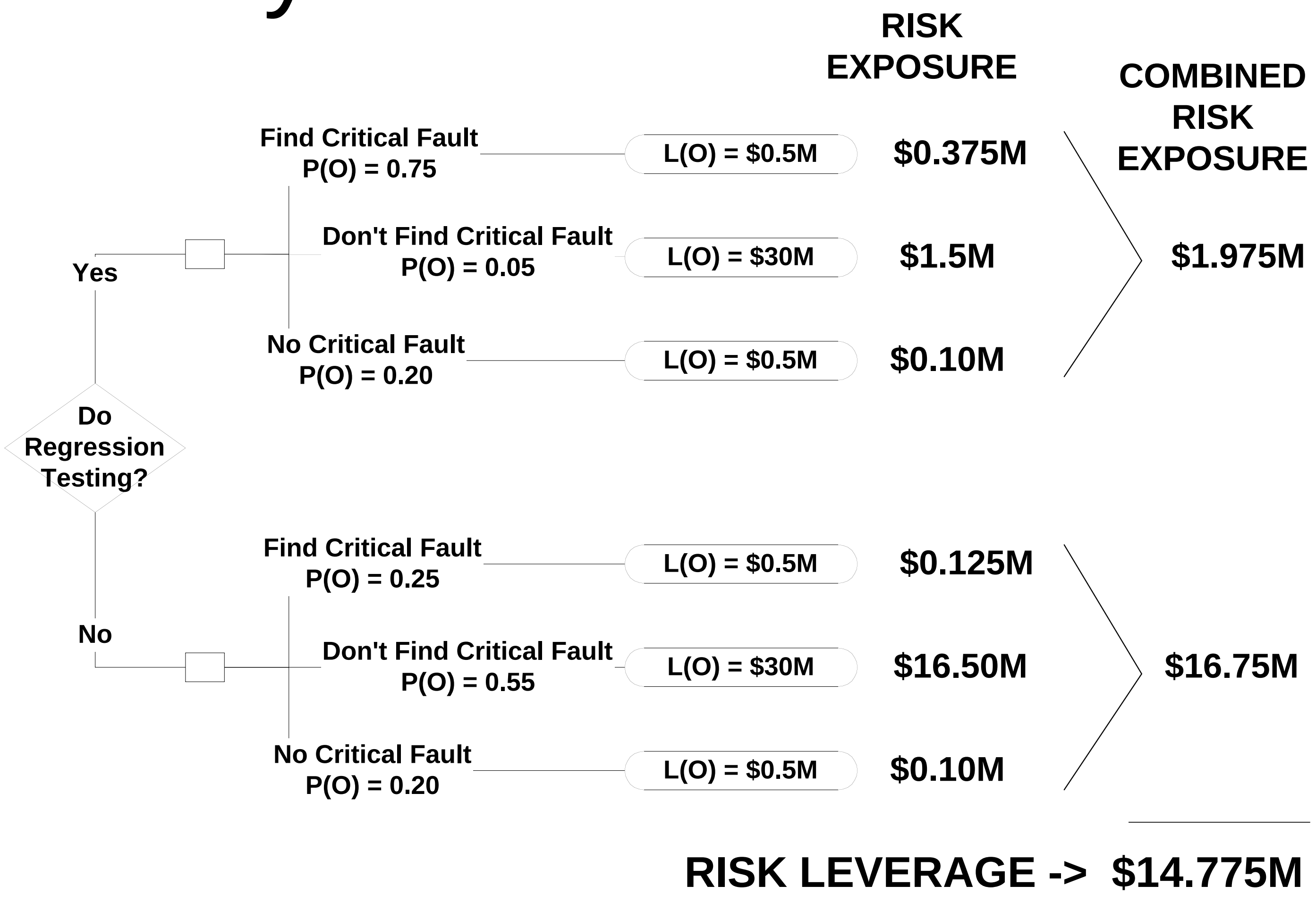


🟨	🟨	🟥 Risk #1	🟥	🟥
🟨	🟨 Risk #4	🟨	🟥	🟥
🟩 Risk #3	🟨	🟨 Risk #2	🟨	🟥
🟩	🟩	🟨	🟨	🟨
🟩	🟩	🟩 Risk #5	🟨	🟨

Risk Assessment: Probability Methods

- Can we quantitize the risk?
- For Each Risk:
  - For Each Possible Action:
    - Estimate Probability of an Given Outcome P(O)
    - Estimate $ Loss of an Given Outcome L(O)
    - Multiply the P(O) by L(O) to give $ exposure for the unwanted outcome
  - Sum all $ exposures for each Possible Action
  - Compare the $ exposures
  - Calculate Risk Leverage
    - (Risk Exposure Before Reduction - Risk Exposure After Reduction) / (Cost of Risk Reduction)

Example Risk Assessment Using Probability Method


RISK EXPOSURE
COMBINED RISK EXPOSURE
Find Critical Fault P(O) = 0.75
L(O) = $0.5M
$0.375M
Don't Find Critical Fault P(O) = 0.05
L(O) = $30M
$1.5M
$1.975M
Yes
No Critical Fault P(O) = 0.20
L(O) = $0.5M
$0.10M
Do Regression Testing?
Find Critical Fault P(O) = 0.25
L(O) = $0.5M
$0.125M
No
Don't Find Critical Fault P(O) = 0.55
L(O) = $30M
$16.50M
$16.75M
No Critical Fault P(O) = 0.20
L(O) = $0.5M
$0.10M
RISK LEVERAGE -> $14.775M

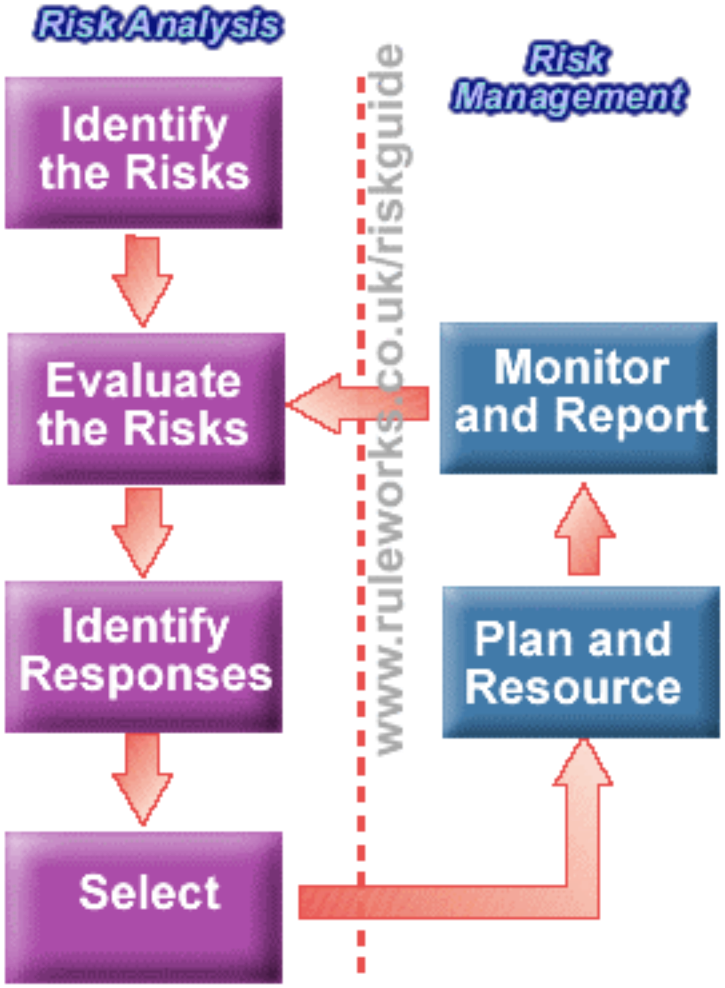
Risk Reduction

- Avoiding Risk
  - Modifying project requirements
- Transferring the Risk
  - By allocation to other systems
  - Buying Insurance to cover financial loses
- Mitigating the Risk
  - Pre-Event Actions to:
    - Reduce Likelihood of Occurrence and/or
    - Minimize Impact, Fail-over, Repair, …
- Some risks cannot be reduced
  - Contingency Plan - how will you deal with the risk

Monitoring Risk

- Periodic Review of Risk Status
  - Changes in Probabilities or Impacts
  - Changes in Avoidance/Mitigation/Contingency Plans
- Periodic Review of Project to Identify New Risks
- Implementation of Risk Avoidance or Mitigation Plans
- Keep Management and Customers Informed!!!
  - Frequent Risk Reviews

Risk Management Process



From: http://www.ruleworks.co.uk/riskguide/manage-risk-nl.htm

Empirical Research in Software Engineering

Leiden Institute of Advanced Computer Science

Empirical Research

Empirical research

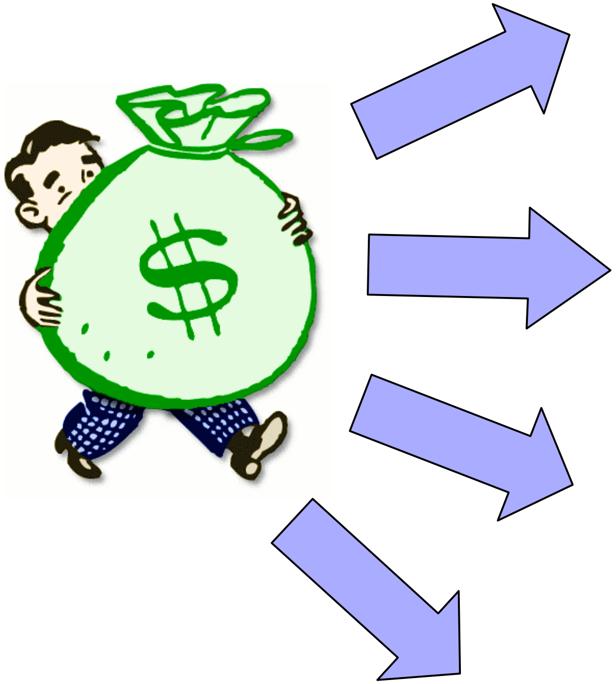
is research that bases its findings on direct or indirect observation as its test of reality.

Astronomy

Chemistry

Physics
Newton's apple

How to best allocate budget?



people

- Experienced .. Novice

technology

- Java vs .Net
- Code generation
- Automated testing, …

process

- In-house or offshore
- …

product

- Which features
- What level of quality?

We must understand the effect of our choices on productivity, quality, …

Examples

- The use of Object Oriented modeling and programming improves quality and productivity

  - □ True ?
  - □ Not True?
  - □ Don’t know

The Bottom Line

“In God we trust,
all others bring data.”
- W. Edwards Deming

What is ‘evidence’?

Use of RUP

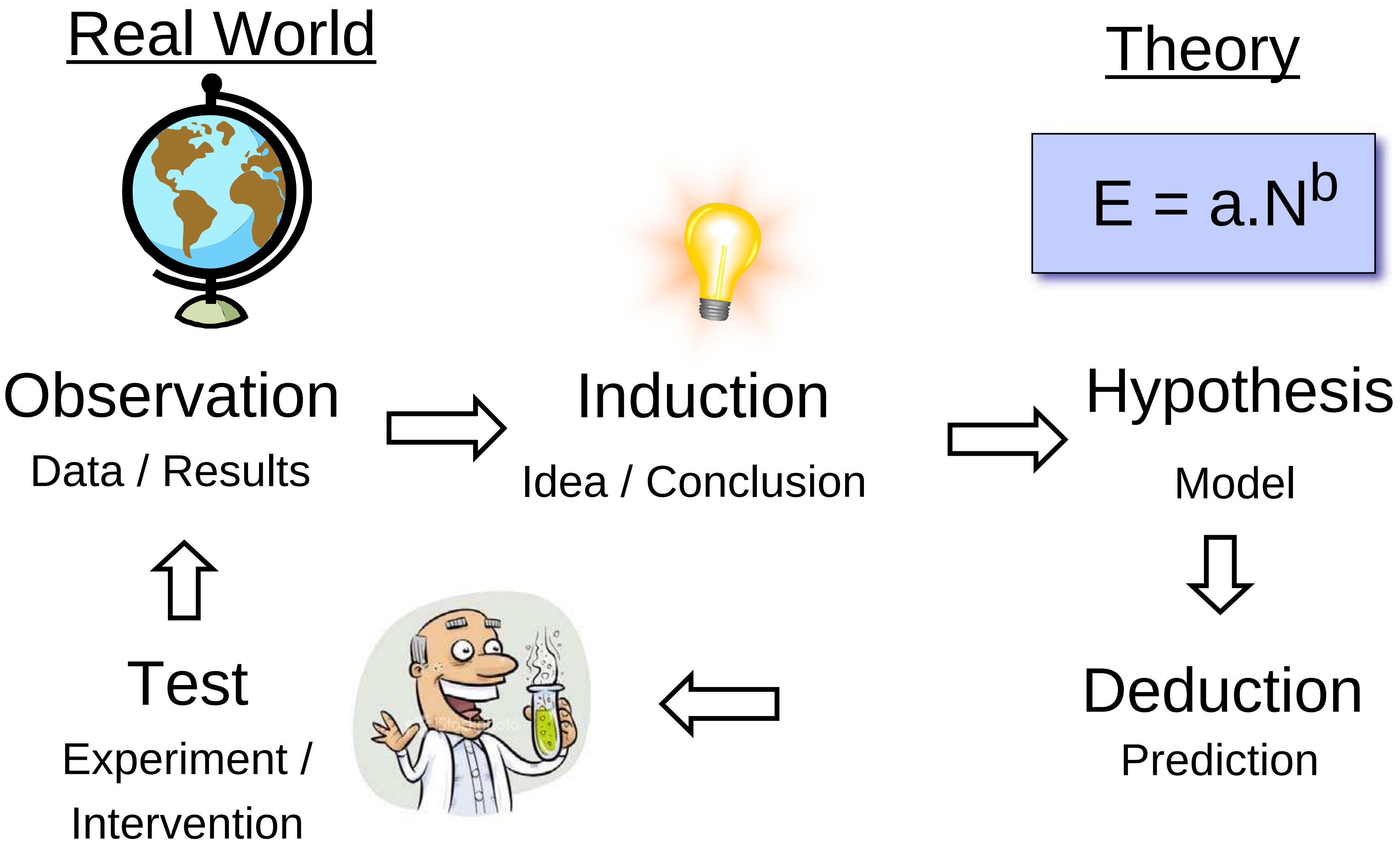
Use of RUP leads to improvement of productivity and quality

Approaches:

- Measure
- Expert opinions (interviews)
- Simulation

Combination of the above (triangulation)

Empirical Cycle



Real World

Theory

$E = a.N^{b}$

Observation

Data / Results

Induction

Idea / Conclusion

Hypothesis

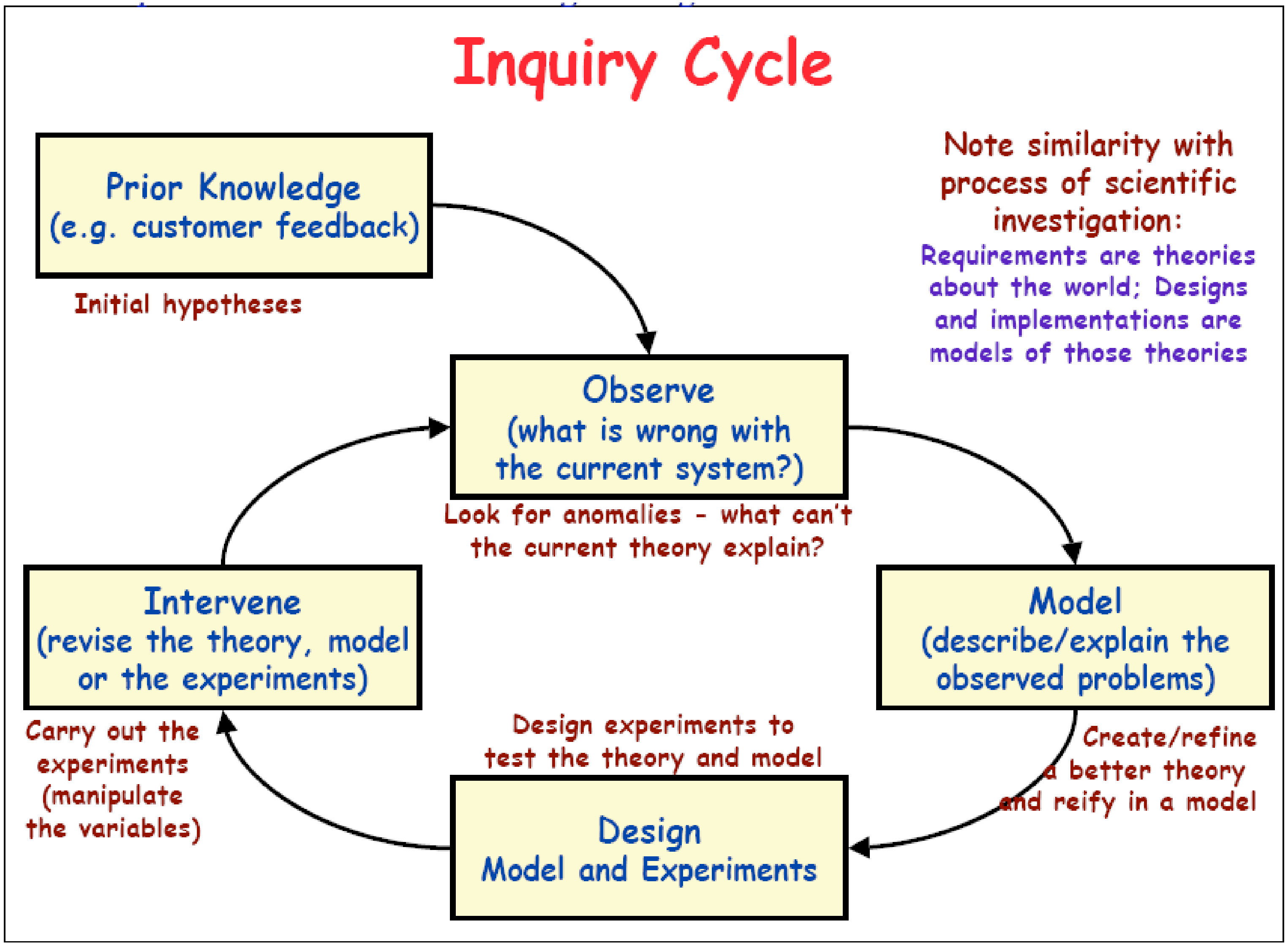
Model

Deduction

Prediction

Test

Experiment / Intervention



Inquiry Cycle

Prior Knowledge (e.g. customer feedback)

Initial hypotheses

Observe (what is wrong with the current system?)

Look for anomalies - what can't the current theory explain?

Model (describe/explain the observed problems)

Create/refine a better theory and reify in a model

Design Model and Experiments

Design experiments to test the theory and model

Intervene (revise the theory, model or the experiments)

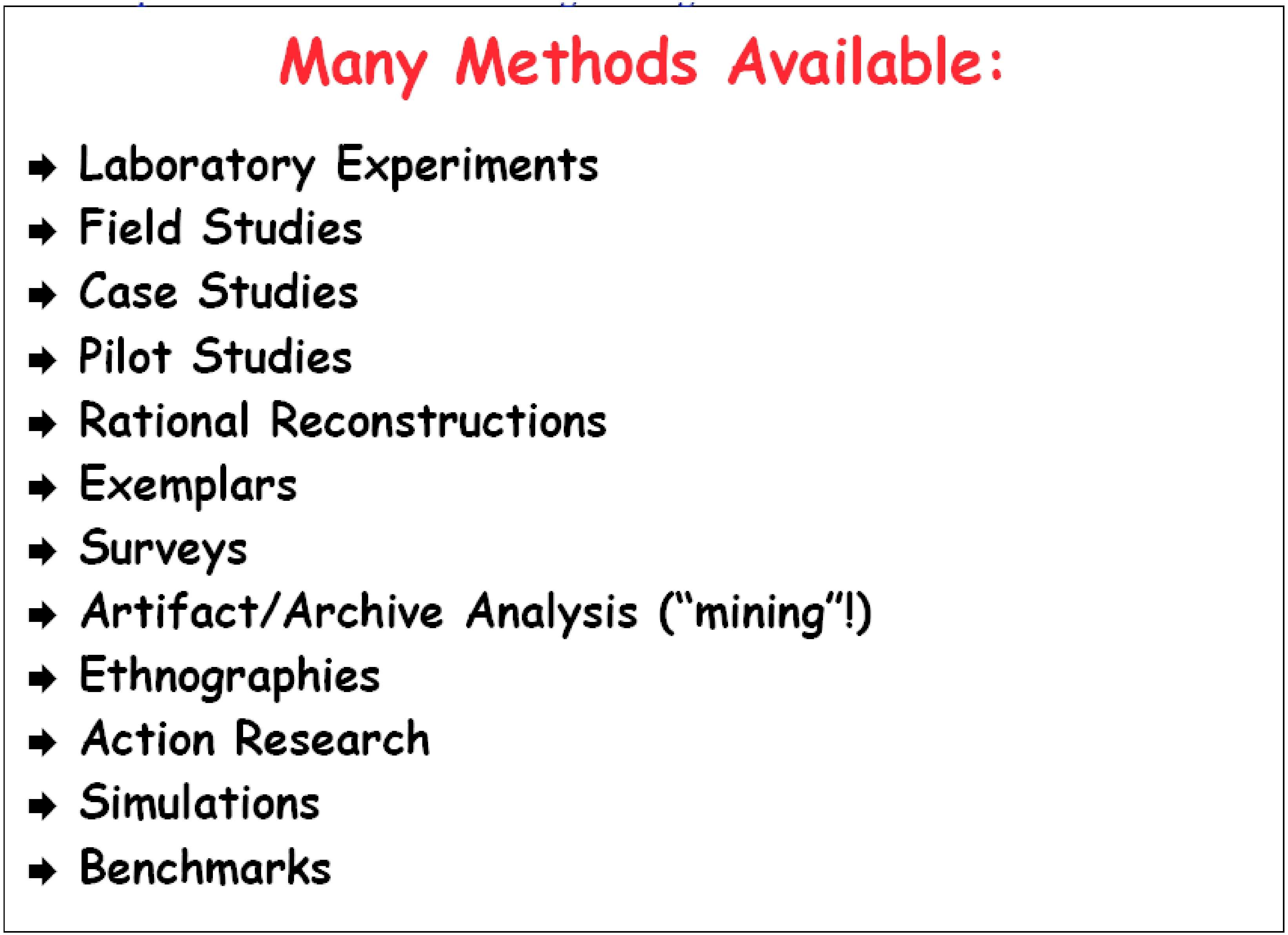
Carry out the experiments (manipulate the variables)

Note similarity with process of scientific investigation:

Requirements are theories about the world; Designs and implementations are models of those theories

Important characteristics of scientific research:

- rigor
- testability / falsifiability
- reproducibility
- precision
- objectivity
- parsimony
- generalisability (if possible)



Many Methods Available:

- ➡ Laboratory Experiments
- ➡ Field Studies
- ➡ Case Studies
- ➡ Pilot Studies
- ➡ Rational Reconstructions
- ➡ Exemplars
- ➡ Surveys
- ➡ Artifact/Archive Analysis ("mining"!)
- ➡ Ethnographies
- ➡ Action Research
- ➡ Simulations
- ➡ Benchmarks

Study - Examples

- Survey
  - □ After a new development process has been introduced: developers answer a questionnaire about their confidence in the new process.
- Experiment
  - □ Source code inspections: one group of participants uses inspection technique A, the other group uses inspection technique B. Compare the number of detected defects.
- Case study
  - □ Run a pilot project using a new tool (e.g. UML case tool) and compare productivity to company baseline

Experiment

- **When appropriate***:* control on who is using which technology, when, where and under which conditions. Investigation of self-standing tasks where results can be obtained immediately
- **Level of control***:* high
- **Data collection***:* process and product measurement, questionnaires
- **Data analysis***:* parametric and non-parametric statistics, compare central tendencies of treatments, groups
- **Pro's***:* help establishing causal relationships, confirm theories
- **Con's***:* representative? Challenging to plan in a real-world environment. Application in industrial context requires compromises

Case study

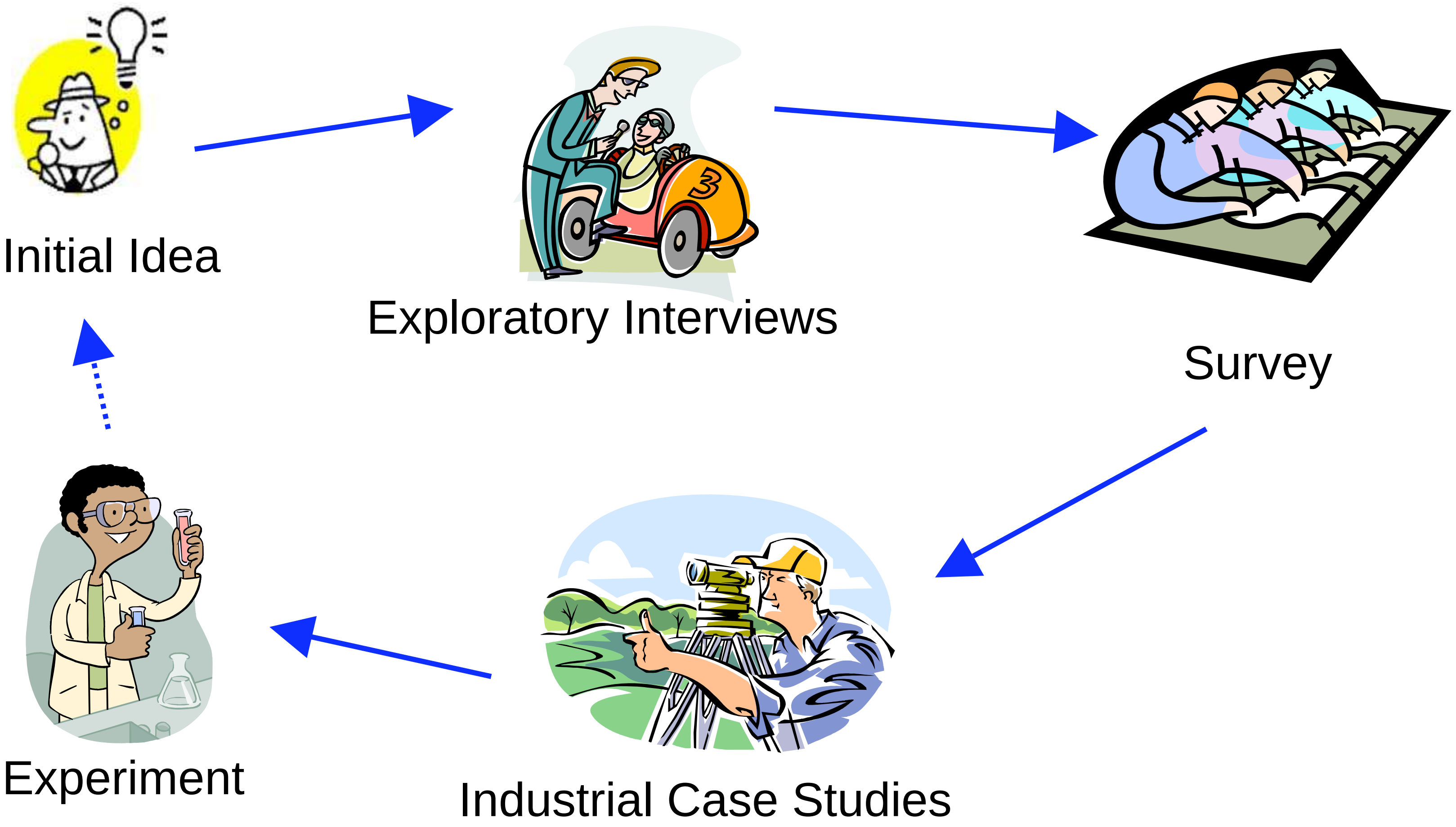
- **When appropriate***:* change (new technology) is wide-ranging throughout the development process, want to assess a change in a typical situation
- **Level of control***:* medium
- **Data collection***:* product and process measurement, questionnaires, interviews
- **Data analysis***:* compare case study results to a baseline (sister project, company baseline)
- **Pro's***:* applicable to real world projects, help answering why and how questions, provide qualitative insight
- **Con's***:* difficult to implement a case study design, confounding factors, analysis of results is subjective

Survey

- **When appropriate**: for early exploratory analysis. Technology change implemented across a large number of projects, description of results, influence factors, differences and commonalities
- **Level of control**: low
- **Data collection**: questionnaires, interviews
- **Data analysis**: comparing different populations among respondents, association and trend analysis, consistency of scores
- **Pro's**: generalization of results is usually easier (than case study), applicable in practice
- **Con's**: little control of variables, questionnaire design is difficult (validity, reliability), execution is often time consuming (interviews)

Empirical Life-cycle

Method Development



Initial Idea

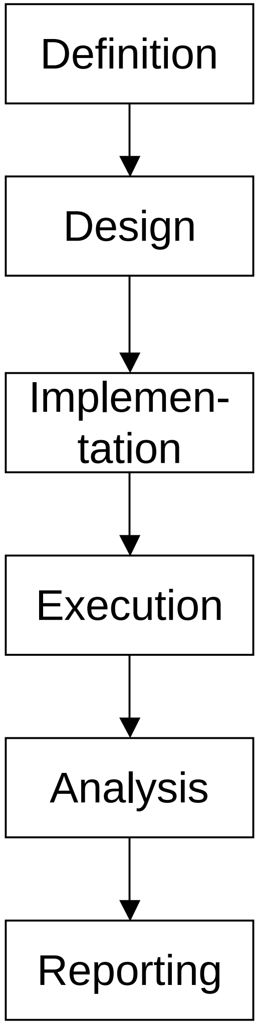
Exploratory Interviews

Survey

Industrial Case Studies

Experiment

A process for conducting empirical studies

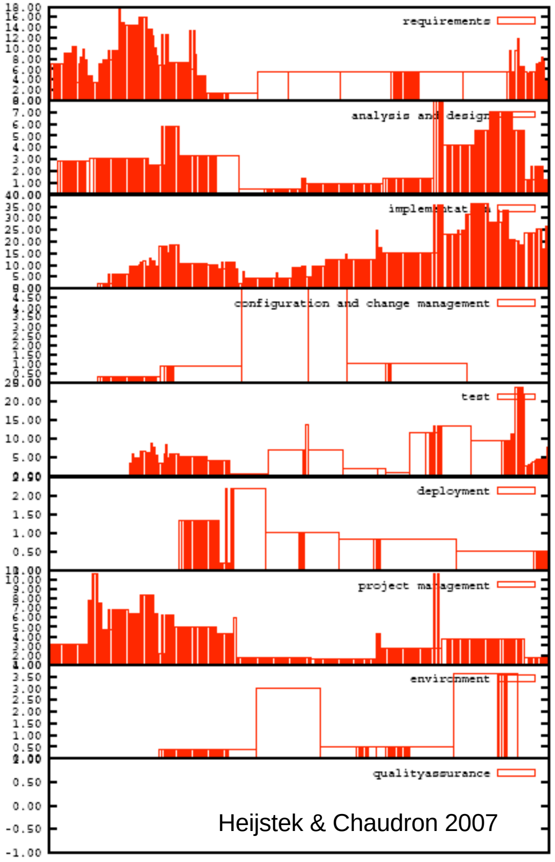


- Determine study goal and research hypothesis. Select type of empirical study to be employed.
- Operationalize study goal and hypothesis.
  Make study plan: what needs to be done by whom and when.
- Prepare material required to conduct the study.
- Run study according to plan and collect required data (data collection).
- Analyze collected data to answer operationalized study goal and hypotheses
- Report your study so that external parties are able to understand results and context of the study.

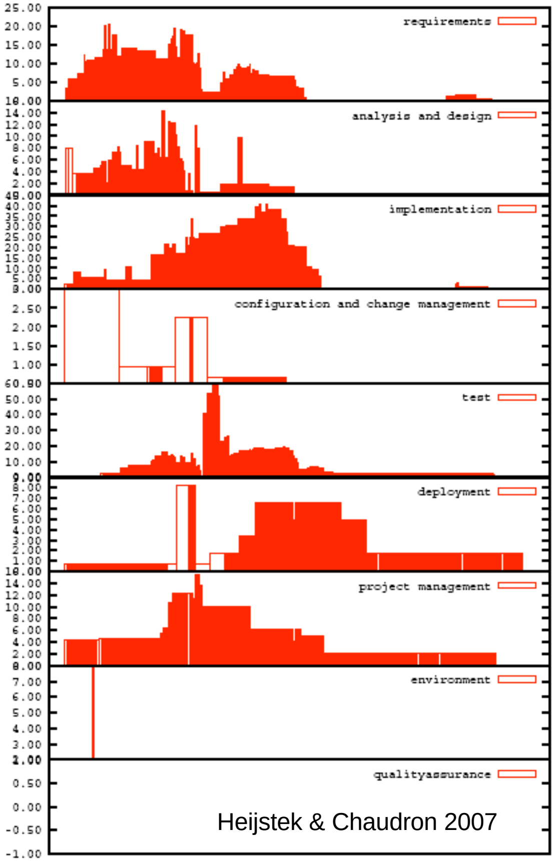
Validity

- Are the results valid for the sample population?
- Are the results valid for the population to which we would like to generalize?
- Threats to Validity
  - Conclusion validity
    - Relation between treatment and outcome
  - Internal validity
    - Treatment → outcome = causal relationship?
  - Construct validity
    - Relation between theory and observation
  - External validity
    - Generalizability of the result

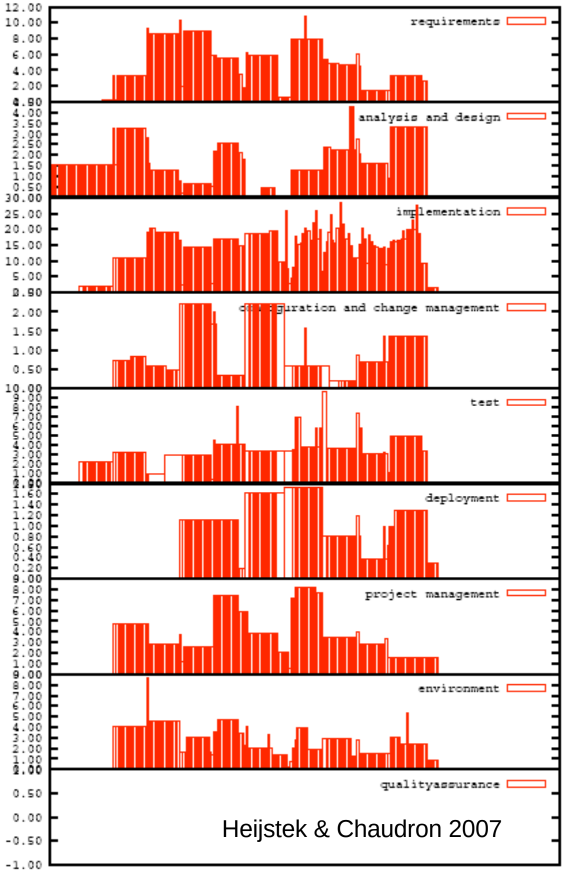
RUP Humps from 3 (largish) projects



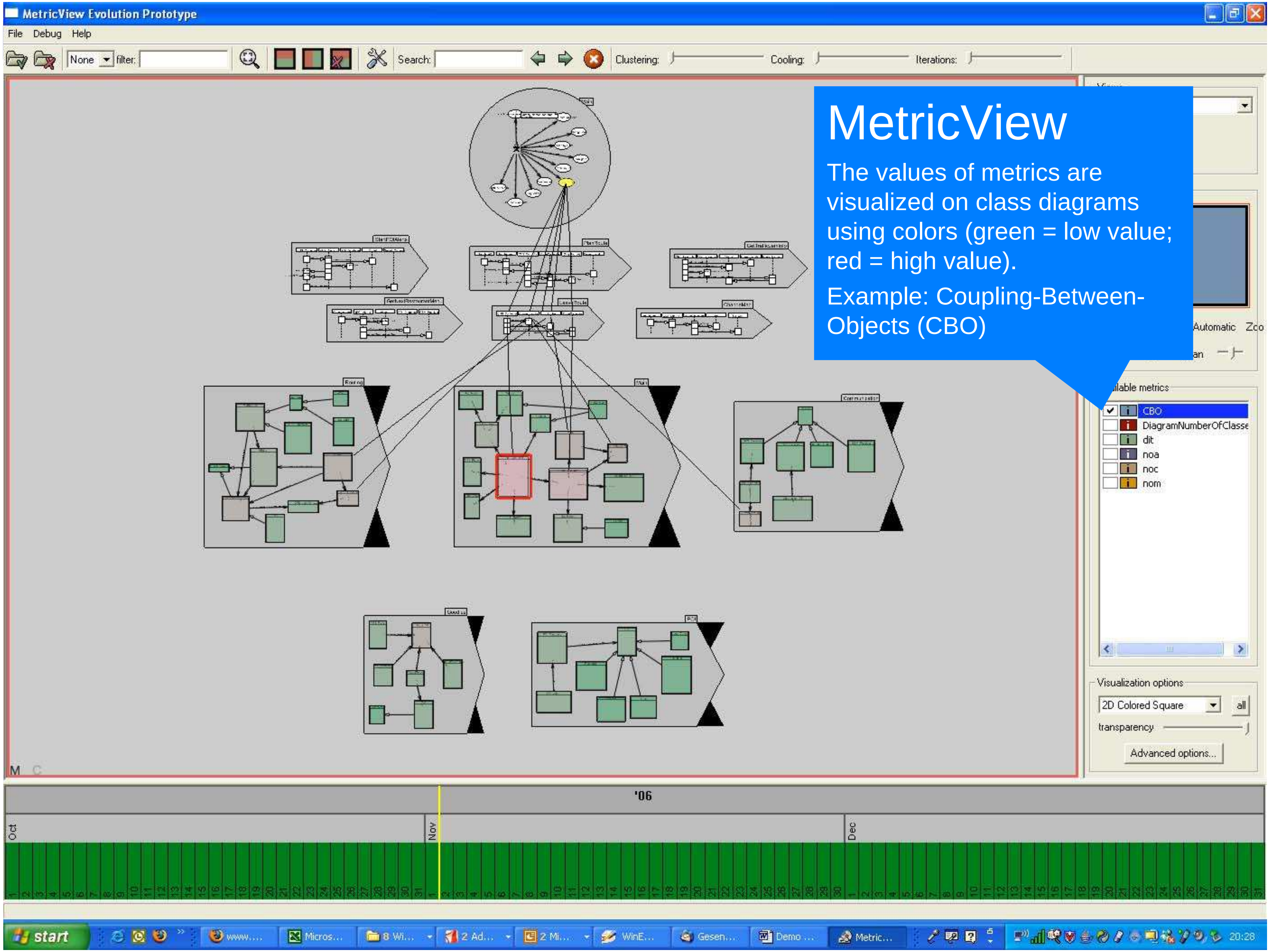
Heijstek & Chaudron 2007



Heijstek & Chaudron 2007



Heijstek & Chaudron 2007



MetricView

The values of metrics are visualized on class diagrams using colors (green = low value; red = high value).

Example: Coupling-Between-Objects (CBO)

ilable metrics

☑ CBO
□ DiagramNumberOfClasse
□ dit
□ noa
□ noc
□ nom

Visualization options

2D Colored Square

all

transparency

Advanced options...

Conclusions

- Empirical Research is essential for validation of methods/techniques/processes in practice;
  - □ Feedback for improvement
  - □ Collaboration between industry and academia is essential
- Different study-types (‘strategies’) are possible.
  - □ Depending on the goal and context
  - □ Good preparation is important
  - □ Good literature is available

References

[1] A. Endres, D. Rombach, A Handbook of Software and Systems Engineering – Empirical Observations, Laws and Theories, Pearson Addison Wesley, 2003.

[2] C. Wohlin, P. Runeson, M. Höst, Magnus C. Ohlsson, Björn Regnell, and Anders Wesslen, Experimentation in Software Engineering - An Introduction. The Kluwer International Series in Software Engineering, Kluwer Academic Publishers, 2000.

[3] R. van Solingen and E. Berghout, The Goal/Question/Metric Method. McGraw-Hill, 1999.

[4] N. Fenton and Shari L. Pfleeger, Software Metrics: A Rigorous Practical Approach. London: International Thompson Computer Press, 1996.

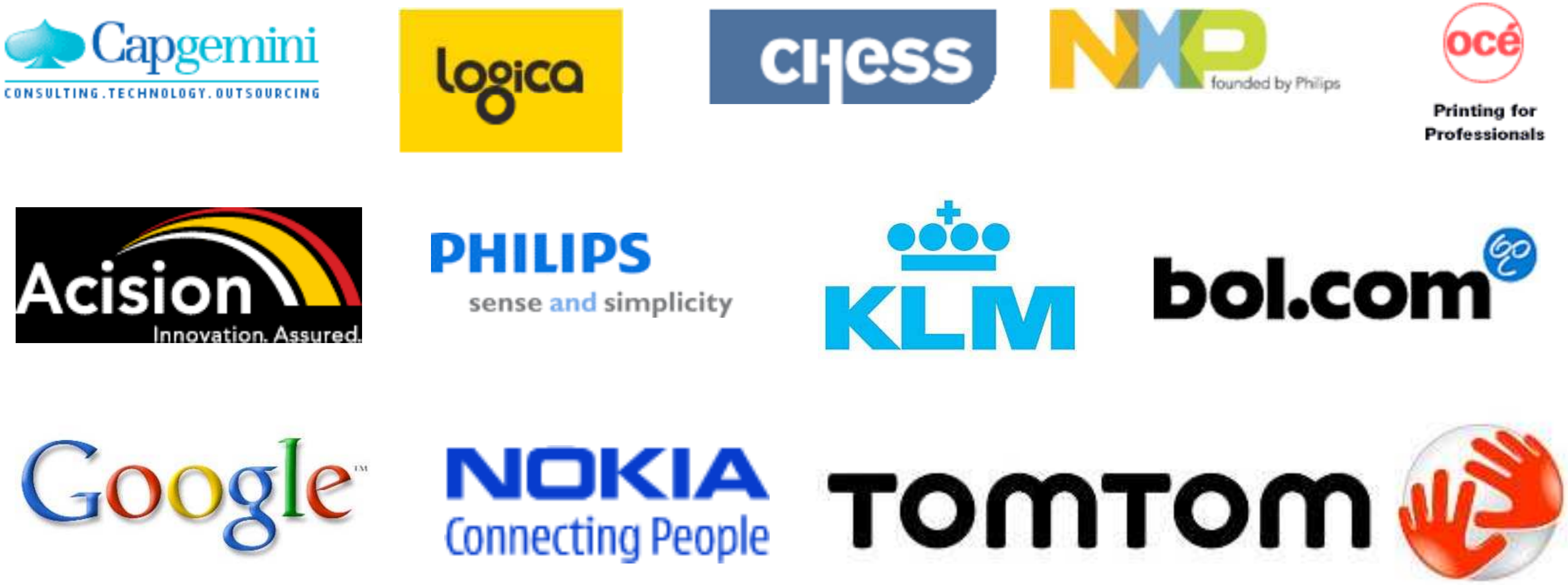
[5] B. Freimut, T. Punter, S. Biffl, M. Ciolkowski, State-of-the-art in Empirical studies, IESE-Report No. 017.02/E & ViSEK report No. 007/02, Kaiserslautern, Fraunhofer IESE, March 2002.

[6] T. Punter, M. Ciolkowski, B. Freimut, I. John, Conducting on-line surveys in software engineering, ACM IEEE Int. Symposium on Empirical Software Engineering (ISESE'03), Los Alamitos, IEEE, pp. 80-88.

[7] B. Kitchenham, Evaluating Software Engineering Methods and Tools - Part 9: Quantitative Case Study Methodology, ACM SIGSOFT Software Engineering Notes, vol. 23, pp. 24-26, Jan. 1998.

[8] M.V. Zelkowitz, D.R. Wallace, Experimental models for validating Technology, IEEE Computer, vol. 31 no. 5, pp. 23-31, May 1998.

M.Sc. Eindprojecten met …

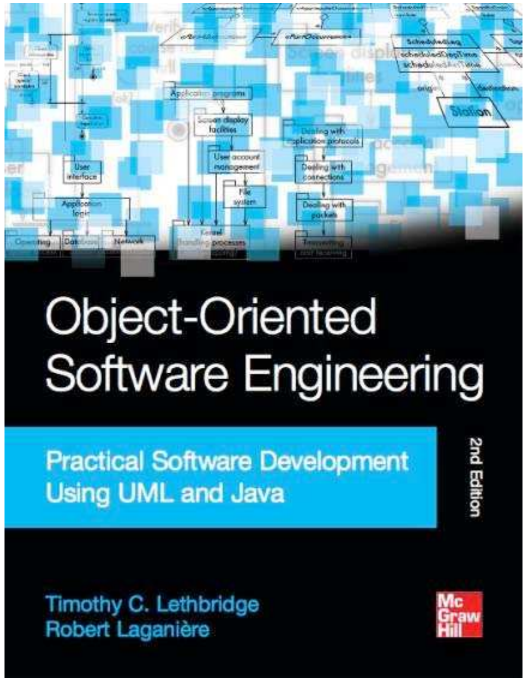


Guest lecture 24 april 11-13

And many more (including companies/universities) abroad …

Highlights SE

Book: Object-Oriented Software Engineering, Timothy C. Lethbridge, Robert Laganière (2nd Ed.)



- Ch 1: introduction to the subject
- Ch 2: OO-basics
- Ch 4: Requirements
- Ch 5 & Ch 8: Modeling using UML
- Ch 6: Design patterns
- Ch 9: Architecture & Designing
- Ch 10: Testing / Quality Assurance
- Ch 11: Management (Estimation, Risk)
- Websites: www.mhhe.com/lethbridge en www.llsoeng.com

Project Management

- People are key
  - Get good people, Make them happy, Set them loose
- Manage Risk Early and Frequently
- Anticipate changes

Requirements Engineering

- Understand the domain
- SMART
- Manage Change

Software Architecture

- Principle decisions about design of a system
- Describe using multiple views
- Validate architecture

  review, measure, prototype

- Mathematics is not a careful march down a well-cleared highway, but a journey into a strange wilderness, where the explorers often get lost.
  Rigour should be a signal to the historian that the maps have been made and the real explorers have gone elsewhere.
  [Anglin, W.S.]

Releasing your potential

Gastcollege 10-04-2008

Testing in practice

Bart Knaack

Logica

Bart.Knaack@logica.com

Releasing your potential

Agenda

- Introduction
- Why testing?
- Testing Theory versus Practice
- Risk and Requirements Based Testing
- Testmanagement
- Pauze
- Future Testing
- Stories from the real world

Introduction

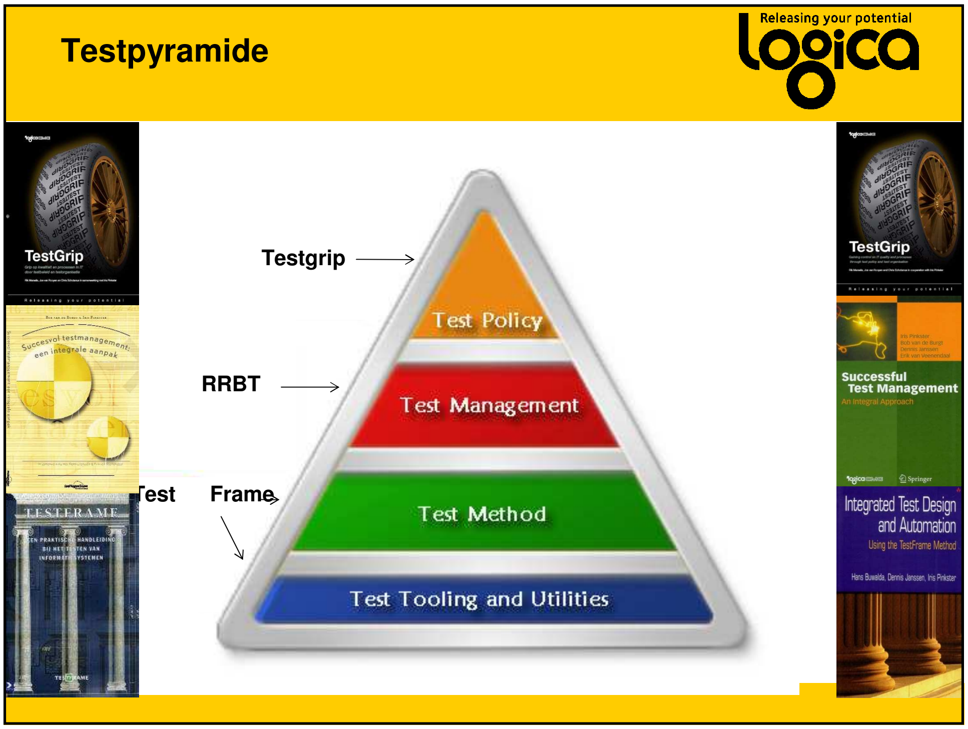
Releasing your potential logica

- Who am I?
- What have I done so far?

Who am I?

Releasing your potential logica

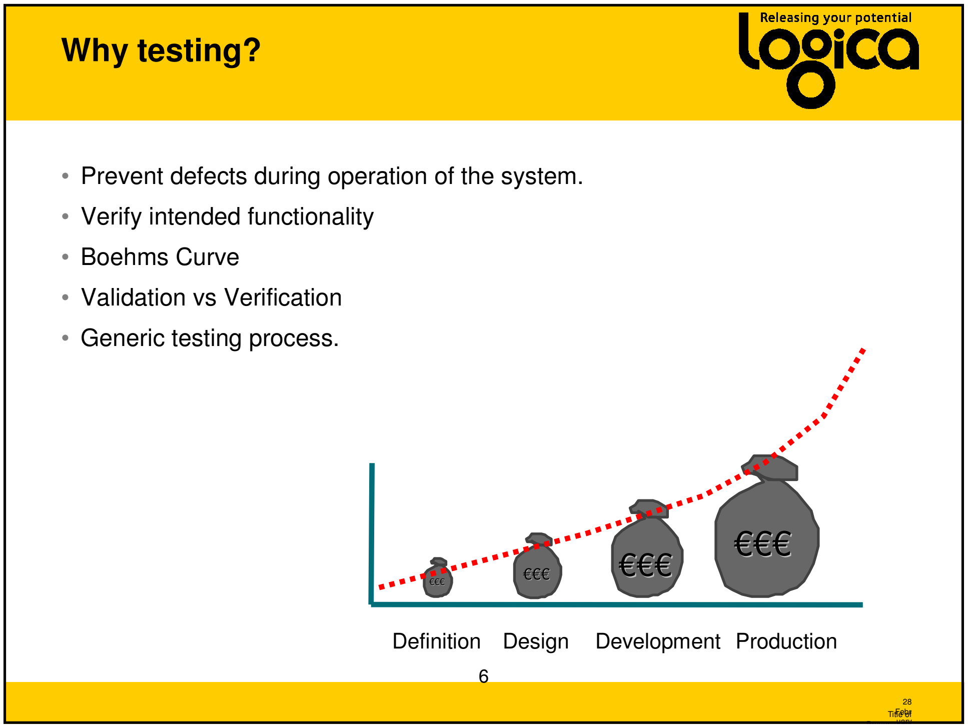
- Bart Knaack, Senior Test Advisor, Logica, The Netherlands
- 15 years experience in IT, of which 12 in testing.
- Developer, Development Lead, Tester, Testautomator, Testcoordinator, Testmanager, Testadvisor.
- Trainer in Testmanagement
- ISEB practionner
- SEI accredited CMMi Appraiser
- Father of 2 kids (age 6 and 8)



Testpyramide

Releasing your potential

Testgrip
RRBT
Test Frame
Test Policy
Test Management
Test Method
Test Tooling and Utilities
TestGrip
Successful Test Management
An Integral Approach
Integrated Test Design and Automation
Using the TestFrame Method
Succesvol testmanagement: een integrale aanpak
TESTFRAME

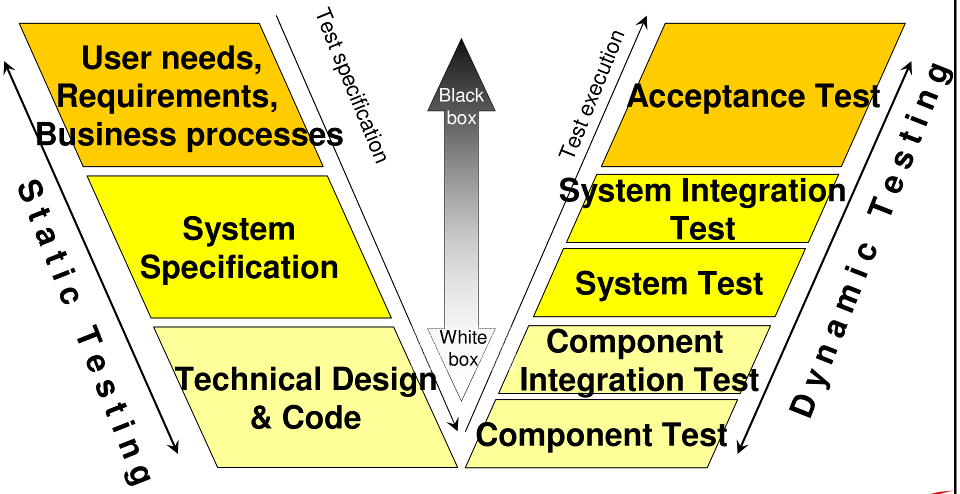


Why testing?

Releasing your potential

- Prevent defects during operation of the system.
- Verify intended functionality
- Boehms Curve
- Validation vs Verification
- Generic testing process.

Test levels in the V-model



User needs, Requirements, Business processes

System Specification

Technical Design & Code

Test specification

Black box

White box

Test execution

Acceptance Test

System Integration Test

System Test

Component Integration Test

Component Test

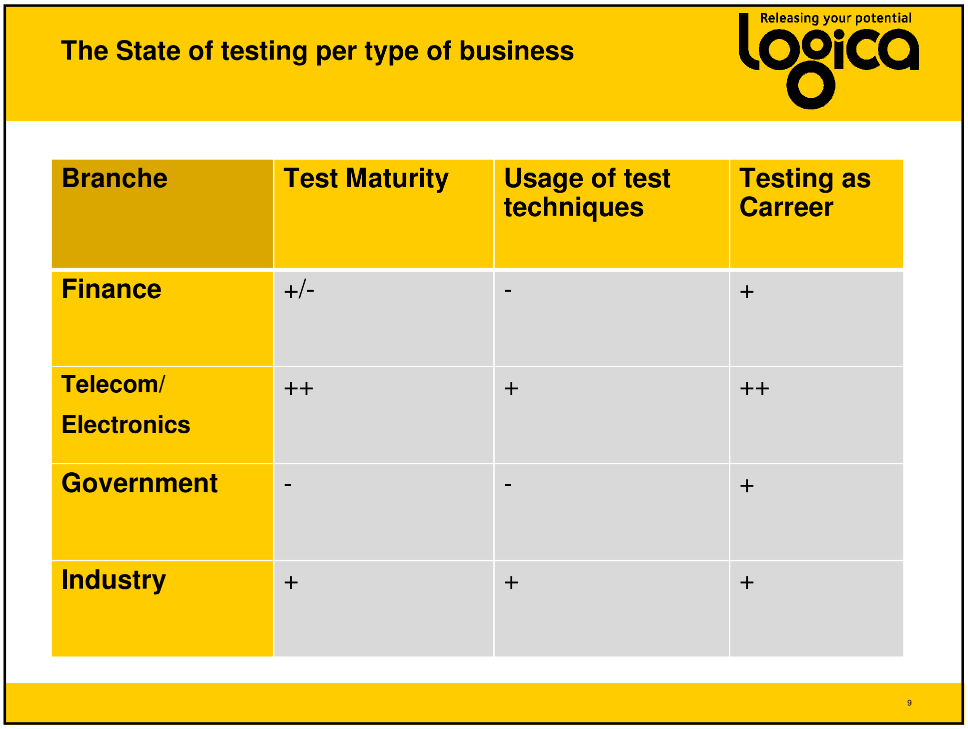
Static Testing

Dynamic Testing

terms according to: ISTQB

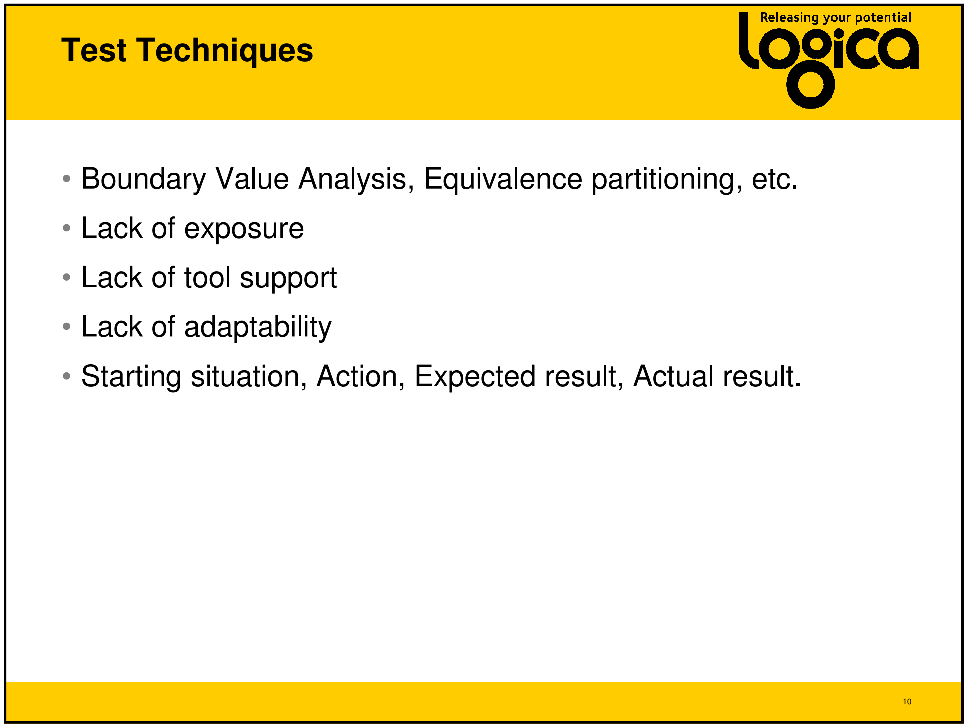
Testing Theory versus Practice

- The State of testing per type of business
- Test Techniques



The State of testing per type of business

Branche	Test Maturity	Usage of test techniques	Testing as Carreer
Finance	+/-	-	+
Telecom/ Electronics	++	+	++
Government	-	-	+
Industry	+	+	+



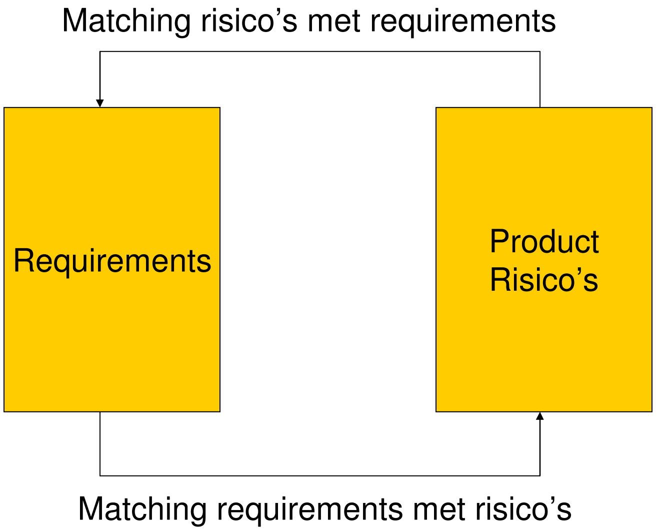
Test Techniques

- Boundary Value Analysis, Equivalence partitioning, etc.
- Lack of exposure
- Lack of tool support
- Lack of adaptability
- Starting situation, Action, Expected result, Actual result.

Risk and Requirements Based Testing

- Risk and Requirements based testing approach
- The role and responsibilities of the testmanager
- The eight-facetet testmanagement model

RRBT: risico’s versus requirements


Matching risico’s met requirements

Matching requirements met risico’s

Wel risico, geen requirement:

•Aanvullen requirement (eerder fouten vinden)

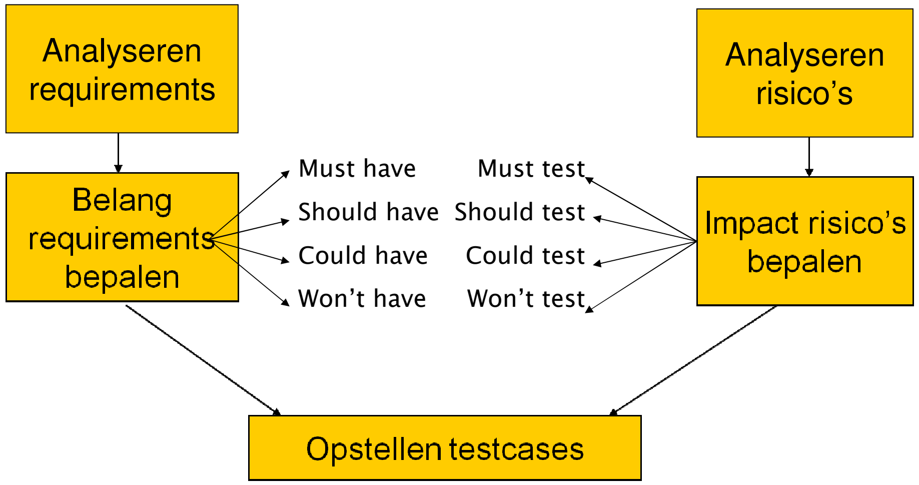
•Afvoeren risico (niet onnodig testen)

Wel requirement, geen risico:

•Risico lijst aanpassen (betere dekkingsgraad test)

•Requirement afvoeren (niet onnodig ontwikkelen, geen “franje”)

Combineren productrisico’s en requirements



Analyseren requirements

Belang requirements bepalen

Must have

Should have

Could have

Won’t have

Must test

Should test

Could test

Won’t test

Analyseren risico’s

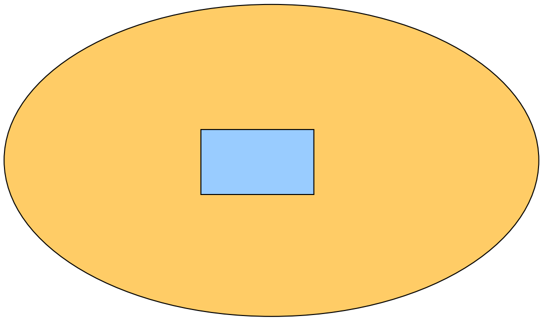
Impact risico’s bepalen

Opstellen testcases

Risk and requirements based testing

- Identify the stakeholders
- Determine productrisks
- Link product risks to requirements and quality attributes
- Determine testsorts.
- Determine acceptance criteria

Stakeholders & verschillende eisen



```
ERROR: undefined
OFFENDING COMMAND: ›–

STACK:

(
 ¿   ¯ »z{  ª   ß Vw| ØP9TxØ 5d”K B —¯LMq=  Ł·  +·(  æzR)f  ß+ ¤ !S»i 5…`  P ¸‹˙Y  qC Æ [u
)
/abreve
-dictionary-
/CharStrings
-dictionary-
-dictionary-
/Private
-dictionary-
-dictionary-
false
-filestream-
-mark-
false
(./n019023l.pfb)
/NimbusSanL-ReguItal
/Helvetica-Oblique
-mark-
/Helvetica-Oblique
1860085
/Helvetica-Oblique
/Font
/Helvetica-Oblique
```