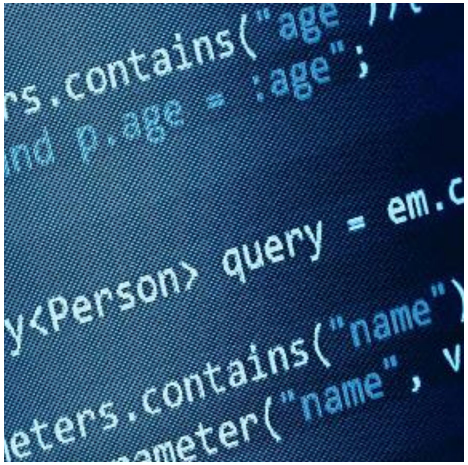
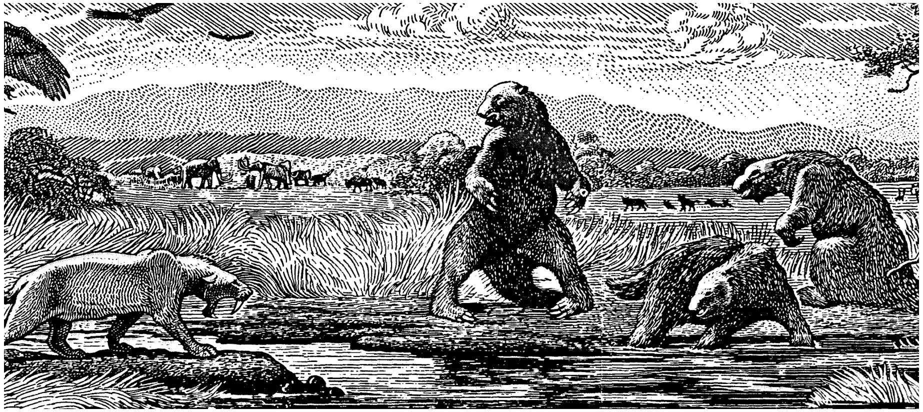
CS3500 Software Engineering



```
s.contains("age"))
nd p.age = :age";
y<Person> query = em.c
eters.contains("name")
ameter("name", v
```

2017/2018



Dept. Computer Science
Dr. Klaas-Jan Stol

University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Welcome to CS3500

Introduction and Overview

Contents

Hello.

- My name is Klaas-Jan Stol
- But you can call me Klaas
- Email: k.stol@cs.ucc.ie
  - Always put CS3500 in subject line
  - Send from your student email address (@ucc.ie)
  - Expect a response on queries within 48 hours.
  - No response after 48h? Try again!

Introduction

1. Goals

2. Assumptions

3. Important information

Goals of CS3500

- To provide a broad introduction to Software Engineering

- To teach you basic terminology, practices, and processes of professional software engineering

- To prepare you for future team projects and work placement (Sem. 2)

- To prepare you for a professional career in SE

What I expect: My assumptions

1. You spend about 8h / week on this module

2. Two 1-hour lectures per week
   At least ~6 hours study/tasks in your own time (incl. the labs)

3. You will ask questions if things are unclear to you.
   Either in class or by email.

4. You want to grow beyond being a good programmer, and become a well-trained Software Engineer.

5. You will read your email at least every 24 hours.
   "I only saw your message now" is not a valid excuse.

6. If you experience difficulties that affect your ability to do work in CS3500, you will let me know as soon as possible.

7. You want to pass this module.

Advice for success in CS3500

- Attend all lectures
- Do all assignments in time
- Make notes during lectures
- Do all reading assignments in time

Advice for success in CS3500

There will be reading assignments

1. Switch OFF your devices
   - Resist the attraction of the blinking light
2. Sit in a quiet room without TV/radio/others
   - "I can listen to music while reading" is a myth debunked by research.
3. Make notes preferably with pen & paper.
   - Research showed your brain remembers stuff better.

Studying is does not simply mean reading material, it means mastering the material.

- Learn by doing

CS3500 Activities

- No textbook – but lots of reading:
  - 1 paper per week
  - Available through Moodle
- A set of graded tasks
  - Team work
  - Teams / team size to be decided later.
- Written final exam
  - Covers lectures
  - Covers selected material from papers
  - Covers skills acquired in graded tasks

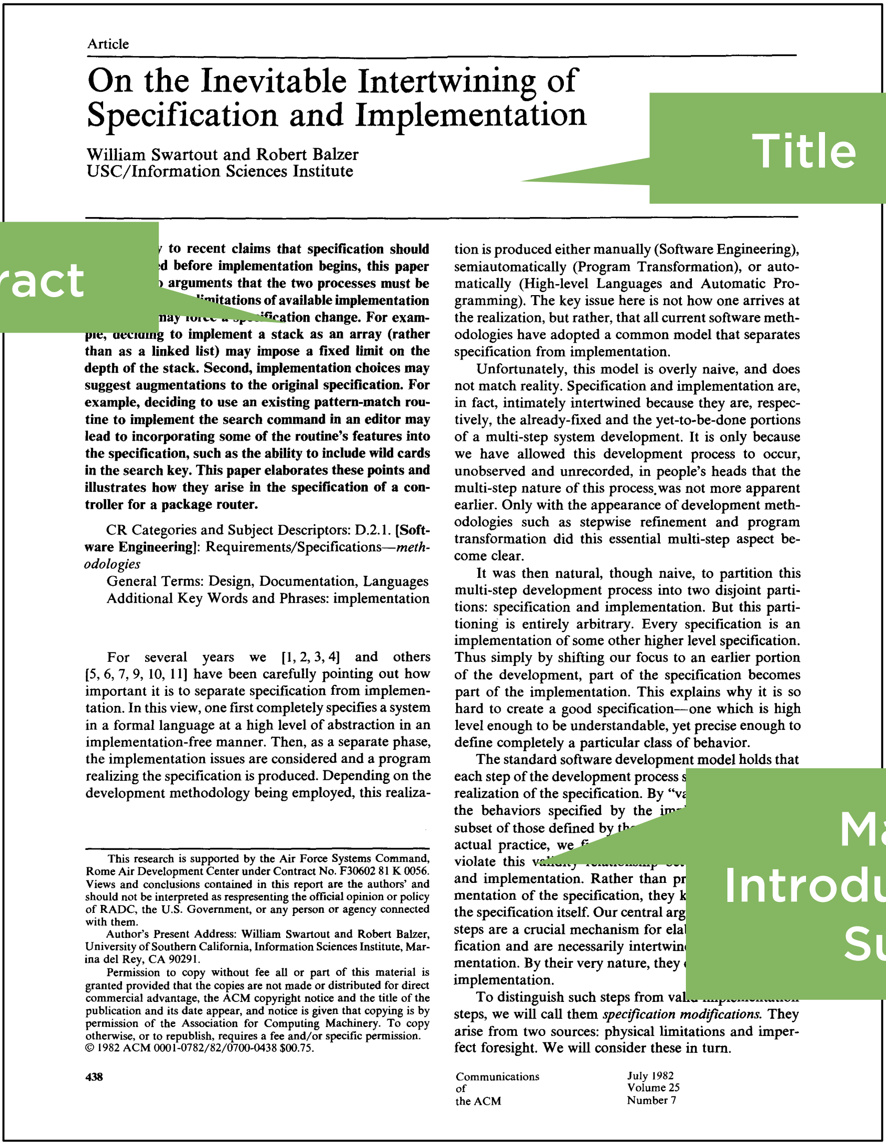
How to read a paper

- Typical paper structure
  - Introduction
  - Body
  - Summary

- References
  - No need to read those

- A paper has only a few take-away messages
  - Train yourself in identifying them.
  - Make notes while reading

What does a paper looks like?



Article

On the Inevitable Intertwining of Specification and Implementation

William Swartout and Robert Balzer
USC/Information Sciences Institute

y to recent claims that specification should
d before implementation begins, this paper
arguments that the two processes must be
limitations of available implementation
may force a specification change. For example, deciding to implement a stack as an array (rather than as a linked list) may impose a fixed limit on the depth of the stack. Second, implementation choices may suggest augmentations to the original specification. For example, deciding to use an existing pattern-match routine to implement the search command in an editor may lead to incorporating some of the routine's features into the specification, such as the ability to include wild cards in the search key. This paper elaborates these points and illustrates how they arise in the specification of a controller for a package router.

CR Categories and Subject Descriptors: D.2.1. [**Software Engineering**]: Requirements/Specifications—*methodologies*

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: implementation

For several years we [1, 2, 3, 4] and others [5, 6, 7, 9, 10, 11] have been carefully pointing out how important it is to separate specification from implementation. In this view, one first completely specifies a system in a formal language at a high level of abstraction in an implementation-free manner. Then, as a separate phase, the implementation issues are considered and a program realizing the specification is produced. Depending on the development methodology being employed, this realization is produced either manually (Software Engineering), semiautomatically (Program Transformation), or automatically (High-level Languages and Automatic Programming). The key issue here is not how one arrives at the realization, but rather, that all current software methodologies have adopted a common model that separates specification from implementation.

Unfortunately, this model is overly naive, and does not match reality. Specification and implementation are, in fact, intimately intertwined because they are, respectively, the already-fixed and the yet-to-be-done portions of a multi-step system development. It is only because we have allowed this development process to occur, unobserved and unrecorded, in people's heads that the multi-step nature of this process was not more apparent earlier. Only with the appearance of development methodologies such as stepwise refinement and program transformation did this essential multi-step aspect become clear.

It was then natural, though naive, to partition this multi-step development process into two disjoint partitions: specification and implementation. But this partitioning is entirely arbitrary. Every specification is an implementation of some other higher level specification. Thus simply by shifting our focus to an earlier portion of the development, part of the specification becomes part of the implementation. This explains why it is so hard to create a good specification—one which is high level enough to be understandable, yet precise enough to define completely a particular class of behavior.

The standard software development model holds that
each step of the development process s
realization of the specification. By "va
the behaviors specified by the im
subset of those defined by th
actual practice, we f
violate this v
and implementation. Rather than pr
mentation of the specification, they k
the specification itself. Our central arg
steps are a crucial mechanism for ela
fication and are necessarily intertwin
mentation. By their very nature, they
implementation.

To distinguish such steps from val
steps, we will call them *specification modifications*. They arise from two sources: physical limitations and imperfect foresight. We will consider these in turn.

This research is supported by the Air Force Systems Command, Rome Air Development Center under Contract No. F30602 81 K 0056. Views and conclusions contained in this report are the authors' and should not be interpreted as respresenting the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

Author's Present Address: William Swartout and Robert Balzer, University of Southern California, Information Sciences Institute, Marina del Rey, CA 90291.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0001-0782/82/0700-0438 $00.75.

438

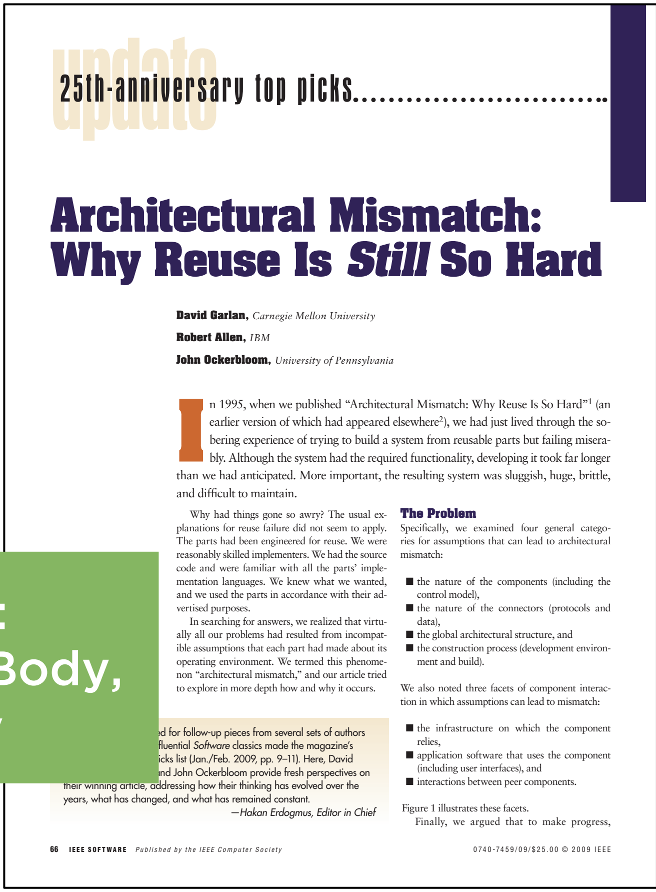
Communications
of
the ACM

July 1982
Volume 25
Number 7

Title

Abstract

Main text:
Introduction, Body,
Summary



update

25th-anniversary top picks..........................

Architectural Mismatch: Why Reuse Is *Still* So Hard

David Garlan, *Carnegie Mellon University*

Robert Allen, *IBM*

John Ockerbloom, *University of Pennsylvania*

In 1995, when we published "Architectural Mismatch: Why Reuse Is So Hard"[1] (an earlier version of which had appeared elsewhere[2]), we had just lived through the sobering experience of trying to build a system from reusable parts but failing miserably. Although the system had the required functionality, developing it took far longer than we had anticipated. More important, the resulting system was sluggish, huge, brittle, and difficult to maintain.

Why had things gone so awry? The usual explanations for reuse failure did not seem to apply. The parts had been engineered for reuse. We were reasonably skilled implementers. We had the source code and were familiar with all the parts' implementation languages. We knew what we wanted, and we used the parts in accordance with their advertised purposes.

In searching for answers, we realized that virtually all our problems had resulted from incompatible assumptions that each part had made about its operating environment. We termed this phenomenon "architectural mismatch," and our article tried to explore in more depth how and why it occurs.

ed for follow-up pieces from several sets of authors
fluential *Software* classics made the magazine's
icks list (Jan./Feb. 2009, pp. 9–11). Here, David
nd John Ockerbloom provide fresh perspectives on
their winning article, addressing how their thinking has evolved over the years, what has changed, and what has remained constant.

—Hakan Erdogmus, Editor in Chief

The Problem

Specifically, we examined four general categories for assumptions that can lead to architectural mismatch:

- the nature of the components (including the control model),
- the nature of the connectors (protocols and data),
- the global architectural structure, and
- the construction process (development environment and build).

We also noted three facets of component interaction in which assumptions can lead to mismatch:

- the infrastructure on which the component relies,
- application software that uses the component (including user interfaces), and
- interactions between peer components.

Figure 1 illustrates these facets.

Finally, we argued that to make progress,

66 IEEE SOFTWARE *Published by the IEEE Computer Society* 0740-7459/09/$25.00 © 2009 IEEE

Lectures & Labs

- All lecture slides will be made available on Moodle
  - CS3500.2018
  - http://cs4.ucc.ie/moodle
  - Lecture slides made available after lecture
- Labs are not yet scheduled and will be announced later on Moodle/Announcements
  - Labs provide opportunity to ask questions, but are primarily reserved lab time to perform graded assignments
- Graded assignments are likely to take more than scheduled lab time

Written final exam

- I will provide example questions during the semester.

- Questions require written answers
  - Not multiple-choice

- Covers all lecture materials, key points from required papers, and lab assignment topics.

Assessment

- 5 ECTS credits
- Total marks to earn: 100
- Written exam: 80 marks
  - 1.5 h (=90 min)
- 5 graded tasks: 5 x 4 marks
  - Late submission → 0 marks
  - No extensions except with dr.'s note

Plagiarism reminder

1. Plagiarism is presenting someone else's work as your own. It is a violation of UCC Policy and there are strict and severe penalties.

2. You must read and comply with the UCC Policy on Plagiarism www.ucc.ie/en/exams/procedures-regulations/

3. The Policy applies to *all* work submitted, including software.

4. You can expect that your work will be checked for evidence of plagiarism or collusion.

5. In some circumstances it may be acceptable to reuse a small amount of work by others, but *only* if you provide explicit acknowledgement and justification.

6. If in doubt ask your module lecturer *prior* to submission. Better safe than sorry!

All staff have a professionally "trained eye" for spotting plagiarism, so expect to be caught if plagiarizing.

Overview

1.

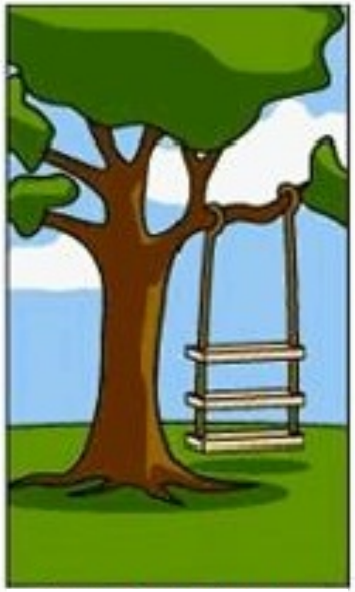
I don’t want to learn theory, I just want to program!

2.

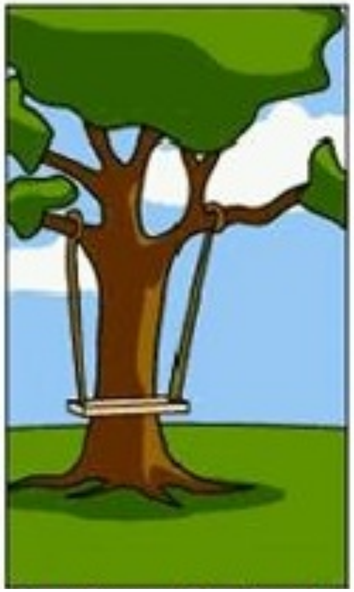
Overview of topics

A Tale of a Typical Software Project

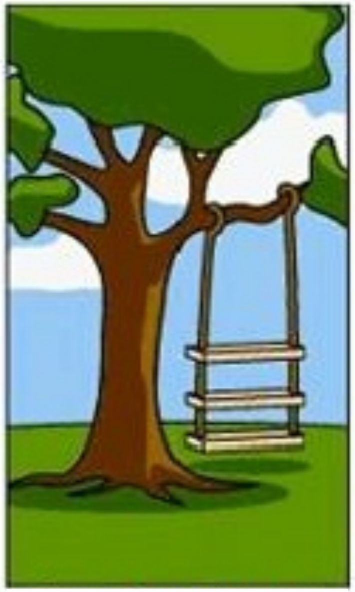
How the customer explained it



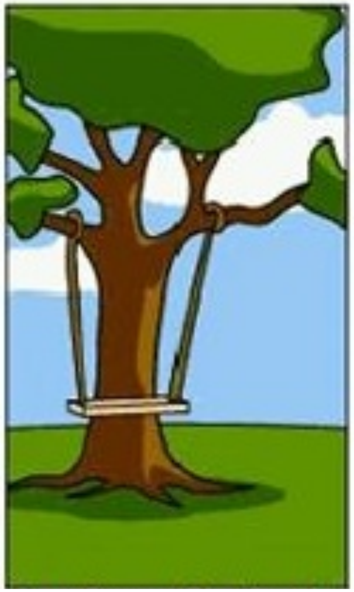
How the customer explained it



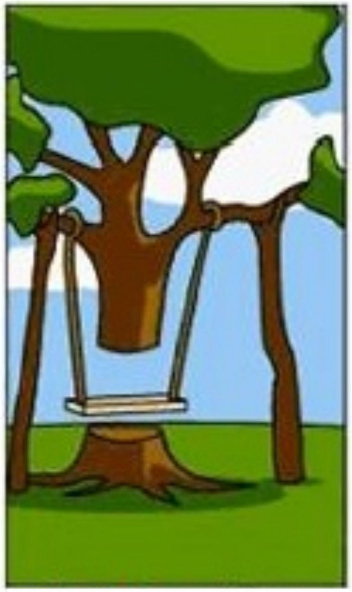
How the project leader understood it



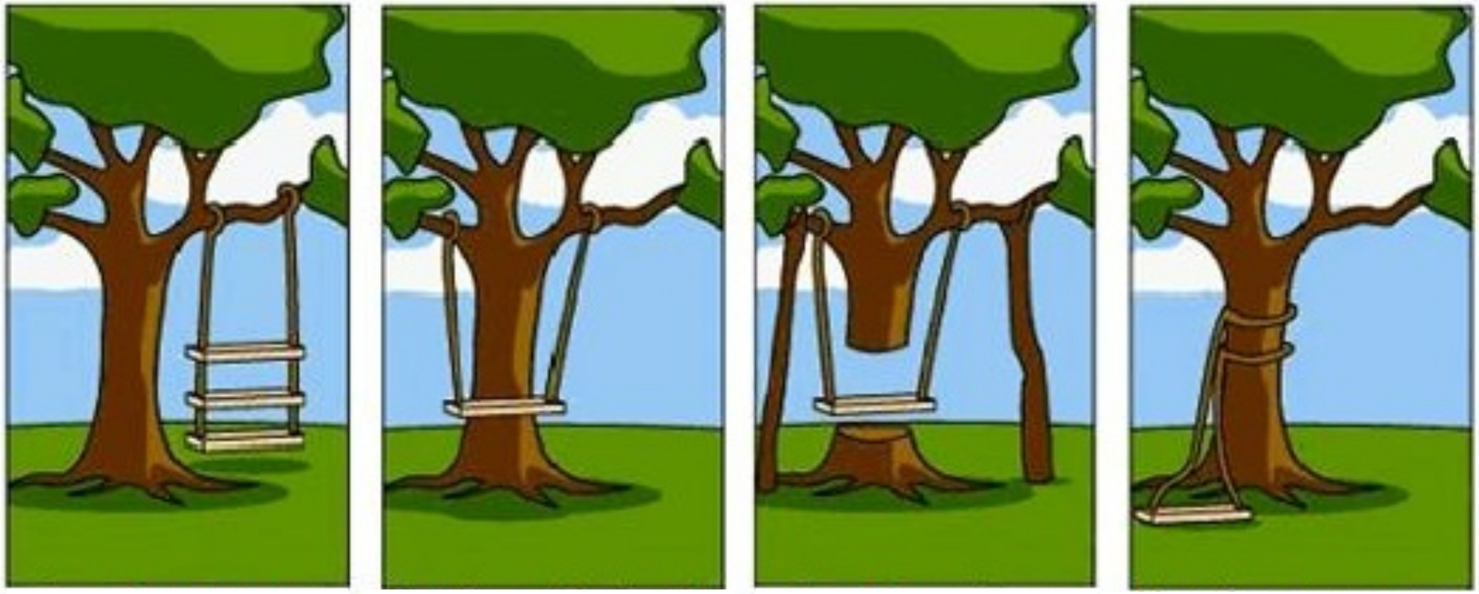
How the customer explained it



How the project leader understood it



How the engineer designed it

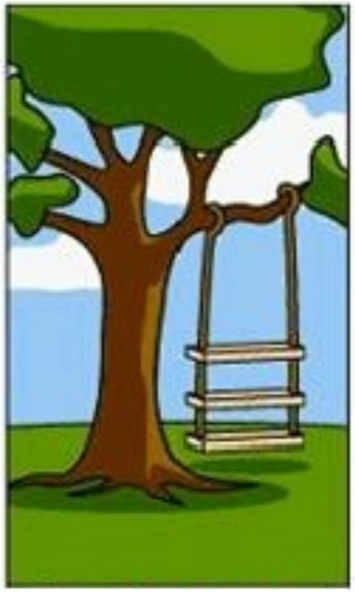


How the customer explained it

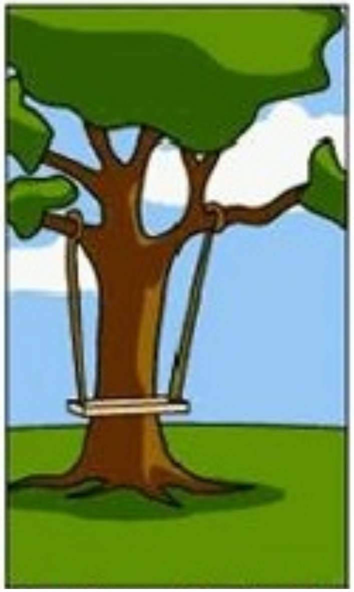
How the project leader understood it

How the engineer designed it

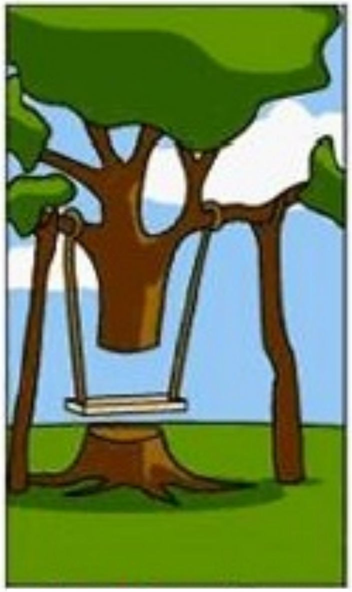
How the programmer wrote it



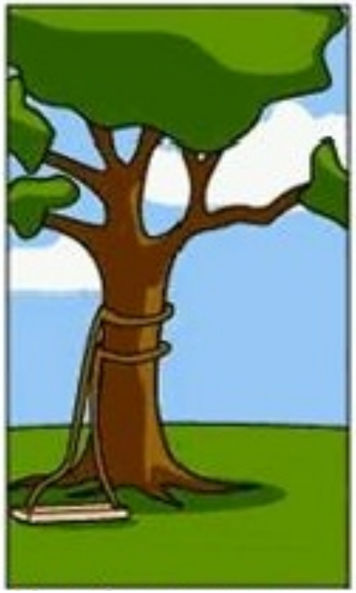
How the customer explained it



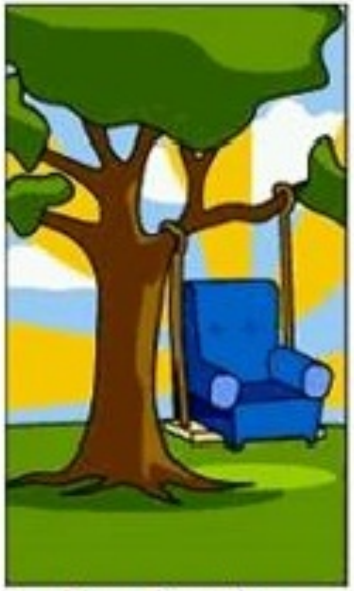
How the project leader understood it



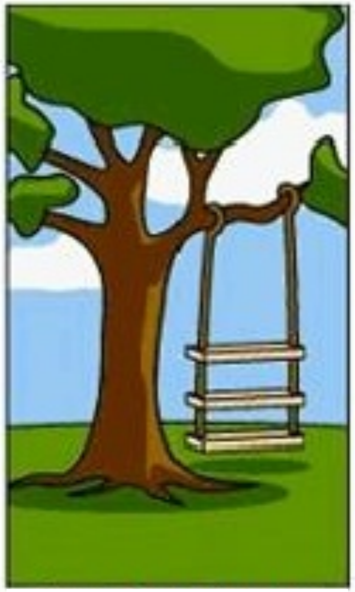
How the engineer designed it



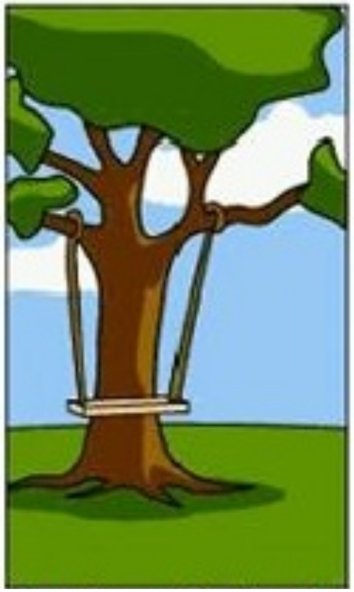
How the programmer wrote it



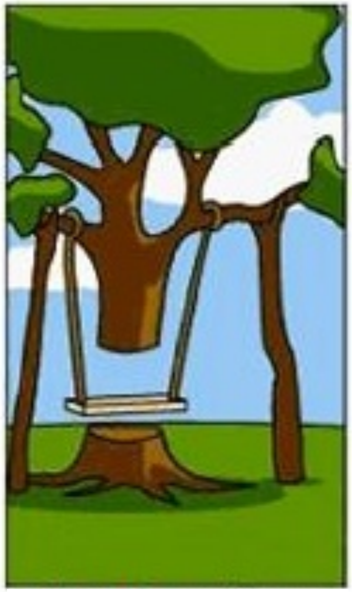
How the sales executive described it



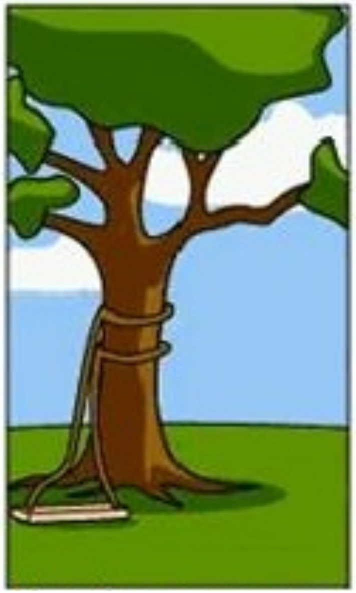
How the customer explained it



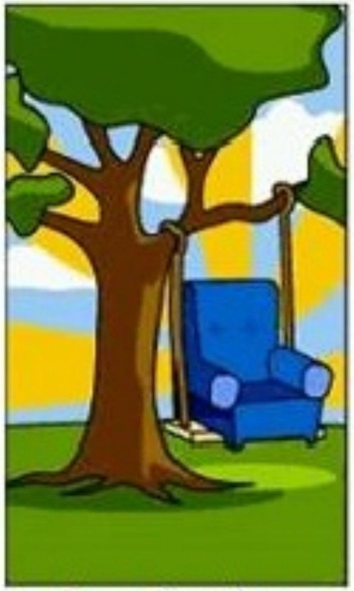
How the project leader understood it



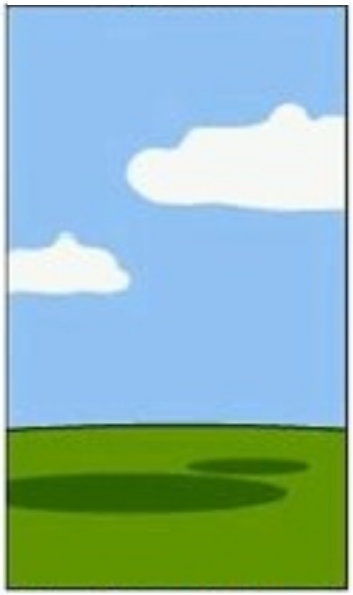
How the engineer designed it



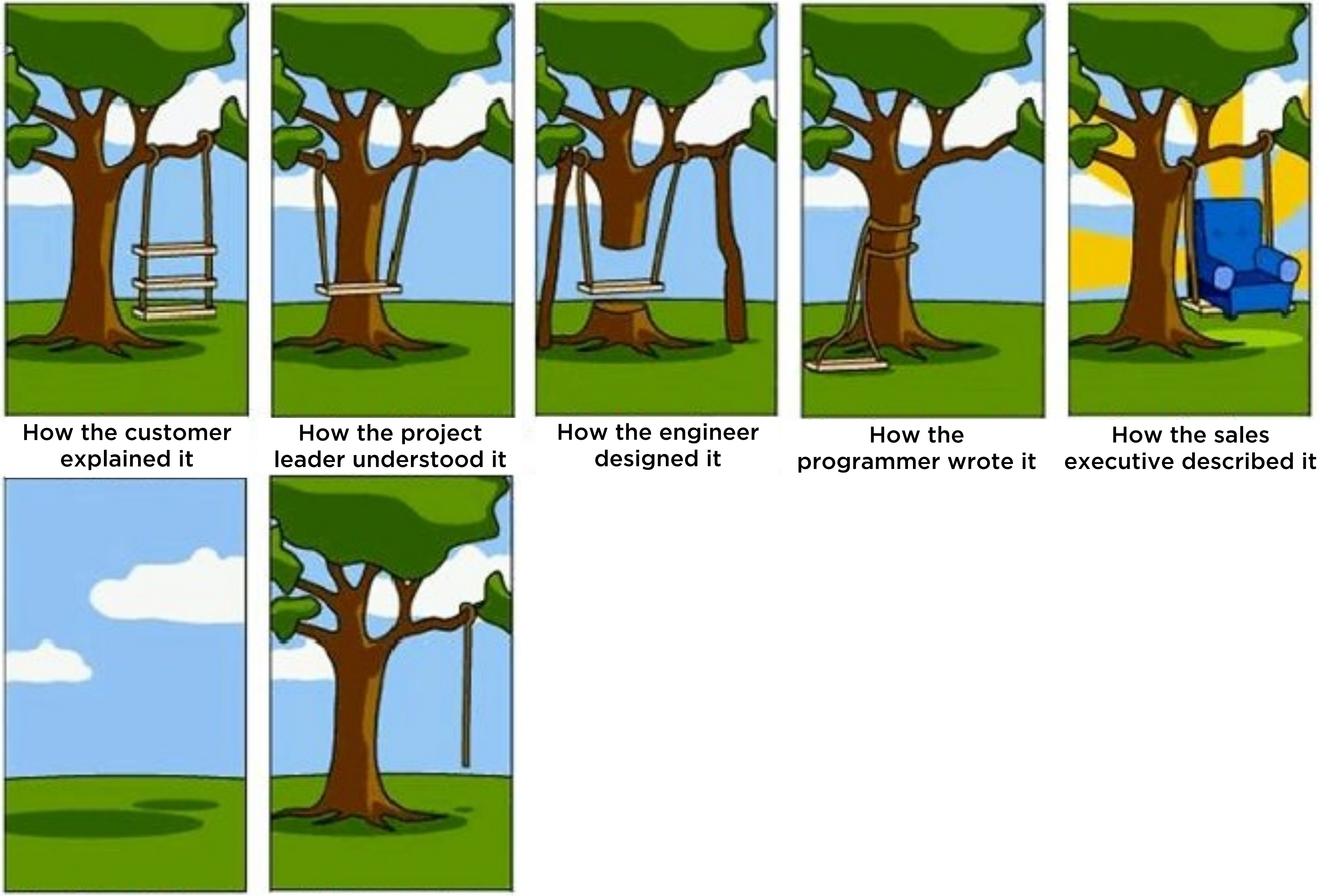
How the programmer wrote it



How the sales executive described it



How the project was documented



How the customer explained it

How the project leader understood it

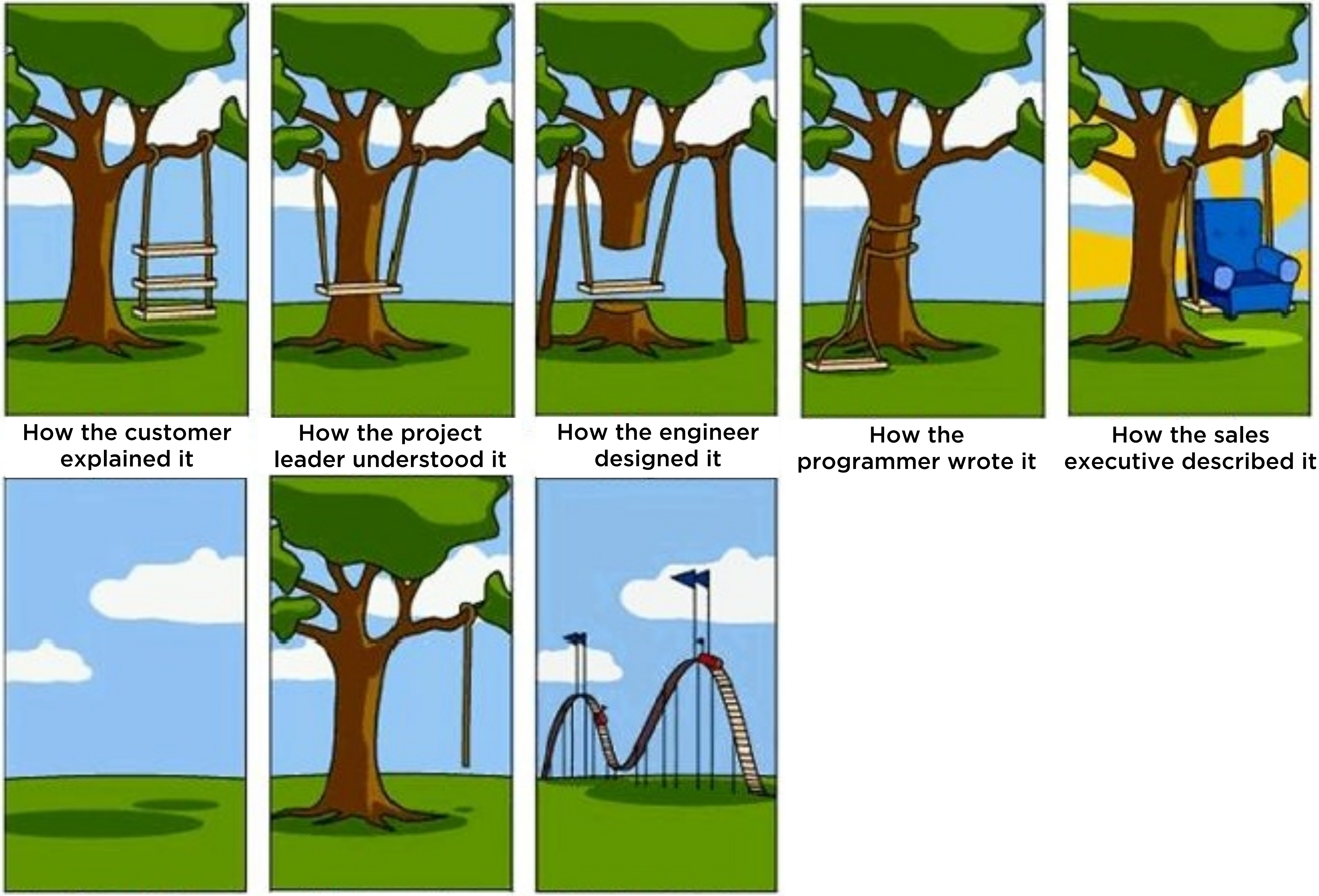
How the engineer designed it

How the programmer wrote it

How the sales executive described it

How the project was documented

What operations installed



How the customer explained it

How the project leader understood it

How the engineer designed it

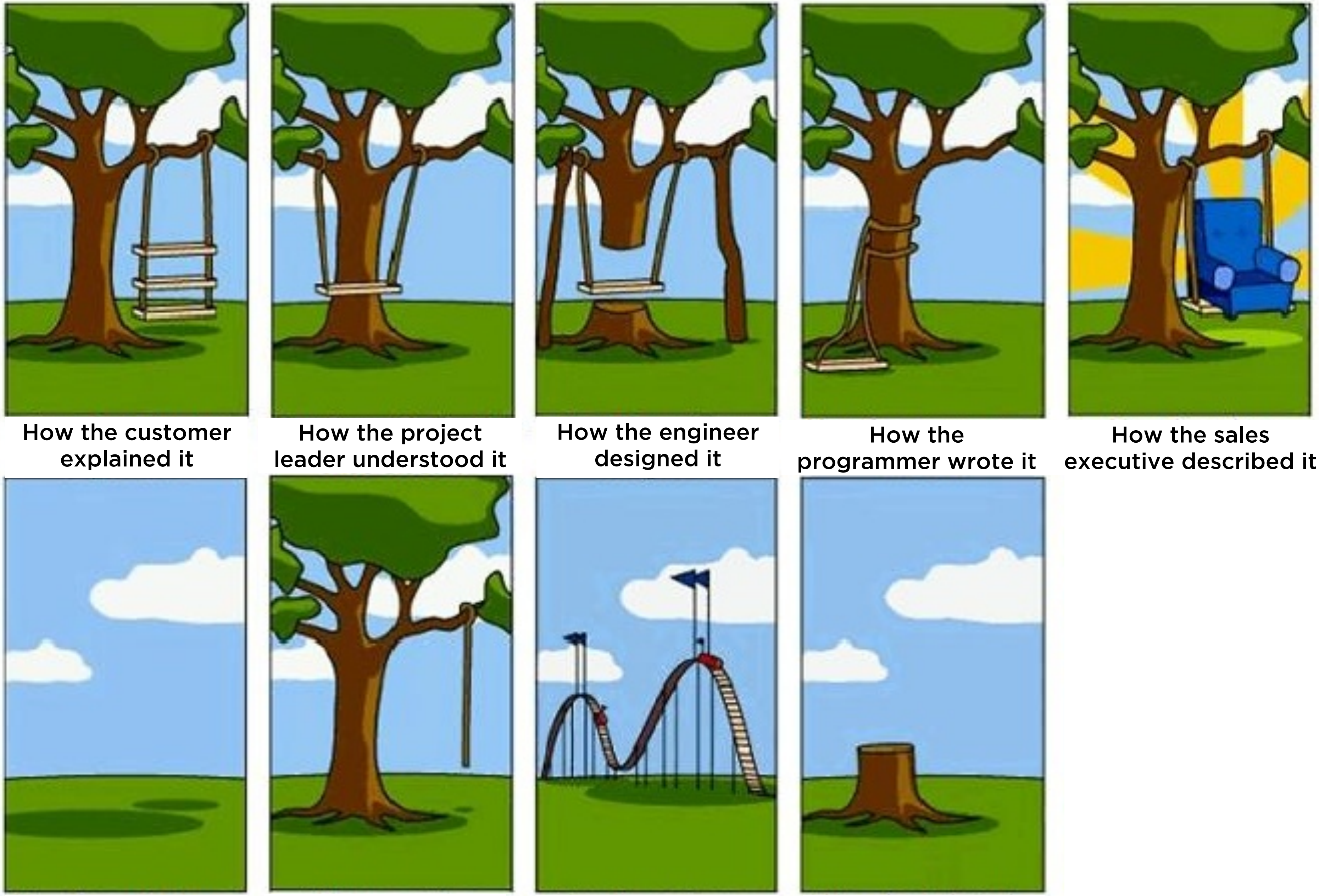
How the programmer wrote it

How the sales executive described it

How the project was documented

What operations installed

How the customer was billed



How the customer explained it

How the project leader understood it

How the engineer designed it

How the programmer wrote it

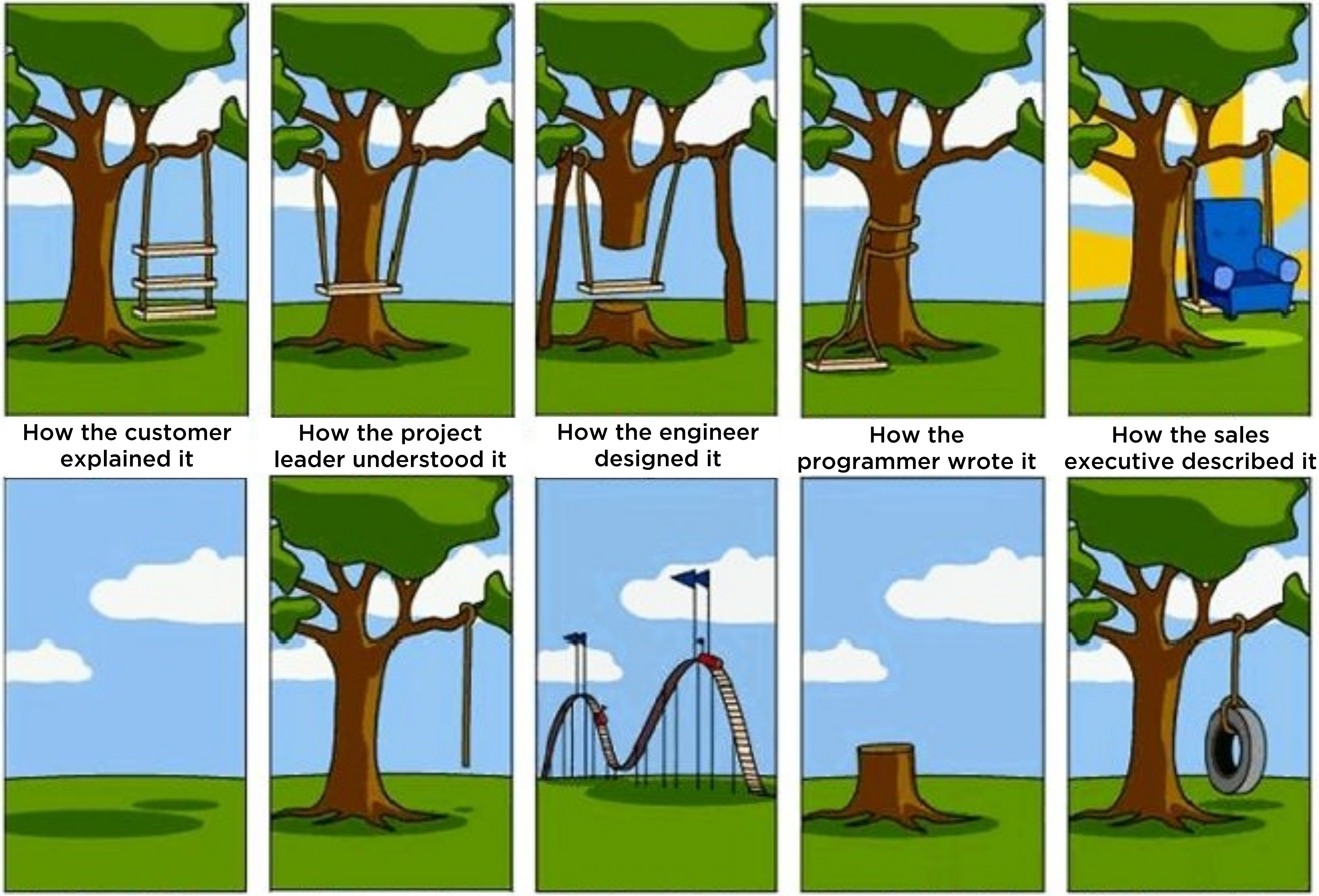
How the sales executive described it

How the project was documented

What operations installed

How the customer was billed

How the helpdesk supported it



How the customer explained it

How the project leader understood it

How the engineer designed it

How the programmer wrote it

How the sales executive described it

How the project was documented

What operations installed

How the customer was billed

How the helpdesk supported it

What the customer really needed

History of SE

Key activities in SE

Software architecture

Software design

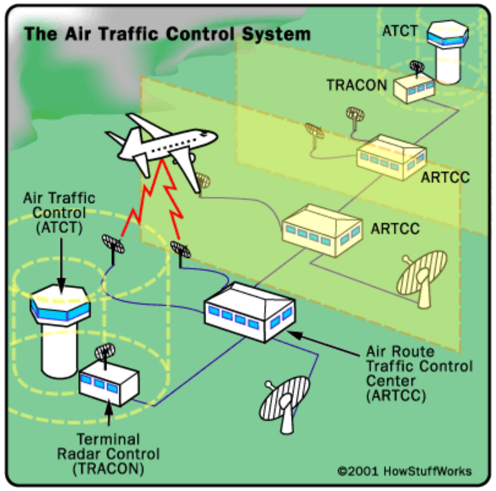
Unified Modeling Language (UML)

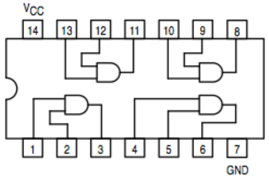
Software processes and methods

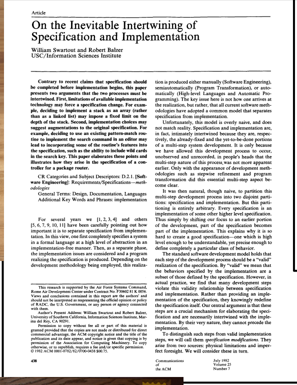
Design patterns

Software project management

CS3500: an overview in pictures







Article

On the Inevitable Intertwining of Specification and Implementation

William Swartout and Robert Balzer
USC/Information Sciences Institute

Contrary to recent claims that specification should be completed before implementation begins, this paper presents two arguments that the two processes must be intertwined. First, limitations of available implementation technology may force a specification change. For example, deciding to implement a stack as an array (rather than as a linked list) may impose a fixed limit on the depth of the stack. Second, implementation choices may suggest augmentations to the original specification. For example, deciding to use an existing pattern-match routine to implement the search command in an editor may lead to incorporating some of the routine's features into the specification, such as the ability to include wild cards in the search key. This paper elaborates these points and illustrates how they arise in the specification of a controller for a package router.

CR Categories and Subject Descriptors: D.2.1. [**Software Engineering**]: Requirements/Specifications—*methodologies*

General Terms: Design, Documentation, Languages

Additional Key Words and Phrases: implementation

For several years we [1, 2, 3, 4] and others [5, 6, 7, 9, 10, 11] have been carefully pointing out how important it is to separate specification from implementation. In this view, one first completely specifies a system in a formal language at a high level of abstraction in an implementation-free manner. Then, as a separate phase, the implementation issues are considered and a program realizing the specification is produced. Depending on the development methodology being employed, this realization is produced either manually (Software Engineering), semiautomatically (Program Transformation), or automatically (High-level Languages and Automatic Programming). The key issue here is not how one arrives at the realization, but rather, that all current software methodologies have adopted a common model that separates specification from implementation.

Unfortunately, this model is overly naive, and does not match reality. Specification and implementation are, in fact, intimately intertwined because they are, respectively, the already-fixed and the yet-to-be-done portions of a multi-step system development. It is only because we have allowed this development process to occur, unobserved and unrecorded, in people's heads that the multi-step nature of this process,was not more apparent earlier. Only with the appearance of development methodologies such as stepwise refinement and program transformation did this essential multi-step aspect become clear.

It was then natural, though naive, to partition this multi-step development process into two disjoint partitions: specification and implementation. But this partitioning is entirely arbitrary. Every specification is an implementation of some other higher level specification. Thus simply by shifting our focus to an earlier portion of the development, part of the specification becomes part of the implementation. This explains why it is so hard to create a good specification—one which is high level enough to be understandable, yet precise enough to define completely a particular class of behavior.

The standard software development model holds that each step of the development process should be a "valid" realization of the specification. By "valid" we mean that the behaviors specified by the implementation are a subset of those defined by the specification. However, in actual practice, we find that many development steps violate this validity relationship between specification and implementation. Rather than providing an implementation of the specification, they knowingly redefine the specification itself. Our central argument is that these steps are a crucial mechanism for elaborating the specification and are necessarily intertwined with the implementation. By their very nature, they cannot precede the implementation.

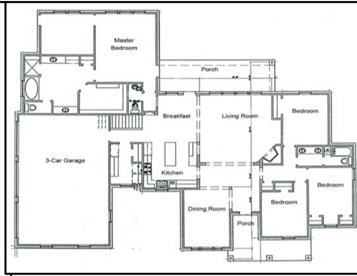
To distinguish such steps from valid implementation steps, we will call them *specification modifications.* They arise from two sources: physical limitations and imperfect foresight. We will consider these in turn.

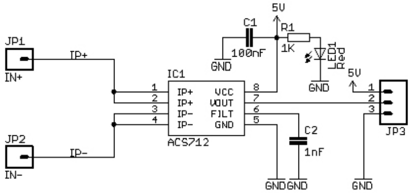
This research is supported by the Air Force Systems Command, Rome Air Development Center under Contract No. F30602 81 K 0056. Views and conclusions contained in this report are the authors' and should not be interpreted as representing the official opinion or policy of RADC, the U.S. Government, or any person or agency connected with them.

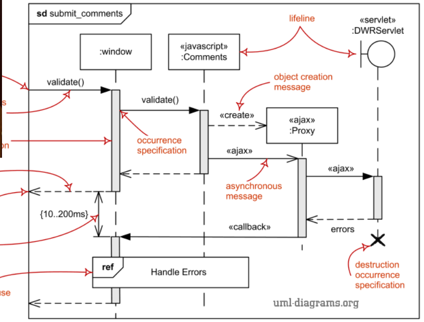
Author's Present Address: William Swartout and Robert Balzer, University of Southern California, Information Sciences Institute, Marina del Rey, CA 90291.

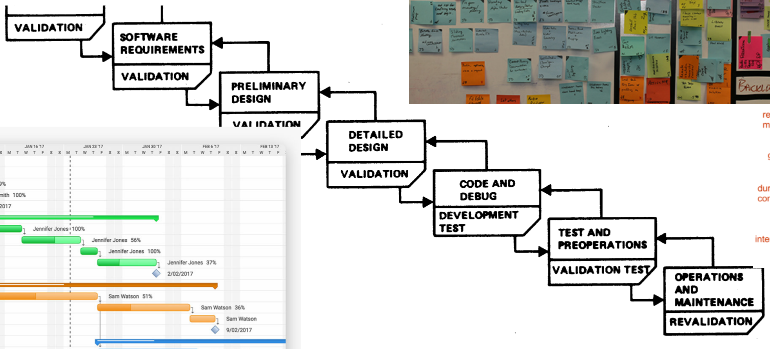
Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1982 ACM 0001-0782/82/0700-0438 $00.75.

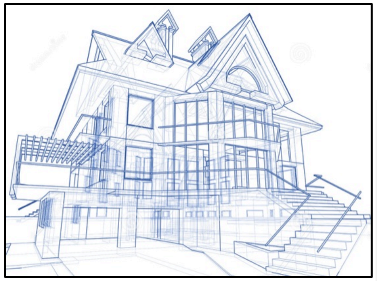
438 Communications of the ACM July 1982 Volume 25 Number 7

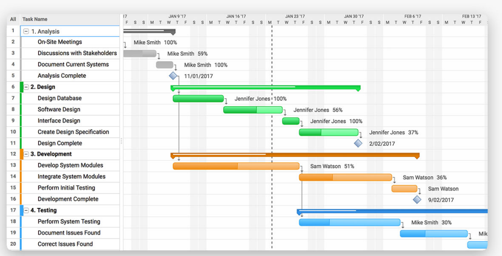




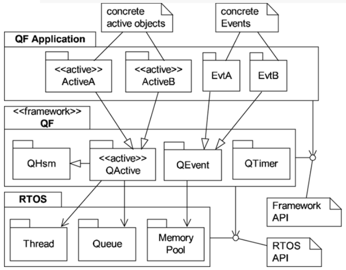


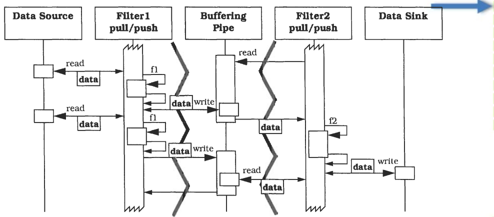


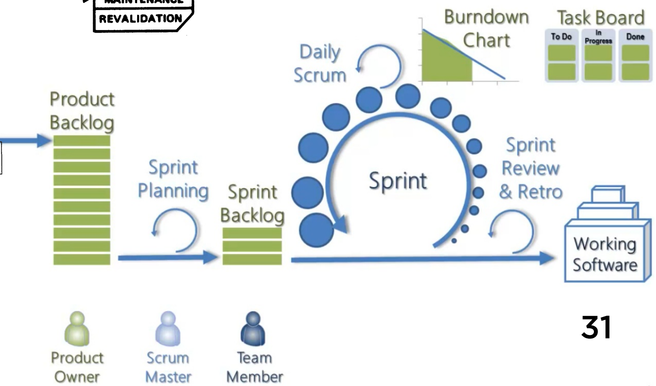




All	Task Name	
1	1. Analysis	
2	On-Site Meetings	Mike Smith 100%
3	Discussions with Stakeholders	Mike Smith 59%
4	Document Current Systems	Mike Smith 100%
5	Analysis Complete	11/01/2017
6	2. Design	
7	Design Database	Jennifer Jones 100%
8	Software Design	Jennifer Jones 56%
9	Interface Design	Jennifer Jones 100%
10	Create Design Specification	Jennifer Jones 37%
11	Design Complete	2/02/2017
12	3. Development	
13	Develop System Modules	Sam Watson 51%
14	Integrate System Modules	Sam Watson 36%
15	Perform Initial Testing	Sam Watson
16	Development Complete	9/02/2017
17	4. Testing	
18	Perform System Testing	Mike Smith 30%
19	Document Issues Found	Mik
20	Correct Issues Found	







What you will learn in CS3500

CS3500 aims to teach you basics of:

- Requirements engineering
- Software design
- Software architecture
- Software patterns
- Software evolution & maintenance
- Project management
- Software processes & methods

Final Points

- This slide deck provides an introduction to CS3500 and an overview of the module.
- Each slide deck has a summary slide like this, but that is not sufficient to remember!
- Feedback on material is welcome!

Thank you for your attention

Questions & suggestions can be sent to:
k.stol@cs.ucc.ie