Introduction to Software Engineering

What is Software Engineering?

- The establishment and use of sound engineering principles in order to obtain economically software that is reliable and works efficiently on real machines.

- As defined in IEEE Standard 610.12:
  - The application of a **systematic**, disciplined, **quantifiable** approach to the **development**, **operation**, and **maintenance** of **software**; that is, the application of engineering to software.

An Opinion

- These definitions are really pretty good
- But they are descriptive, not prescriptive
  - They do not say how to do anything
  - They just say what qualities S.E. should have
  - As a result many people understand SE differently

What is Software Engineering?

- Often compared to civil engineering
  - E.g., building a bridge
- A surprisingly good analogy
  - Size matters: a dog house vs. a skyscraper
  - Team effort with careful planning
  - Similar difficulties to change a given design
  - Many terms come from this metaphor:
    - building, scaffolding, architecture, components, …

What is particular about engineering?

- Any ideas?

What is particular about engineering?

- Many aspects, just some:
  - If you manage to get something to work, remember for next time
  - If something does not work, carefully analyze why and remember the pitfalls
  - Many products have a history of previous versions that get improved, don't start from scratch unless necessary
  - Use standard components with known properties
  - Work & Quality Assessment documented
  - Extensive testing of prototypes before production

Software Engineering vs. Civil Engineering

- But software building often cannot leverage components
  - "Computing is the only profession in which a single mind is obliged to span the distance from a bit to a few hundred megabytes, or nine orders of magnitude." Steve McConnell, "Code Complete"
- Physics guides civil engineering
  - "Einstein argued that there must be a simple explanation of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer."

Software: Axis of variability

- Size
- How humans interact with it
- Requirements stability/knowledge
- Need for reliability
- Need for security
- Portability
- Cost

Microsoft Word

- Size:
- Interactiveness:
- Requirements:
- Reliability:
- Security:
- Portability:
- Cost:

Microsoft Word

- Size: large
- Interactiveness: high
- Requirements: frequent new features
- Reliability: moderate
- Security: low
- Portability: high
- Cost: high

Space shuttle software

- Size:
- Interactiveness:
- Requirements:
- Reliability:
- Security:
- Portability:
- Cost:

Space shuttle software

- Size: large
- Interactiveness: low
- Requirements: stable
- Reliability: very high
- Security: low
- Portability: low
- Cost: high

eBay software

- Size:
- Interactiveness:
- Requirements:
- Reliability:
- Security:
- Portability:
- Cost:

eBay software

- Size: moderate
- Interactiveness: high
- Requirements: frequent new features
- Reliability: moderate
- Security: high
- Portability: low
- Cost: low

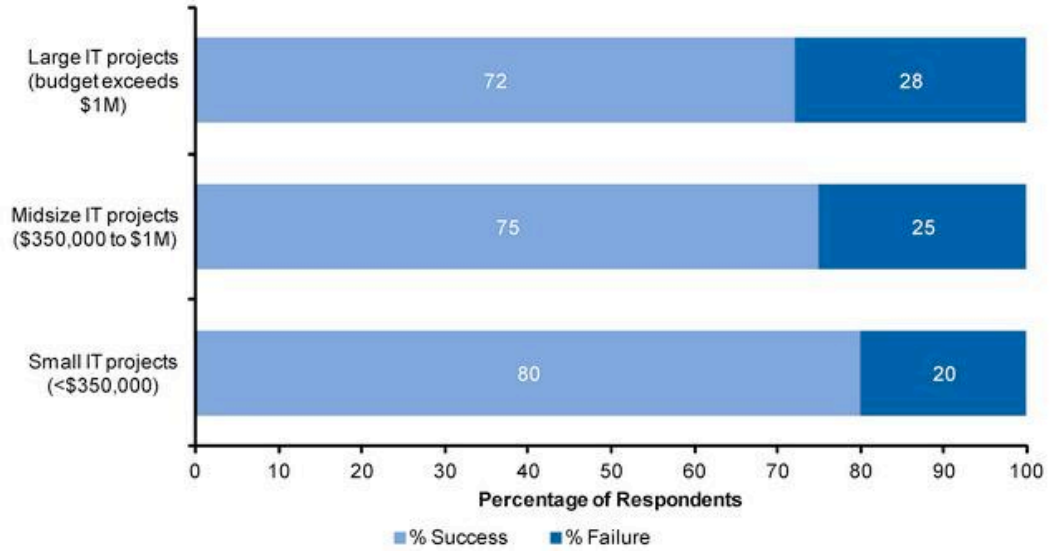
Sample Sizes

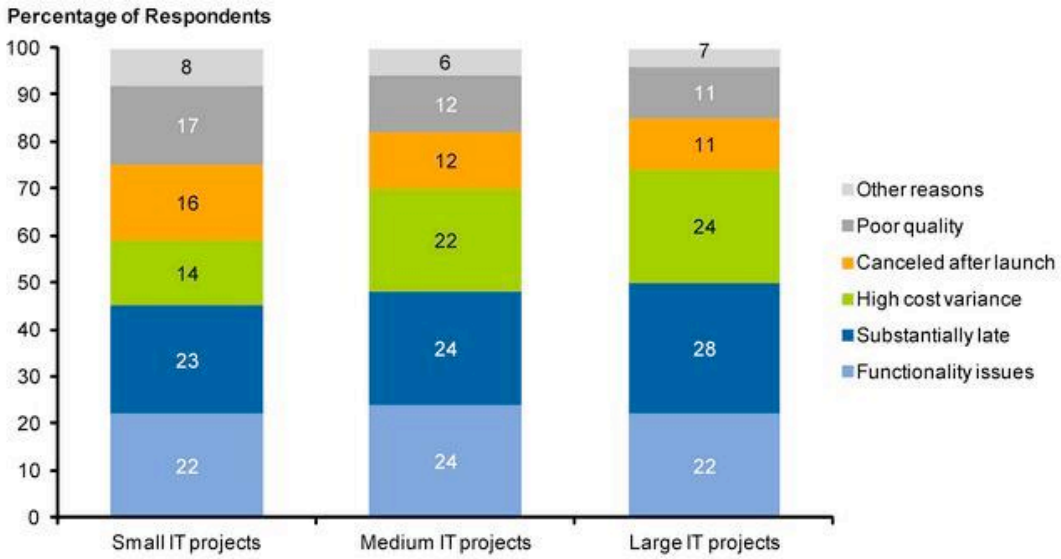
How many lines of code in…

- OS X
  - 86 million
    - http://www.engadget.com/2006/08/07/live-from-wwdc-2006-steve-jobs-keynote/
- Windows XP
  - 40 million
    - http://www.dwheeler.com/sloc/
- Open Office
  - > 10 million
    - http://www.ohloh.net/p/openoffice/analyses/latest
- Eclipse
  - > 10 million
    - http://www.ohloh.net/p/eclipse/analyses/latest

Size doesn't matter?

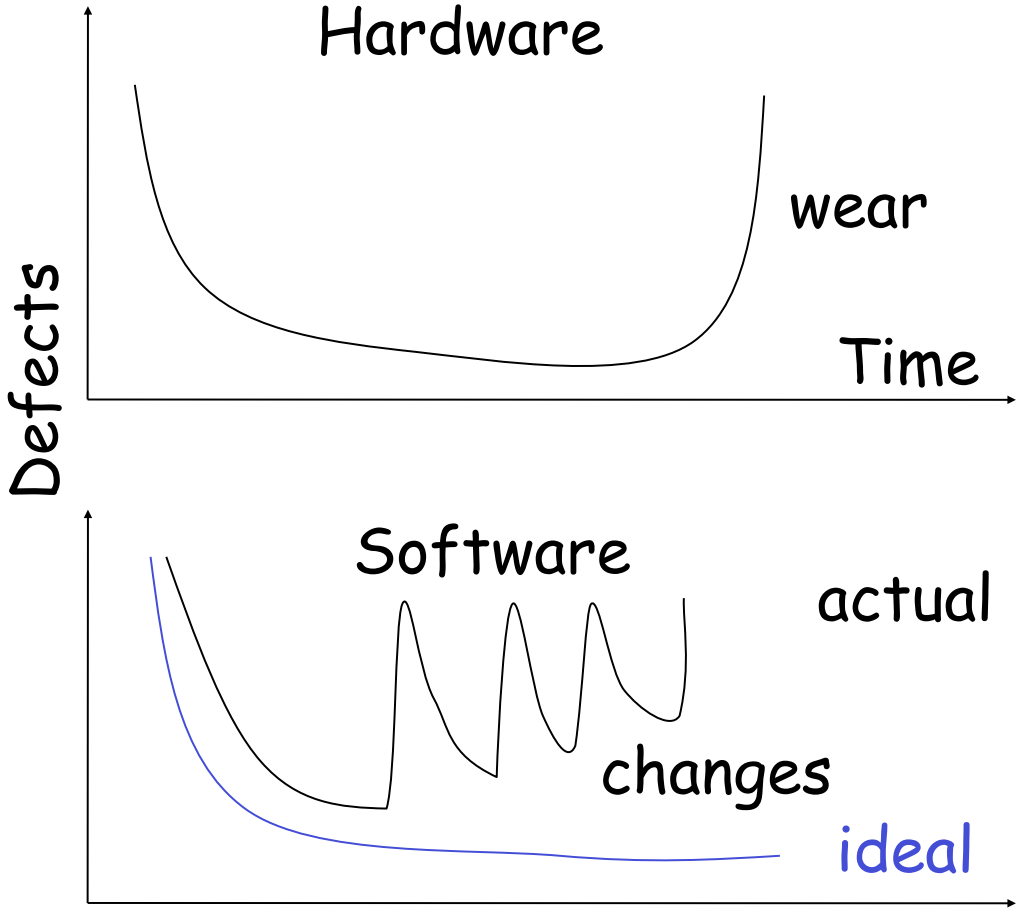
- 6 frequently mentioned reasons for project failure:
- Functionality issues
- Substantially late
- Quality issues
- High cost variance
- Canceled after launch
- Rejected or not implemented for other reasons



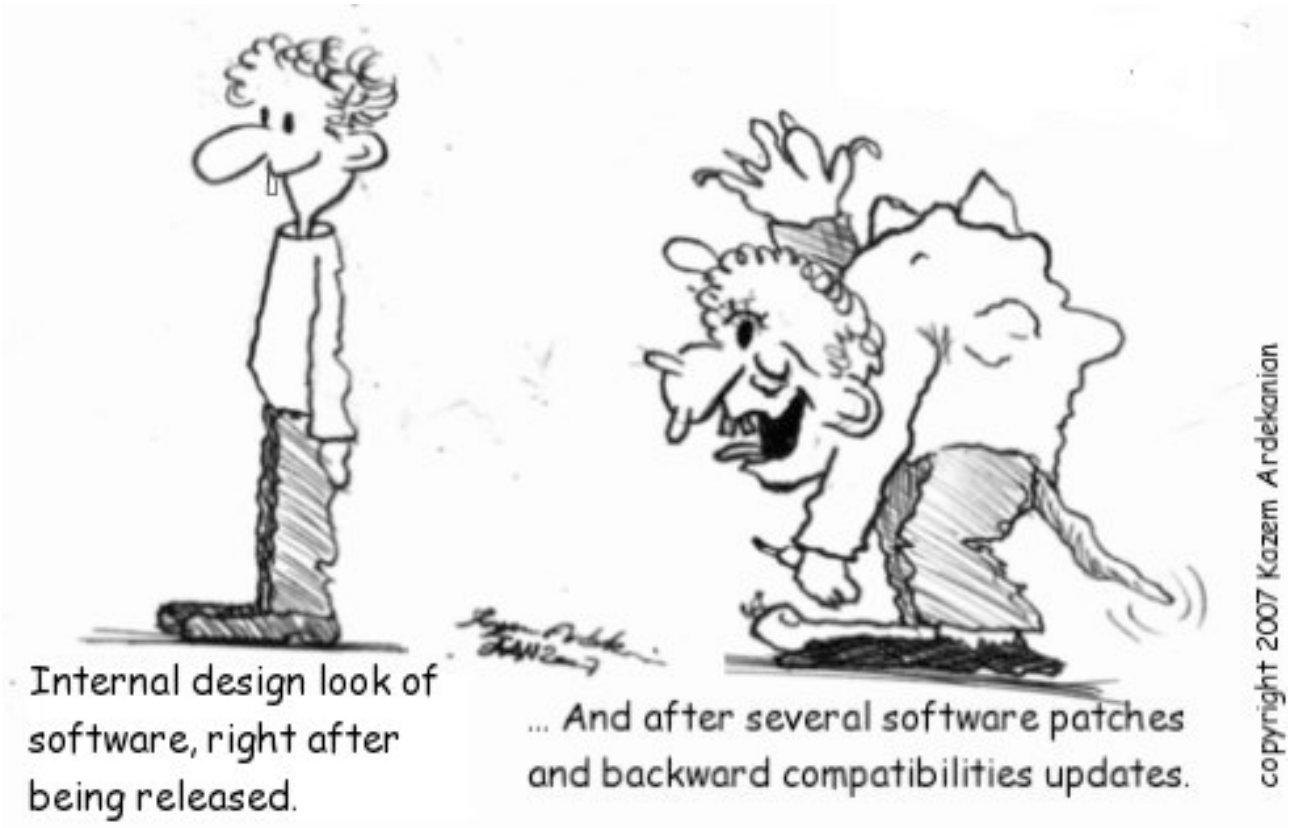


from http://thisiswhatgoodlookslike.com/2012/06/10/gartner-survey-shows-why-projects-fail/

Software vs. Hardware Reliability Curve



- Hardware wears out
- Software changes
  - or its environment changes
  - called "bit-rot"


Internal design look of software, right after being released.
... And after several software patches and backward compatibilities updates.
copyright 2007 Kazem Ardekanian

Software Engineering Myths: Management

- "We have books with rules. Isn't that everything my people need?"
  - Which book do you think is perfect for you?
- "If we fall behind, we add more programmers"
  - "Adding people to a late software project, makes it later" – Fred Brooks (The Mythical Man Month)
- "We can outsource it"
  - If you do not know how to manage and control it internally, you will struggle to do this with outsiders

Software Engineering Myths: Customer

- "We can refine the requirements later"
  - A recipe for disaster.
- "The good thing about software is that we can change it later easily"
  - As time passes, cost of changes grows rapidly

Software Engineering Myths: Practitioner

- "Let's write the code, so we'll be done faster"
  - "The sooner you begin writing code, the longer it'll take to finish"
  - 60-80% of effort is expended after first delivery
- "Until I finish it, I cannot assess its quality"
  - Software and design reviews are more effective than testing (find 5 times more bugs)
- "There is no time for software engineering"
  - But is there time to redo the software?

My List: What is Software Engineering For?

- We want to build a system
- How will we know the system works?
- How do we develop a system efficiently?
  - Minimize time
  - Minimize costs
  - Minimize …

Problem 1: How Do We Know It Works?

- Buggy software is a huge problem
  - But you likely already know that
- Defects in software are commonplace
  - Much more common than in other engineering disciplines
- Examples
  - (your exploration of recent IT disasters)
- This is not inevitable---we can do better!

What is It?

- But how do we know behavior is a bug?

Software Bug => Space Disaster

- Ariane 5 Space mission
- $7,000,000,000
- 10 Years in the making
- 40 seconds after take off the rocket exploded

Software Bug => Space Disaster

Smoke from the explosion
June 4, 1996 (AP Photo)

Attempt to cram a 64-floating point number to a 16-bit integer failed

AT&T long distance service failed for 9 hours

- On January 15, 1990, one of AT&T's #4ESS toll switching systems in New York City experienced an intermittent failure that caused a major service outage.
- Wrong BREAK statement in C-Code
- Complete coverage could have revealed this bug during testing

```
/* ``C'' Fragment to Illustrate AT&T
   Defect */


do {
   switch expression {
         ...
         case (value):
                  if (logical) {
                       some statements
                       break;
                  }
                  else
                  {
                        some statements
                  }
                  more statements
         case (value2):
                  ...
}
...
```


Reference: ACM SIGSOFT, Software Engineering Notes, Vol. 15, No. 2, Page 11ff, April 1990

Software Bugs: Cause of Deaths

- Several deaths of cancer patients were due to overdoses of radiation resulting from a race condition between concurrent tasks in the Therac-25 software.

180 Degree Bug

Torpedoes, that deviate more than 90 degrees, explode to avoid self destruction of the ship.

Once upon a time a ship fired a torpedo but the torpedo was jammed in the tube. Then the captain gave the command: Let's turn around and return to the harbor!

... No, they did not live happily ever after.

What is It?

- But how do we know behavior is a bug?

- Because we have some separate specification of what the program must do
  - Separate from the code

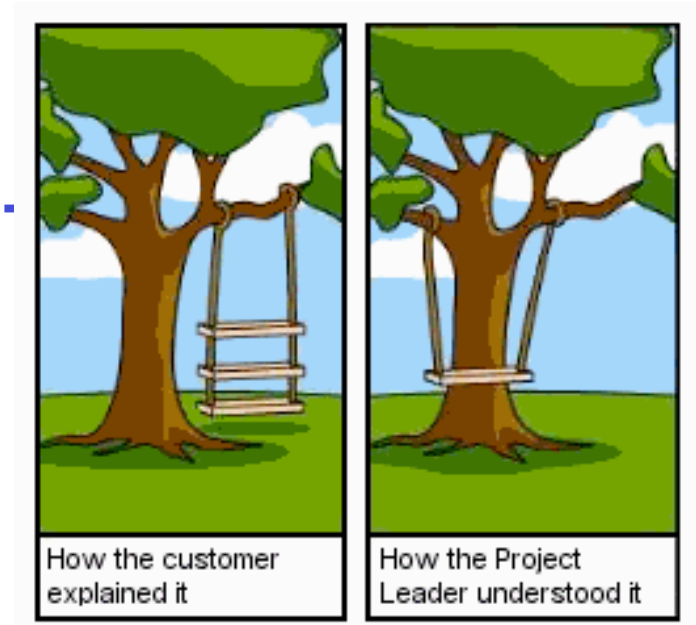
- Thus, knowing whether the code works requires us first to define what "works" means
  - A specification

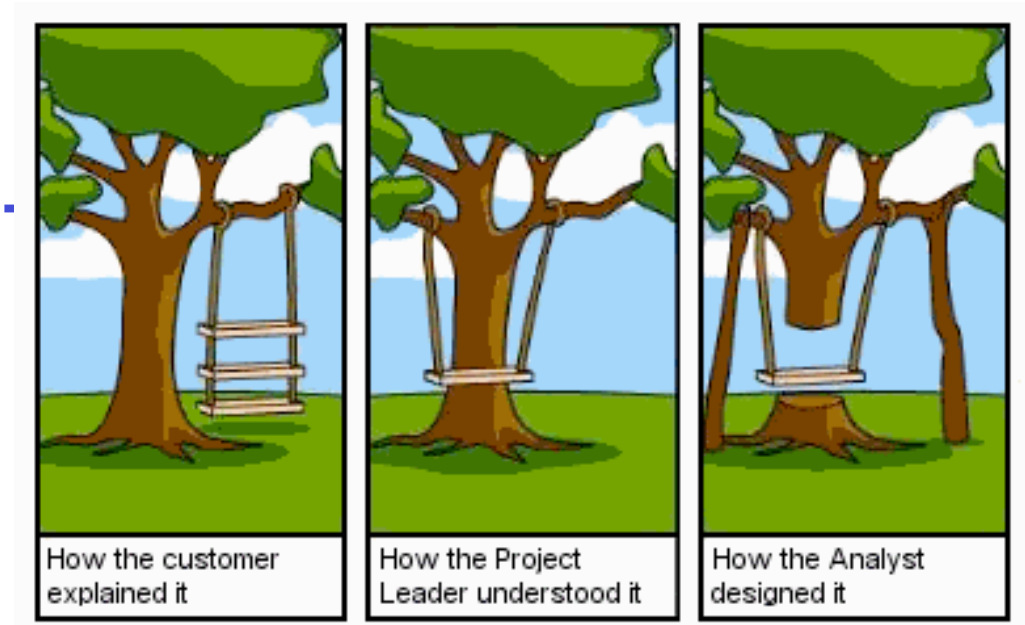
Teams and Specifications

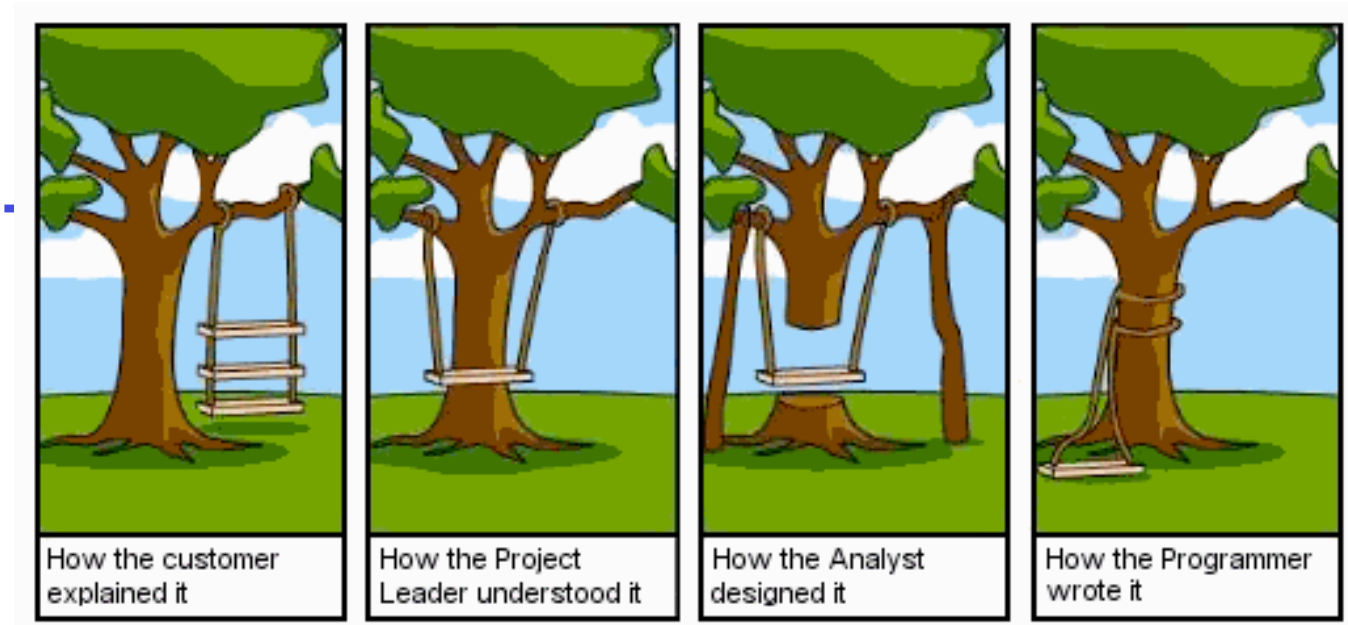
- Do we really need to write specifications?

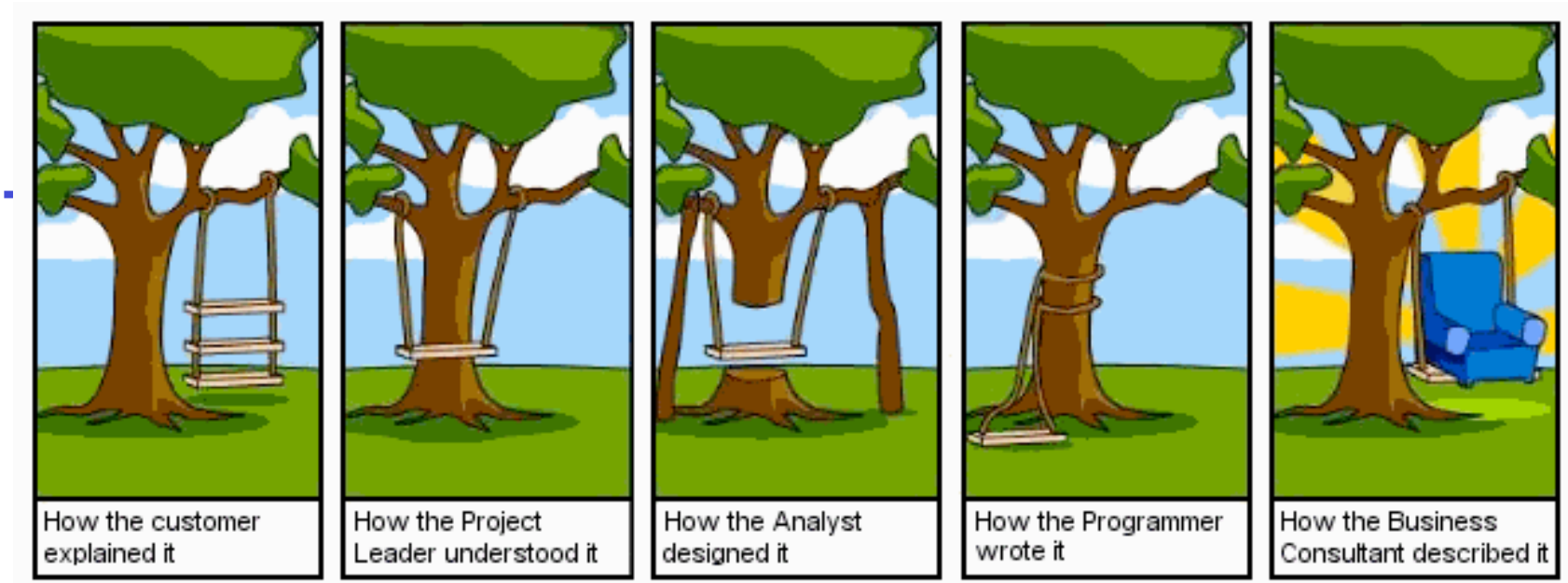
- People will
  - Discuss what to do
  - Divide up the work
  - Implement incompatible components
  - Be surprised when it doesn't all just work together

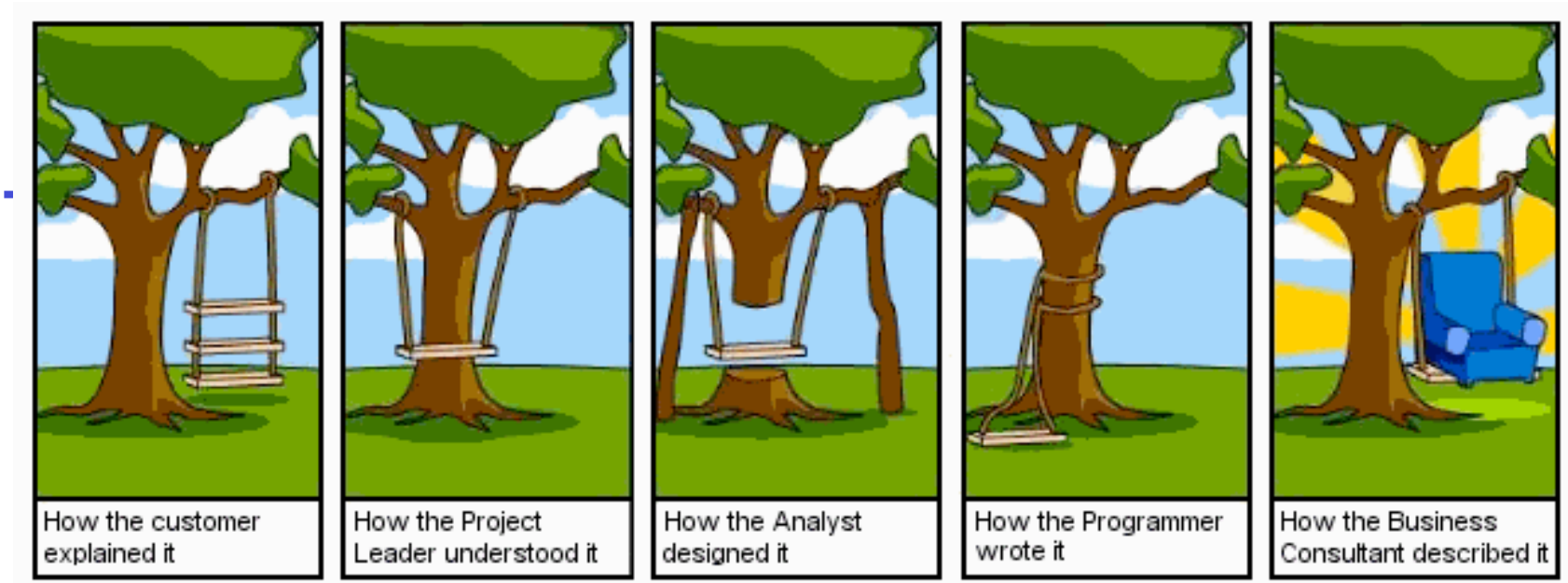
How the customer explained it

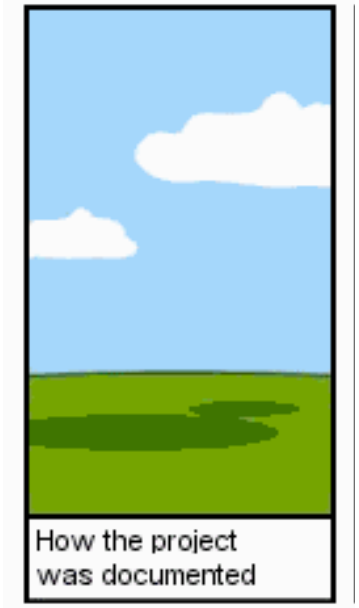

How the customer explained it
How the Project Leader understood it

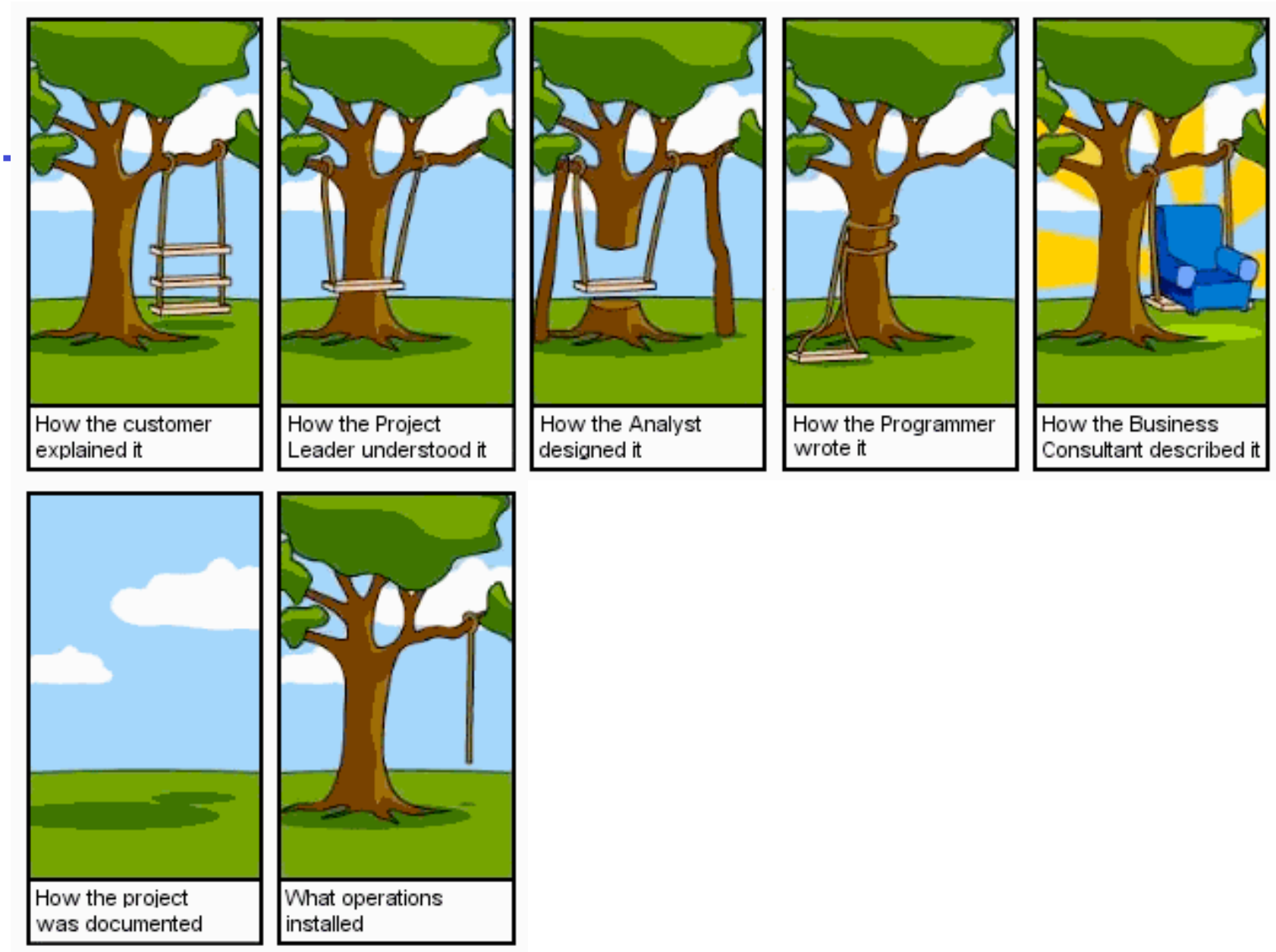

How the customer explained it
How the Project Leader understood it
How the Analyst designed it

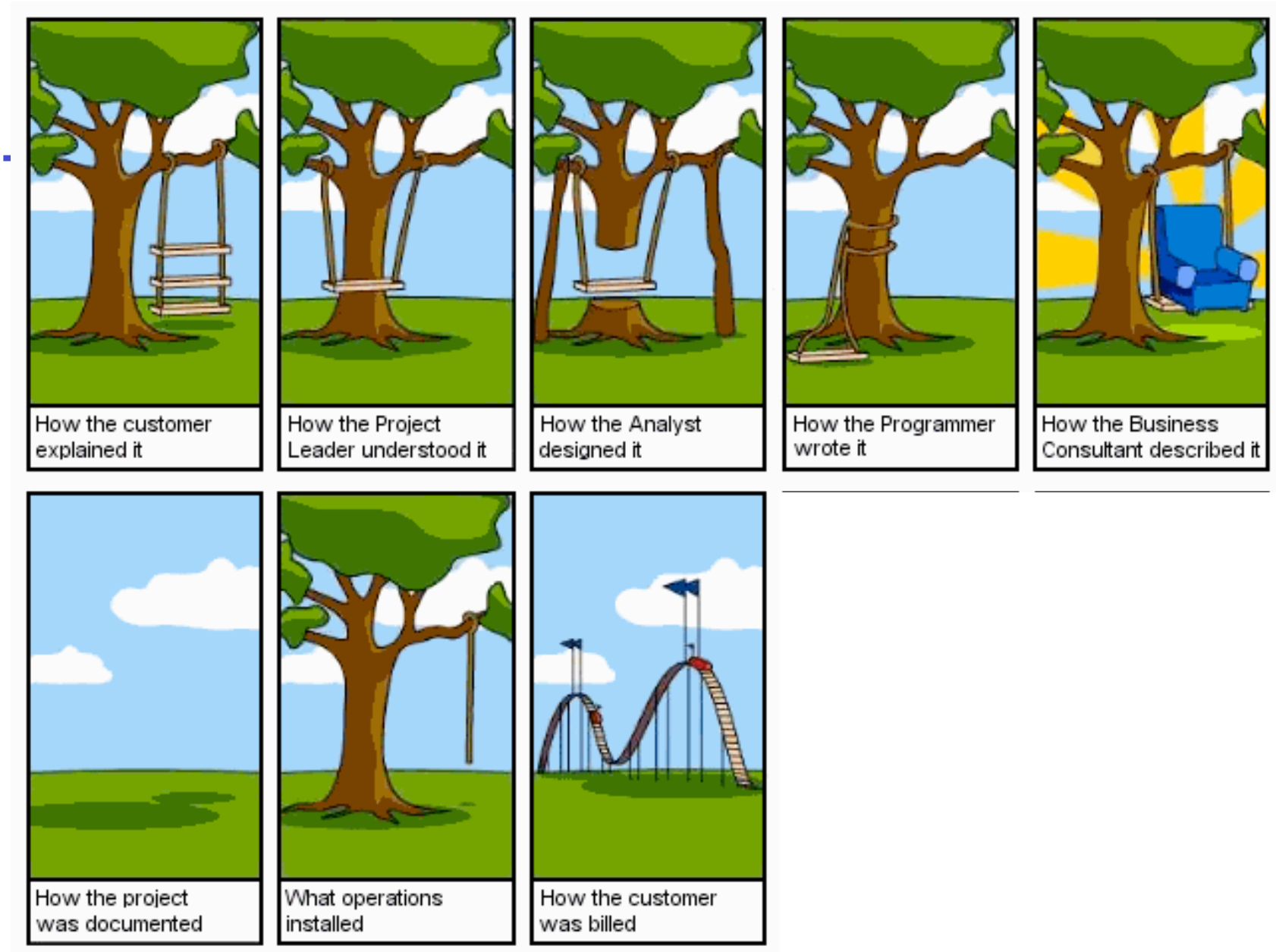

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it

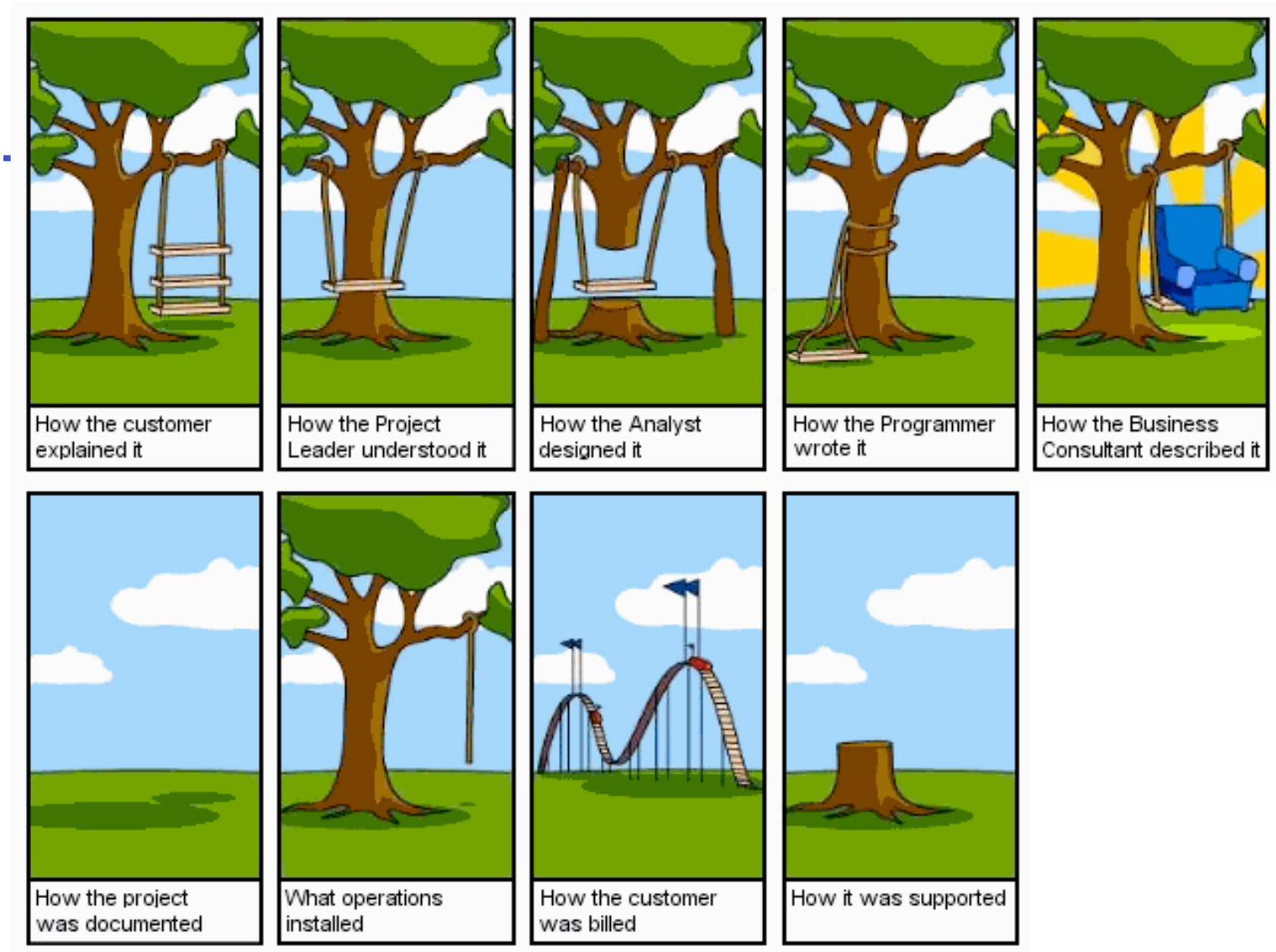

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it

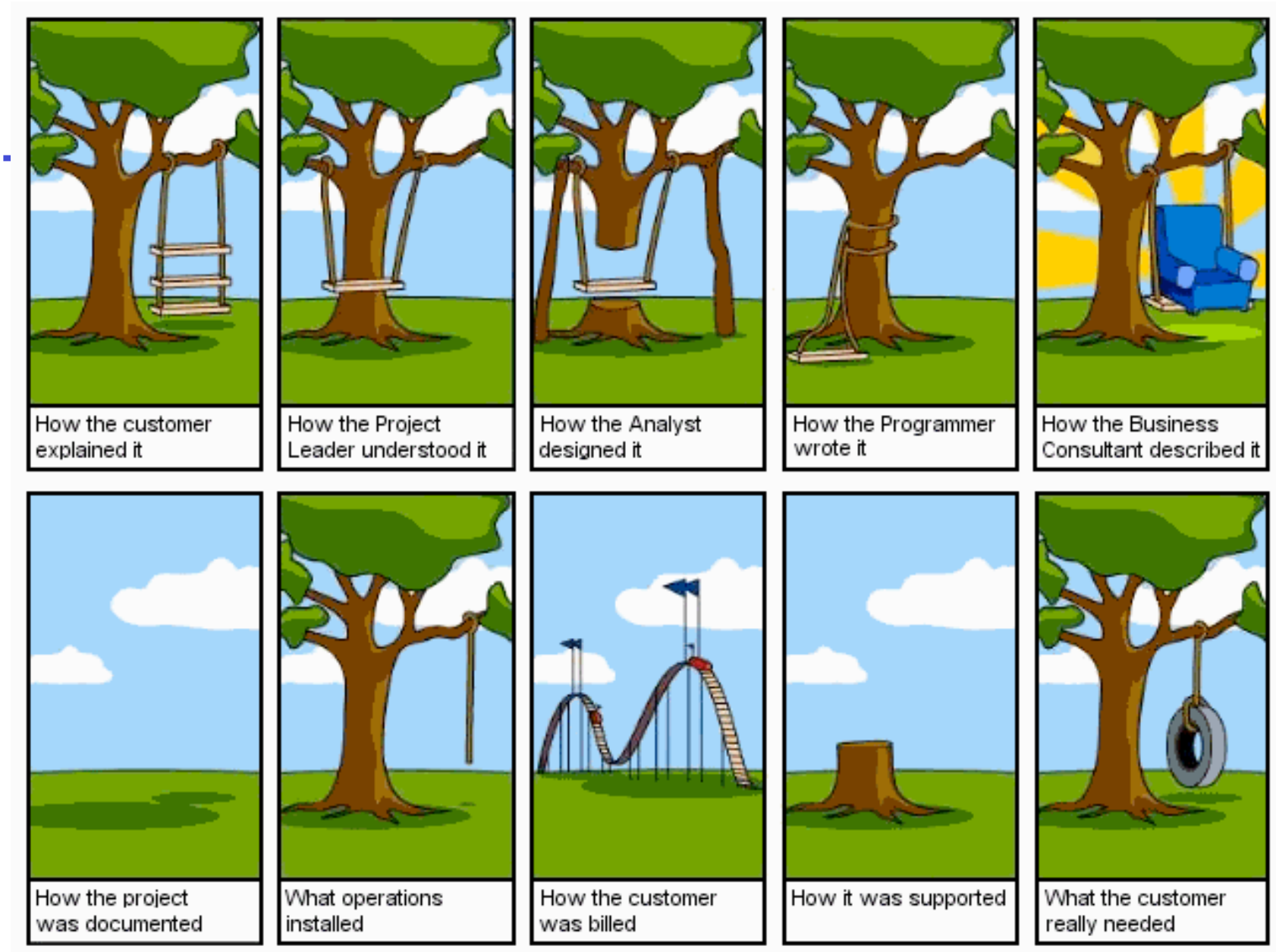

How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it


How the project was documented


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant 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 Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

What Can We Do?

- Write specifications
  - Write down what it is supposed to do
  - Make sure everyone understands it
  - Keep the specification up to date

- This does not solve the problem completely
  - There are always ambiguities, contradictions
  - These lead to bugs
  - But the problem is reduced to manageable size

Summary of Problem #1

- A specification allows us to:
  - Check whether software works
  - Build software in teams at all
- Actually checking that software works is hard
  - Code reviews
  - Static analysis tools
  - Testing and more testing
  - We will examine this problem closely

Problem #2: How Do We Code Efficiently?

- Assume we want to minimize time
  - Usually the case
  - Time-to-market exerts great pressure in software
- How can we code faster?
  - Obvious answer: Hire more programmers!

Parallel Development

- How many programmers can we keep busy?
  - As many as there are independent tasks
- People can work on different modules
  - Thus we get parallelism
  - And save time
- What are the pitfalls?

Pitfalls of Parallel Development

- The problems are the same as in parallel computing
- More people = more communication
  - Which is hard
- Individual tasks must not be too fine-grain
  - Increases communication overhead further
- Inherent sequential constraints
  - E.g., pipeline architecture

Interfaces

- The chunks of work must be independent
  - But work together in the final system
- We need interfaces between the components
  - To isolate them from one another
  - To ensure that the final system works
- The interfaces must not change (much)!

Defining Interfaces

- What are interfaces?
- They are just specifications!
- But of a special kind
  - Interfaces are the boundaries between components
    - And people

Defining Interfaces

- Specifying interfaces is most important
  - Interfaces should not change a lot
  - Effort must be spent ensuring everyone understands the interfaces
  - Both things require preplanning and time
- But often we can stop at specifying interfaces
  - Let individual programmers handle the internals themselves

Software Architecture

- To define interfaces, we must decompose a system into separate pieces with boundaries
- How do we do this?
- Your thoughts

My Opinions

The decomposition of a system is driven by:

- What it does
- How we build it
- Who builds it

Decomposition: What the System Does

- The application itself often dictates natural decomposition

- A compiler is a pipeline consisting of
  - Lexer
  - Parser
  - Type checker
  - Optimizer
  - Etc.

Decomposition: How We Build It

- Buildings need scaffolding during construction
- So does software!

- Two areas in particular:
  - Lots of extra code that is not really part of the final product
  - Influence of third-party subsystems

- Test harnesses, stubs, ways of building and running partial systems

Decomposition: Who Builds It

- Software architecture reflects the structure of the organization that builds it

- Often, 5 developers = 5 components

Summary of Problem #2

- Efficient development requires
  - Decomposing system into pieces
  - Good interfaces between pieces
- The pieces should be large
  - Don't try to break up into too many pieces
- Interfaces are specifications of boundaries
  - Must be well thought-out and well communicated

Conclusions

- Software engineering boils down to several issues:
  - Specification: Know what you want to do
  - Design: Develop an efficient plan for doing it
  - Programming: Do it
  - Validation: Check that you have got what you wanted
- Specifications are important
  - To even define what you want to do
  - To ensure everyone understands the plan

Conclusions (Cont.)

- Is that all?
- NO!
- Why?
  - Because specifications do change!
  - Because you were wrong about what you wanted
  - Because the world changes
  - We'll talk about this next time . . .

Caveats in This Course

- CS Professors usually good at well-defined technical problems
- May not be great at ill-defined non-technical problems
- Take everything in this class with a pinch of salt
  - Ultimately, the most important things you learn are those you learn through experience

Acknowledgements

- Many slides courtesy of Rupak Majumdar