

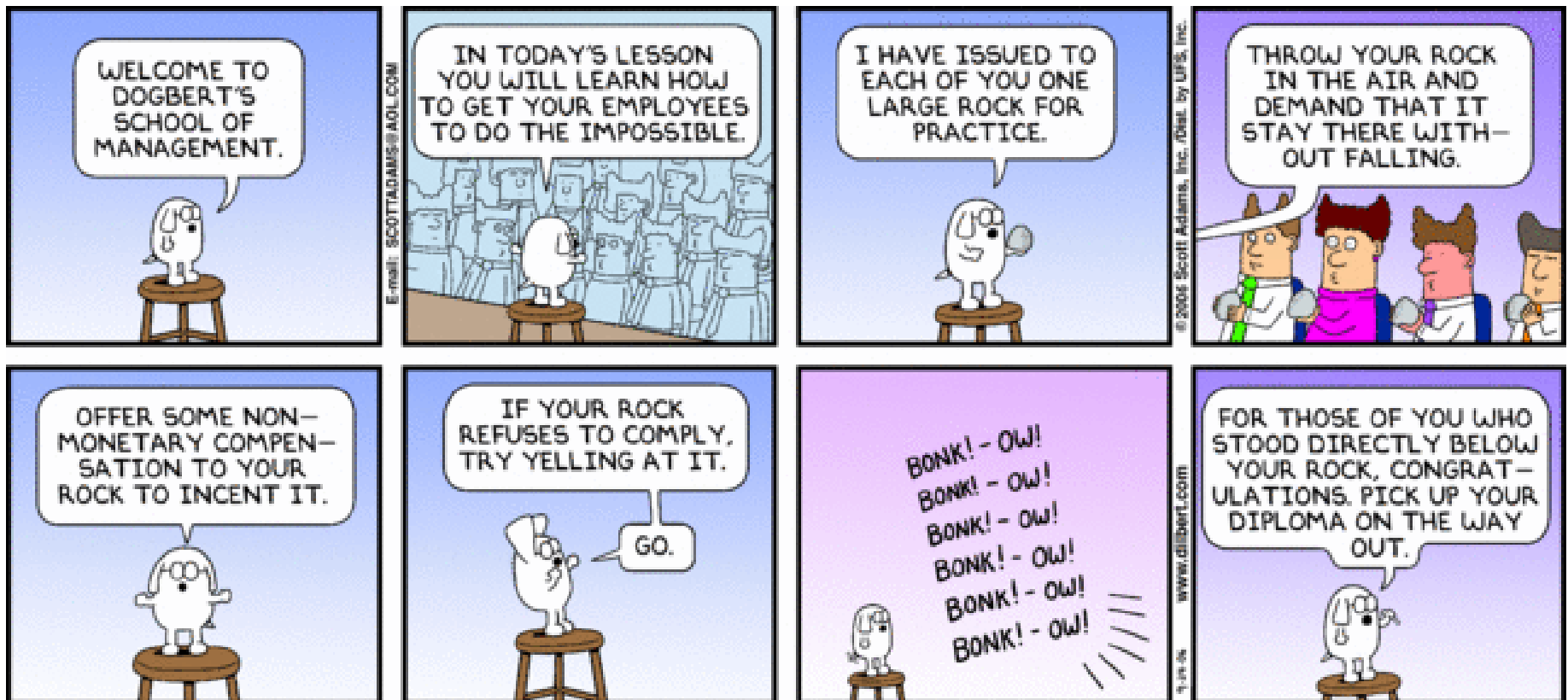
Before We Get Started: A Motivating Overview

CSE 4322, Fall 2013
Christoph Csallner
University of Texas at Arlington (UTA)

Some of these slides are by:

- Yannis Smaragdakis, University of Athens (Greece)
- Eknauth Persaud, Ayoka

Great Source: <http://dilbert.com/>



What Should a Software Project Manager Do?

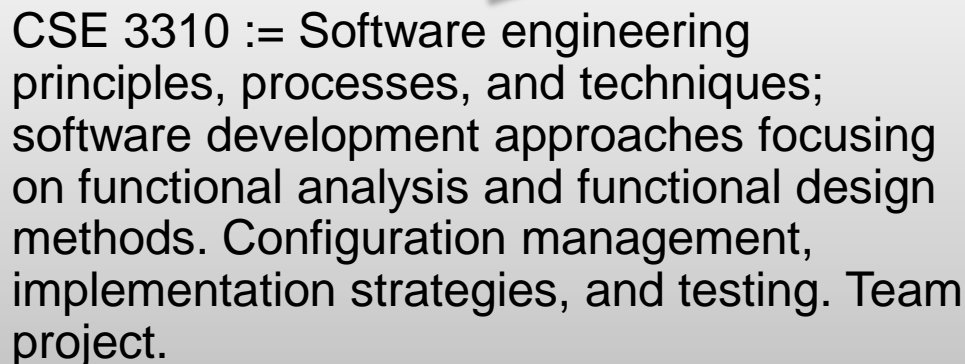


- Joel on Software (Joel Spolsky):
“Your first priority as the manager of a software team is building the development abstraction layer.”
<http://www.joelonsoftware.com/articles/DevelopmentAbstraction.html>
- Provide abstract layer on which engineers can work
 - Lets your engineers focus on what they are good at
- Protect engineers from non-development chores
 - Network, backup, power supply, air conditioning
 - Provide quiet working environment
 - Protect from interruptions, unimportant meetings

CSE 4322 UTA Course Catalog

CSE 4322:

- Introduction to **software project management**.
- Issues include
 - *See next slide*
- Prerequisite: CSE 3310.



CSE 3310 := Software engineering principles, processes, and techniques; software development approaches focusing on functional analysis and functional design methods. Configuration management, implementation strategies, and testing. Team project.

Prerequisite: CSE 1325 and CSE 2315.

CSE 1325 := OO Programming

CSE 2315 := Discrete Structures

Main SE Tasks & Artifacts

- What is the problem to be solved?
- How should we solve the problem?
- Actually solve the problem

Main SE Tasks & Artifacts

- What is the problem to be solved?
 - Describe **domain**
 - Describe **requirements**
- How should we solve the problem?
 -
 -
- Actually solve the problem

Main SE Tasks & Artifacts

- What is the problem to be solved?
 - Describe **domain**
 - Describe **requirements**
- How should we solve the problem?
 - High-level: **architecture**
 - Low-level: **design**
- Actually solve the problem
 - Write **code**

Main SE Tasks & Artifacts

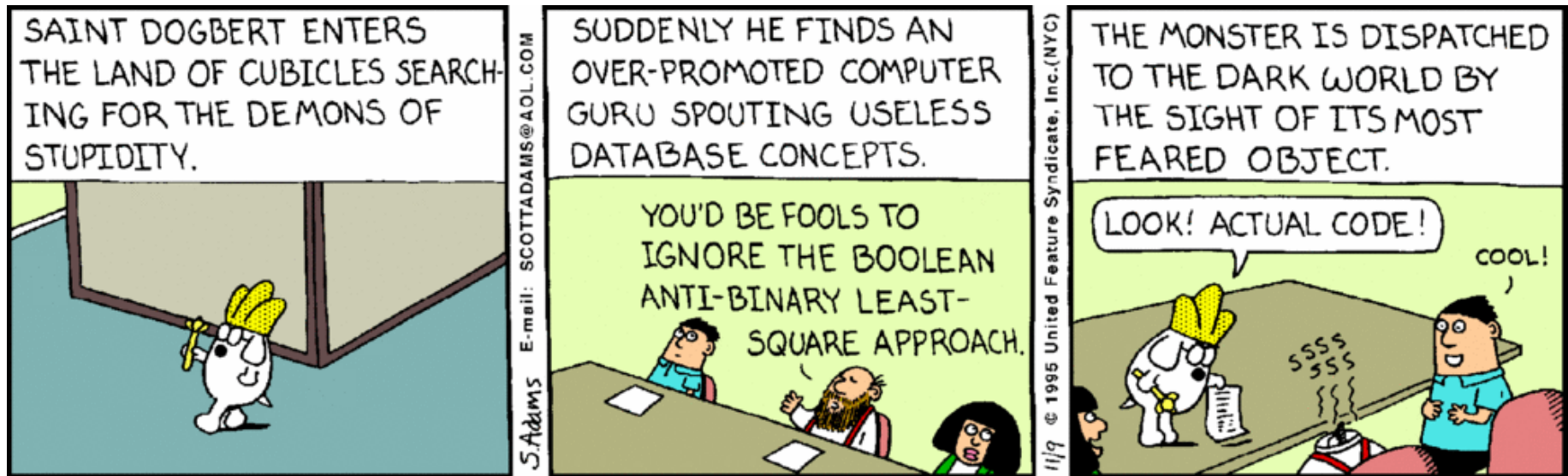
- What is the problem to be solved?
 - Describe **domain**
 - Describe **requirements**
- How should we solve the problem?
 - High-level: **architecture**
 - Low-level: **design**
- Actually solve the problem
 - Write **code**
- Check deliverables
 - **Review, analysis, test**

But how to do this exactly?

- Describe domain How?
- Describe requirements How?
- High-level: architecture How?
- Low-level: design How?
- Write code How?
- Review, analysis, test How?
- **How to move between steps? Who does what?**
 - Process, management

Academic Software Engineering vs. Software Engineering in Practice

- What problems do actual software engineers have?
- Will academic solutions (eventually) help actual software engineers in solving their actual problems?



What Do People in the Trenches Think? (about process, SE practices, etc.)

- Yannis asked people who write software for a living
 - in a group environment, large software company
- No shortage of opinions!
 - their inputs will come up a lot in what we will do
 - mostly unedited quotes
- Yannis worked with two of these people before
- For emphasis, I made some words **bold**



Setup of a Real-World Project

- How many full-time developers do you think there are in a project that produces software you use?
- What is the ratio of developers-testers-managers?
- How much of the total effort is spent coding?
- How much of their time do developers code?

Rough Breakdown

- *7 person group, long term project:*
 - 3 devs
 - 3 testers
 - 1 PM
- Scale up by multiplying or grouping groups to form larger ones
- *Devs spend maybe 40% of their time coding, so 15-20% of total team effort is on 'coding'*

Being a Tester vs. Being a Programmer (Developer, Engineer)



Always Tension Between Coding and Rest of Tasks

- “Even if coding is 20% of total development time, it is the only activity in SE that you can’t skip!”
 - (almost direct quote from “Code Complete”)
 - others will argue you can’t skip anything
 - There are rare exceptions, where customer only wants specs
- Hardliner view:
 - *All the other stuff is just to compensate for human weaknesses at coding*
 - *3 dev + 3 test + 1 PM in my project in the past 18 months would be about as productive as 2 dev*

Coding vs. “Overhead”



More Quotes:

Preliminaries, “Minimum Bar”

- *No matter what size project you're on, you must have a source control system, a bug tracking system, unit tests that are easy to run, and a rolling build system that ensures checkins don't break the build. If any of those are missing, then just fold up the tent and go home.*
- *Get your dev tools, source control, build processes, debugger and profiler setup, automated test tools, basic performance tests, etc. in place at the beginning.*

More Quotes:

Preliminaries, “Minimum Bar”

- *No matter what size project you're on, you must have a **source control system**, a **bug tracking system**, **unit tests** that are easy to run, and a rolling **build system** that ensures checkins don't break the build. If any of those are missing, then just fold up the tent and go home.*
- *Get your dev tools, source control, build processes, **debugger** and **profiler** setup, **automated test tools**, basic performance tests, etc. in place at the beginning.*
- **Tools** are important in practice
 - **Tool demonstrations, anyone?**

Scaffolding

- ***No one invests enough in good build systems and tools.*** *It surprises me at [company] how much effort we put into the product itself, but then the build system that actually produces the product is always duct tape and bailing wire. In my limited experience, it's a universal nightmare. There's always an awful mix of makefiles and perl scripts and [proprietary tool] and all kinds of crazy unreliable crap, and if you're like me and actually care about this part of the process and how it is sucking away everyone's productivity, then it can drive you nuts.*

Conventions

- *Have an "architectural checklist" for key issues such as concurrency, re-entrancy, fault-tolerance (that might be too advanced), asynchronous I/O that may block indefinitely (much more realistic), handling of bad input (including weird cases like an object created by the wrong instance of the factory), low-memory conditions, etc..*

Also decide on key conventions such as (in unmanaged code) caller or callee initializing output parameters, silly things like parameter order (outputs last?), etc. I've found that in our code for [latest project], there's an amazing amount of boiler-plate for this sort of stuff (including the architectural aspects)

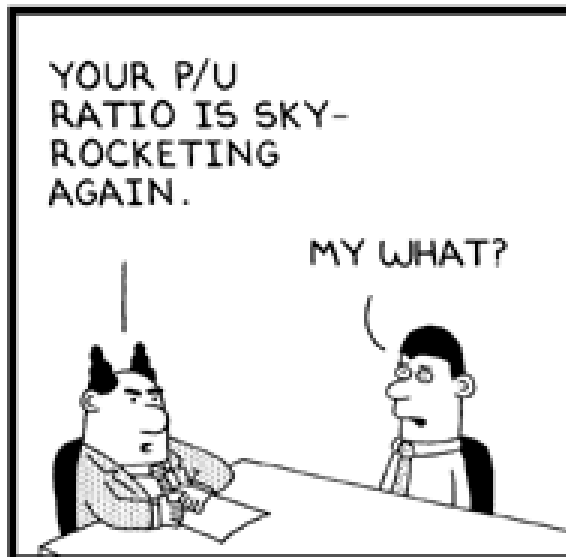
Process

- *If I were in charge of a team, I would emphasize rapid prototyping. The less "adventurous" a project is, the less this is needed, but unless there's a **really complete spec**, I'd say a prototype pays for itself very quickly. One can also use the prototype to build up an initial test suite, although it's probably best to keep that away from the devs when they start the real implementation, or else they'll just code against it.*
- *"Scrum" is a very popular method lately; I've only had a taste of it, but short milestones, good prioritization, and high-touch interactions among team members seem like good stuff for small teams.*
- Scrum is an iterative process model

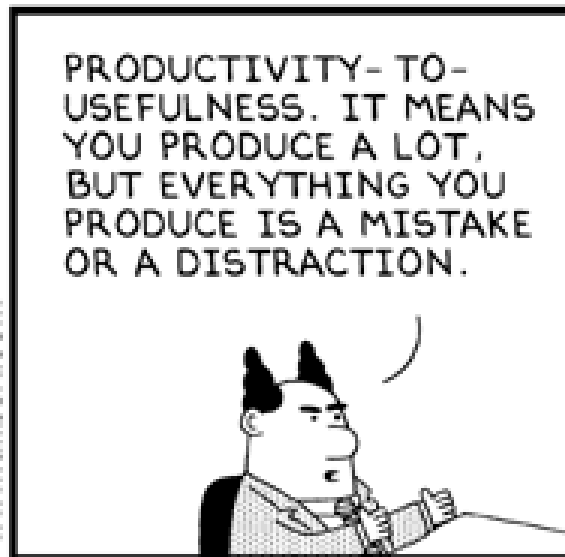
Prioritizing tasks in Software Development

- *I've noticed that when people are given some procedure to follow (e.g. fill out a "security threat model" form), they're happy to do that, as they'll be able to make concrete progress for that hour or whatever of their day. It's very easy to fill up the hours of an employee's day with this sort of stuff, and it will get prioritized, as it's far easier for a manager to say "you didn't fill in form X", than to say "you're coding too slowly".*

Control productivity and usefulness



www.dilbert.com
scottadams@aol.com



10-13-04 ©2004 Scott Adams, Inc./Dist. by UFS, Inc.



© UFS, Inc.



www.dilbert.com
scottadams@aol.com



11-29-07 ©2007 Scott Adams, Inc./Dist. by UFS, Inc.



Process and Project Size

- *Large teams working on large code bases with more complexity have got to have a lot of 'process' in place. **Good large teams will choose the minimal processes for the maximum benefit.** But in the end, if you are personally not big on process, you will probably be happier on a smaller team/product.*

Process and Design

- *We definitely needed more time for design up front. We had plenty of time (months!), but squandered it in "working groups" that met many times and agonized over some basic points. Then we were distracted by another project management dumped on us, and then we were given a week to fill in the remaining 95% of the details.*

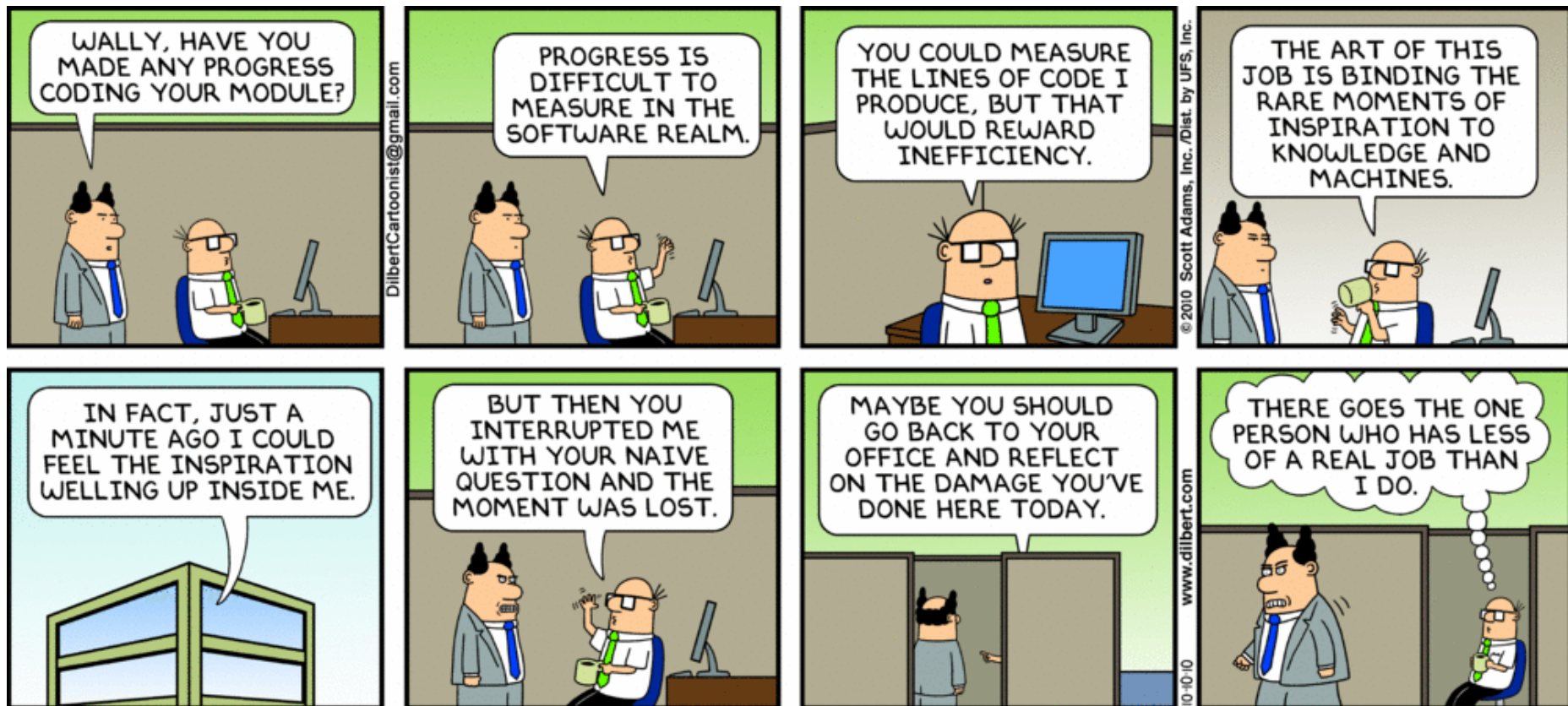
Process and Design

- *A friend of mine is on another team at [company] that is having a "quality crisis"; I guess in their last milestones they had a regression rate of 30% per checkin (I have no idea about industry stats overall, but that doesn't seem too extremely high to me; maybe 15% is more reasonable?) and so the "solution" is that one of their architects (whom my friend thinks is a useless idiot) is trying to impose a bunch of metrics (e.g. cyclomatic complexity, comment percentage per lines of code, ...) which will be enforced on checkins to improve the code base. My friend sent me a copy of the doc this guy sent, it was basically phrased as "to fix our problems, we'll do this" with metrics apparently picked out of the air with no sound justification and no plans to evaluate if the new process is working.*

Metrics

- *A friend of mine is on another team at [company] that is having a "quality crisis"; I guess in their last milestones they had a regression rate of 30% per checkin (I have no idea about industry stats overall, but that doesn't seem too extremely high to me; maybe 15% is more reasonable?) and so the "solution" is that one of their architects (whom my friend thinks is a useless idiot) is trying to **impose a bunch of metrics** (e.g. cyclomatic complexity, comment percentage per lines of code, ...) which will be enforced on checkins to improve the code base. My friend sent me a copy of the doc this guy sent, it was basically phrased as "to fix our problems, we'll do this" with metrics apparently picked out of the air with no sound justification and no plans to evaluate if the new process is working.*
- Metrics are often not a good management technique
 - Most metrics can and will be circumvented

Progress is difficult to measure in the software realm



Software developers are smart: Most software development metrics can and will be circumvented



Process and Testing

- *One process I really like is test-driven development. I wrote a blog entry about it a while back [...]*
*It also mentions "code coverage" as a useful metric; it is the only metric I know that is nearly universal; most teams at [company] seem to measure it, with the idea that **if you've got less than ~80% block coverage from your tests, then you are doing something wrong.***

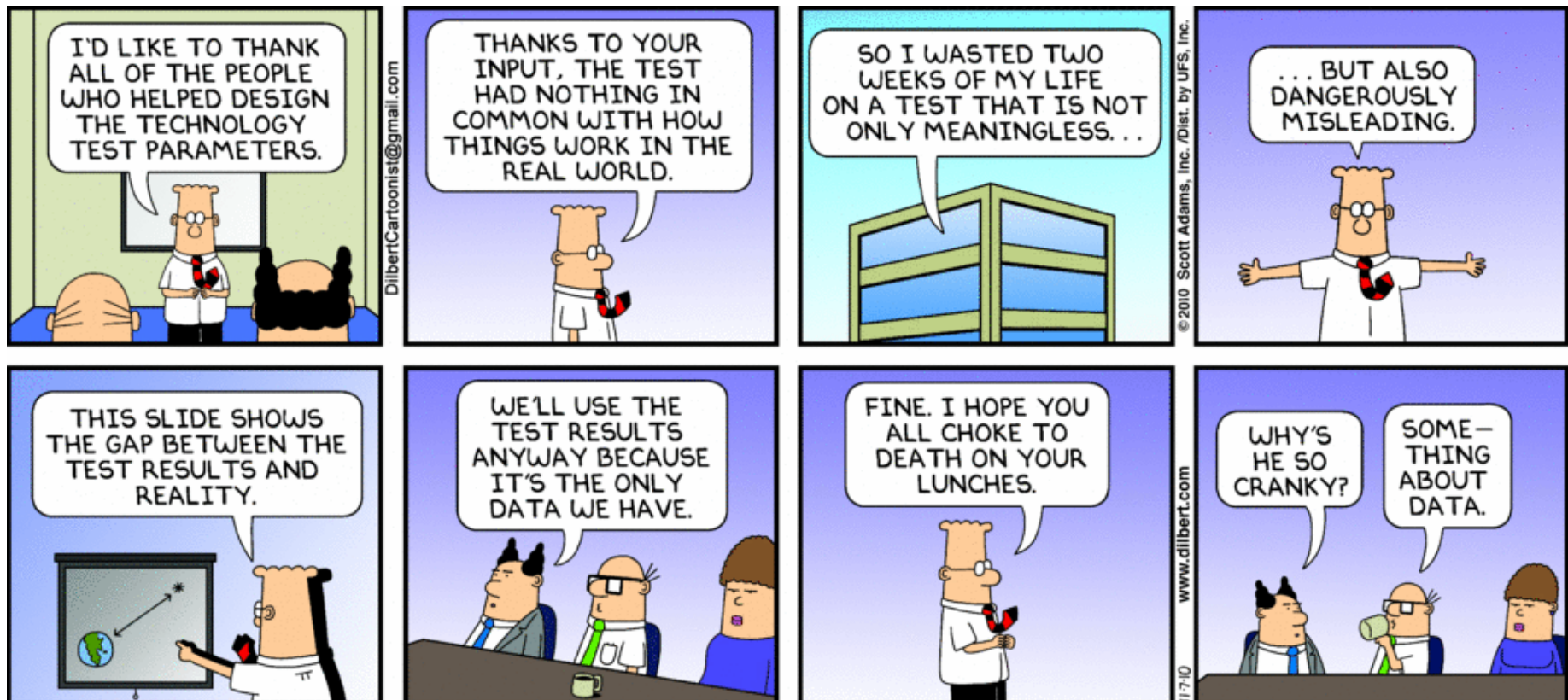
Process and Testing

- *[T]he best approach to software engineering is to find a way of "doing everything twice", sorta like double ledger bookkeeping. In some cases, one relies on the type system. In others, on asserts. In others, on unit tests. **I'd say this is a key thing to focus on -- is there some way the machine can check absolutely every aspect of a program against another aspect?** Check as much as possible as early as possible, using compile-time asserts and complex types that may compile to almost nothing. (I found a ton of bugs a couple of months ago when I changed a typedef that was used in subtly different ways into a class template with four incompatible instantiations that required explicit conversions between them.) **Have a build that runs the automated tests extremely slowly, but does massive run-time consistency checks on all data structures.***

Process and Testing

- [This is the same person who said $3\text{dev} + 3\text{test} + 1\text{PM} = 2\text{dev}$]
*One problem we've had with separate testers is that there's zero accountability for them. They can just say "it's tested" and everyone (well, management) believes them. I suggested two potential levels of accountability... The first is to allow devs to insert at key points in the code some sort of macro/whatever to say "this codepath should be tested". ... [T]ests should be required to hit *all* of them. Our testers rely on code coverage numbers, and because of various little error-handling macros and the like, numbers like 70% are considered acceptable. Alas, no-one knows if major cases are being missed. The even higher level of accountability would be to allow devs to add macros that introduce deliberate bugs in a special build.*
- [different person, not responding to previous]
Trust me, I speak from experience. My testers are finding plenty of bugs.

Testing the Product Aspects That Will Matter in Practice



Similar stories from other companies

FOR EXAMPLE: INTUIT

Automation is key

- Jon Burt of Intuit on managing the 10 million LOC of the QuickBooks small-business accounting system
- “Most of all, the key to managing a large project was automation. ‘We automate everything that can be automated,’ says Burt. ‘The tools make a huge difference. We maintain all the different versions of QuickBooks, on all our supported platforms, with about 60 code-writing developers. We couldn't do that without automation.’”
 - <http://www.drdobbs.com/tools/building-quickbooks-how-intuit-manages-1/240003694?pgno=2>
 - <http://news.ycombinator.com/item?id=4315578>

From Ayoka CEO Eknauth Persaud during 2011 UTA CSE
Industrial Advisory Board meeting (used with permission)

CLOSER TO HOME

Ayoka



- Located in Arlington, TX
- “made in U.S.A. software services”
- In the last 5 years hired 40+ UTA alumni
 - Computer Science
 - Business, Management, Marketing
- Application development
 - Java, .Net, C#, Silverlight
 - Ruby on Rails, Python, Php
 - Mobile Apps – Apple, Android, Blackberry
- Enterprise systems, systems integration, ...

UTA: Opportunities for Improvement **[verbatim from CEO's slides]**

- *Coursework*
 - *Teach modern languages, inspect code*
 - *Learn web architecture, read + write to databases*
 - *Understand differences of embedded vs. web*
 - *Teamwork: code repository, commenting, etc.*
 - *Model class projects to real-world*
- *Behavior*
 - *Work ethic – ability to deal with pressure*
 - *Timeliness, dress properly, social grace, ethics*
 - *Business etiquette / adult behavior*

Today's Software Engineering ≠ Other Engineering

Software vs. Houses

Quote from online discussion:

- I've seen a large number of houses that are just shacks. Houses that can, and often do, simply collapse in heavy weather. They don't use windows and the door is a sheet of plywood.
- To get away with building a shack instead of a house, you have to build your shack where there are no housing codes or standards. Or, you have to evade inspection by claiming you don't "live" there. Or you have to be heavily armed so that the inspectors don't bother you.
- Also relevant is that many of us live in countries where **housing is strictly regulated**, and poor quality is considered unacceptable even if both builder and purchaser want to cut corners. In the **absence of that enforced regulation**, housing runs the whole range including ghastly deathtraps; that is what we see in the field of software.

Software vs. Houses (continued)

Quote from online discussion:

- Most consumer products have warranties (expressed or implied). Some even have applicable standards for safety.
- Consumer shrink-wrapped software specifically avoids offering a warranty of any kind. It's a sad, shabby business. To avoid warranty claims, they don't let you purchase software; you merely license it, or purchase a right to use. Read your EULA (End-User License Agreement). There's no warranty. Quality doesn't matter.
- In-house software, developed by large IT organizations, has no warranty of any kind either. It's entirely based on corporate politics, internal reputation and influence.

Software Development is Hard

Quote from online discussion:

- It used to be considered a truism that when IBM wrote OS/360 it was, *at that point [1960s]*, the most logically complex system ever developed by humans.
- Since then we've developed techniques of handling more and more complex systems. Our languages, APIs and tools have added layers of conceptualization and abstraction not dreamed of when OS/360 was hand-cranked together. The trouble is the complexity of what we are trying to achieve has increased in step - or maybe even a little more than.
- [T]he modern world practically runs on the software developed over the past 20 - 30 years and while there's lots of room for improvement we're not doing too badly - all in all.
- Software development is in its infancy. While engineers have had hundreds, even thousands of years to perfect their craft, software engineers have had a handful of decades.

Fewer Parts = Fewer Things Can Fail

Quote from online discussion:

- One major reason is that for the most part, software "engineers" aren't really trained as engineers. One of the most important principles in engineering is to keep designs as simple as possible in order to maximize reliability (fewer parts = fewer things that can fail).
- Most software developers that I've worked with over the years are not just unaware of the KISS principle, but also actively committed to making their software as complicated as possible. Programmers by their nature enjoy working with complexity, so much so that they tend to add it if it isn't there already. This leads to buggy software.

Today's Software Engineering ≠ Other Engineering

Will it always be like this?