Intro to Software Testing

chapter 4

Agile Testing

Dr. Brittany Johnson-Matthews
(Dr. B for short)

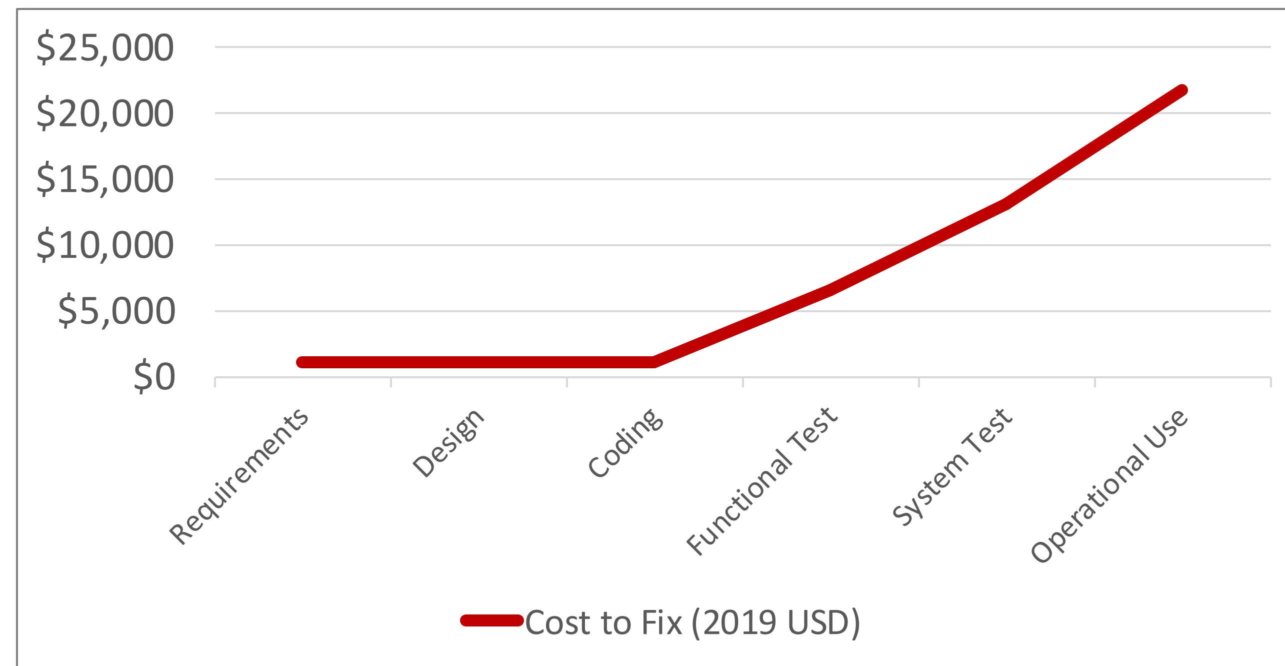
https://go.gmu.edu/SWE637

Adapted from slides by Jeff Offutt and Bob Kurtz

Traditional Testing

Driven by up-front analysis and modeling

We know that it's less expensive to find and fix problems early in the software lifecycle



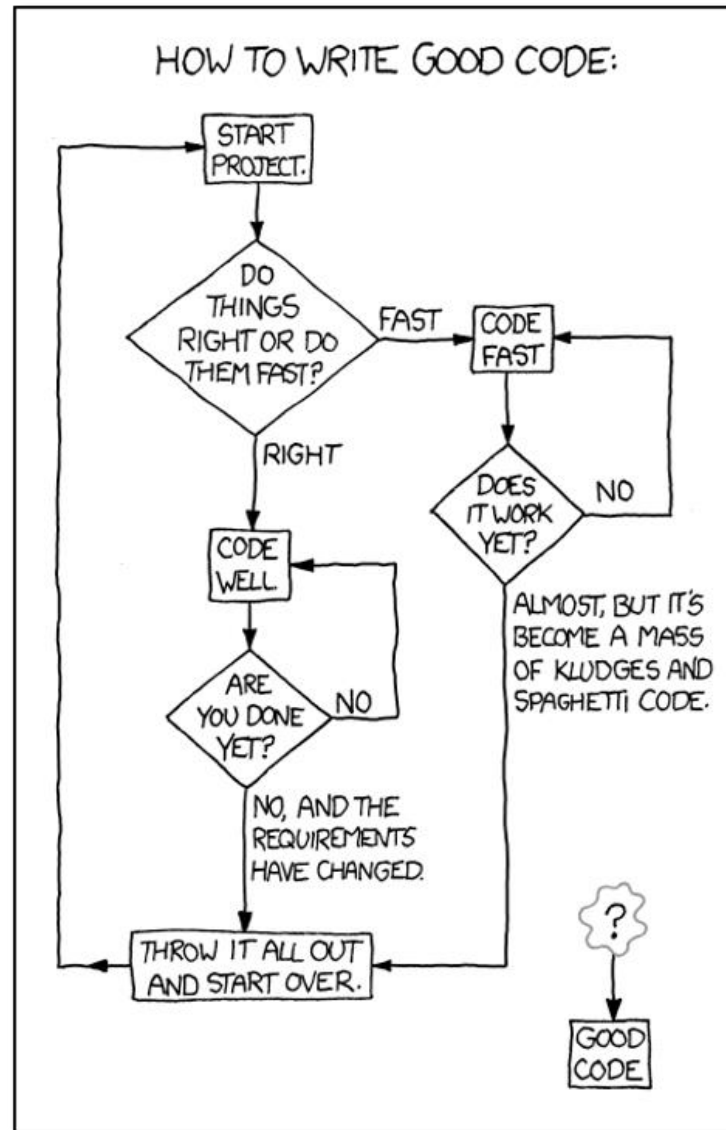
Software Engineering Institute; Carnegie Mellon University; Handbook CMU/SEI-96-HB-002

Traditional Testing Assumptions

Analysis and modeling can identify potential problems early in development

- …assuming the requirements are complete and unchanging, and how often does that happen? *(Hint: never)*
- In general, the waterfall model of software development is broken

How to write good code



Agile Development

Agile methods start by realizing that:

- engineers (and customers) are not good at developing requirements
- we don't anticipate the changes that we'll need, and often don't need the changes that we anticipate
- requirements get **out of date quickly** and may **change continuously**

In agile development, we start small and evolve over time

The Agile Manifesto

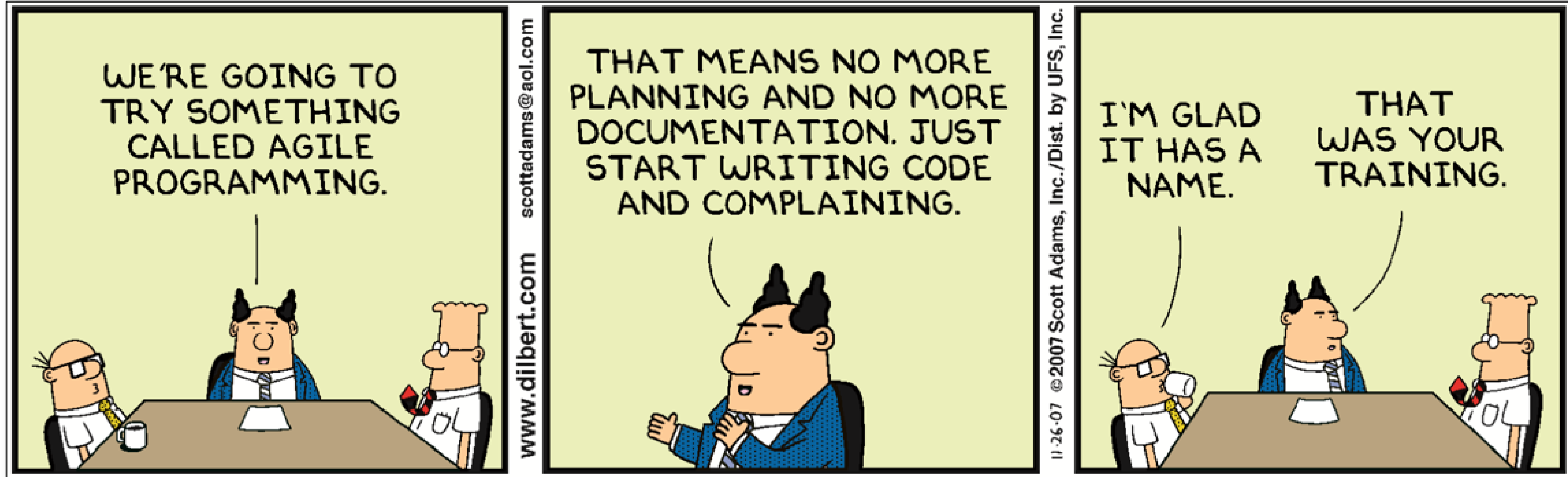
We are uncovering better ways of developing
software by doing it and helping others do it.
Through this work we have come to value:

Individuals and interactions *over processes and tools*
Working software *over comprehensive documentation*
Customer collaboration *over contract negotiation*
Responding to change *over following a plan*

That is, while there is value in the items on
the right, we value the items on the left more.

https://agilemanifesto.org

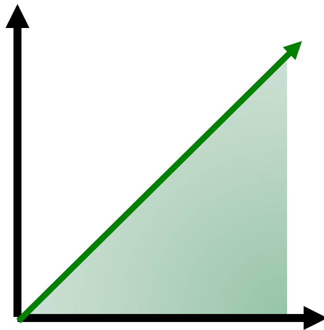
According to Dilbert…



The Test Harness as Guardian

Traditional correctness is ***universal***

$x,y,\ x \geq y$



Agile correctness is ***existential***

{ (1, 1) □ T
(1, 0) □ T
(0, 1) □ F
(10, 5) □ T
(10, 12) □ F }

Continuous Integration (CI)

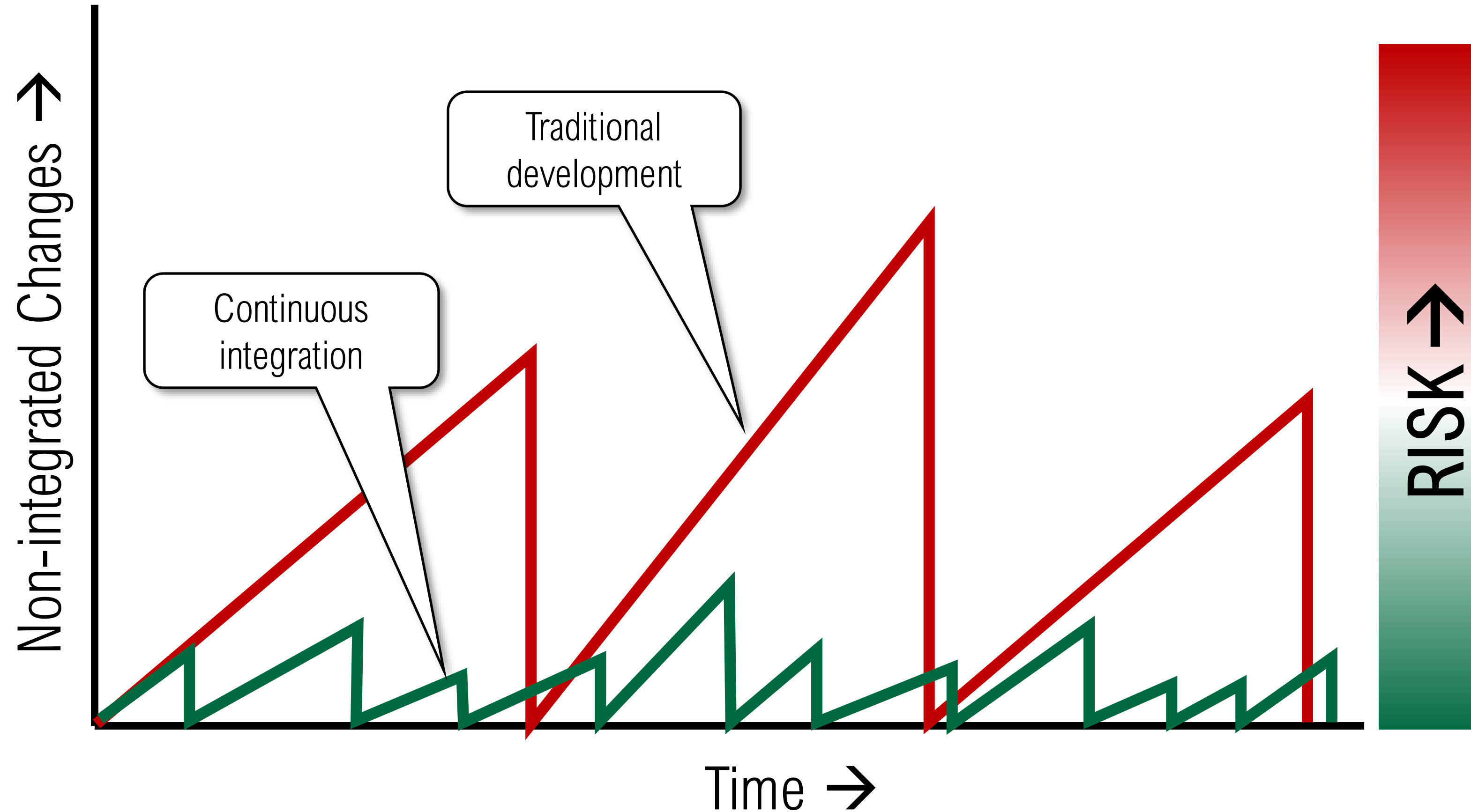
Agile methods work best when tests can be run against the current version of the software at any time (or *all* the time)

- Mistakes are caught earlier
- The value of tests is amplified by running them frequently

A continuous integration server (like Jenkins) rebuilds the system and reverifies test results whenever updates are checked in to the source code repository

A continuous Integration system doesn't just run tests, it decides if the modified system is still correct

CI Reduces Risk



With continuous integration you are never very far away from a working, fully-tested system.

Test Driven Development

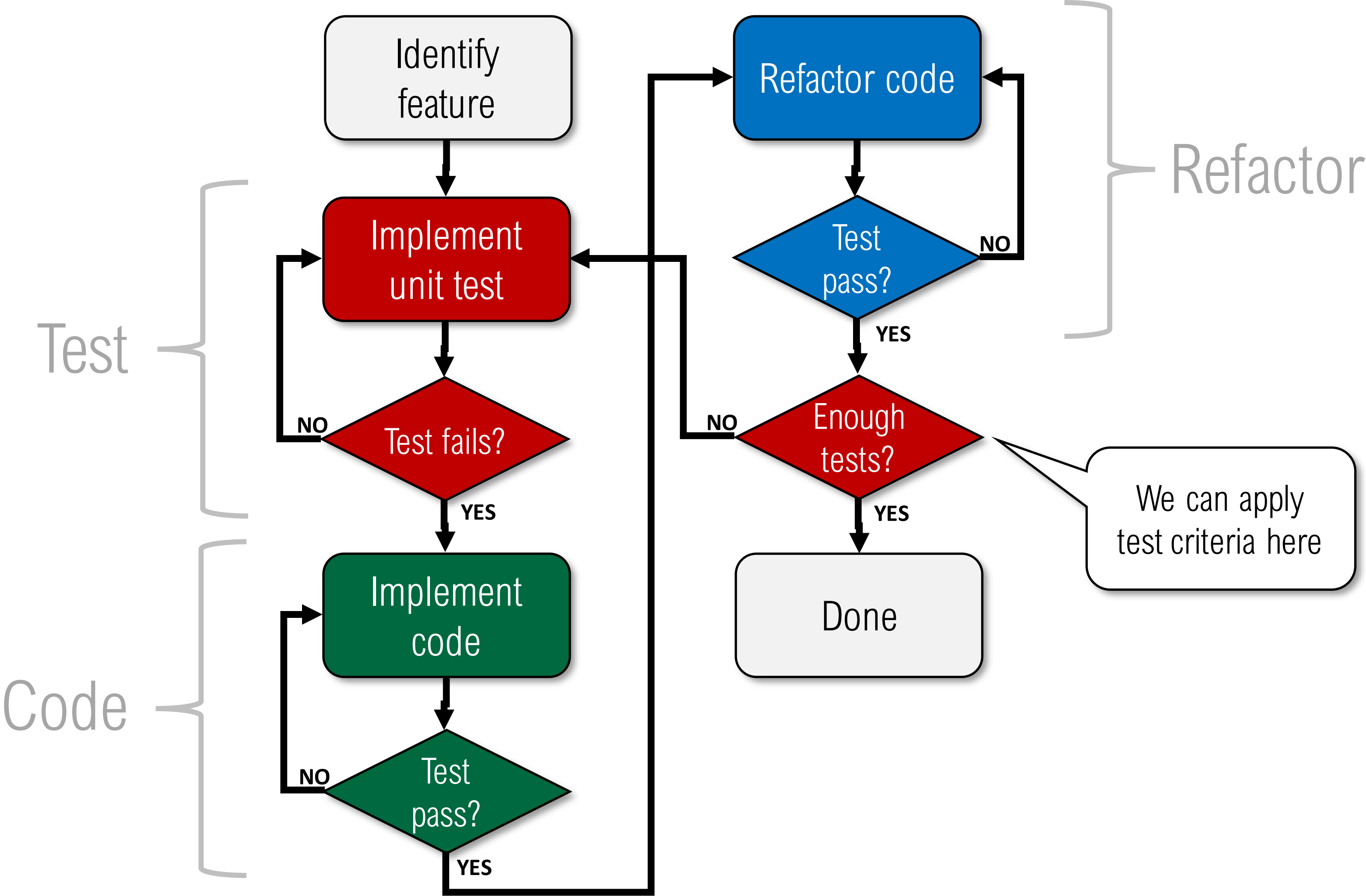
TDD is a method for developing low-level tests before implementing the software

- A test is written based on a feature derived from a requirement or a user story, then just enough code is written to satisfy the test

The core principle is **"you ain't gonna need it!"** (YAGNI)

- Don't build software until you're sure it's necessary for the desired functionality
- Reduces scope creep and software size, which in turn reduces the number of faults

Test Driven Development



Refactoring

“Refactoring consists of improving the internal structure of an existing program’s source code, while preserving its external behavior.”

– Agile Aliance, www.agilealliance.org

Why refactor?

- Improve code attributes like size, duplication, complexity; improve performance and maintainability
- Reduces *code smells* and *technical debt*

If you have a strong set of unit tests, you can refactor without fear.

Testing Legacy Systems

Much of today's software is **legacy**

- Often lacks up-to-date (or any) tests
- Legacy requirements may be outdated
- Design documentation, if there ever was any, is probably lost
- The only source of information is the code itself

Companies often choose not to change legacy software due to fear of failure

- Writing tests for millions of lines of legacy software is too expensive

We can't write full tests, but we can't just give up either…
what can we really do?

Incremental TDD

When a change is made, use TDD to write tests just for that change

Refactor the changed code to improve its structure and maintainability

As the project continues, the collection of tests grows, until eventually a substantial part of the system will have strong tests

The Testing Shortfall

Do TDD/ATDD tests test the software thoroughly?

- Do the tests achieve good coverage?
- Do the tests find most of the faults?
- If the tests pass, should we feel confident that the software is reliable?

Maybe not...

The Testing Shortfall

Most agile tests focus on "happy paths"

- Things that happen during normal use
- "Normal" use cases

They miss things like

- Things that *confused* users might do
- Things that *creative* users might do
- Things that ***malicious*** users might do

How to design better tests

Human-based approach

- Create additional user stories that describe exceptional paths
- This is an intuitive process that's hard to teach
- It's hard to know when you're finished

Modeling and criteria

- Model the input domain
- Model the software behavior with graphs, logic, or grammars
- It's a discrete process
- Built-in notion of completion

This is part 2 of the textbook.