Introduction to Software Testing
(*2nd edition*)
Chapter 4

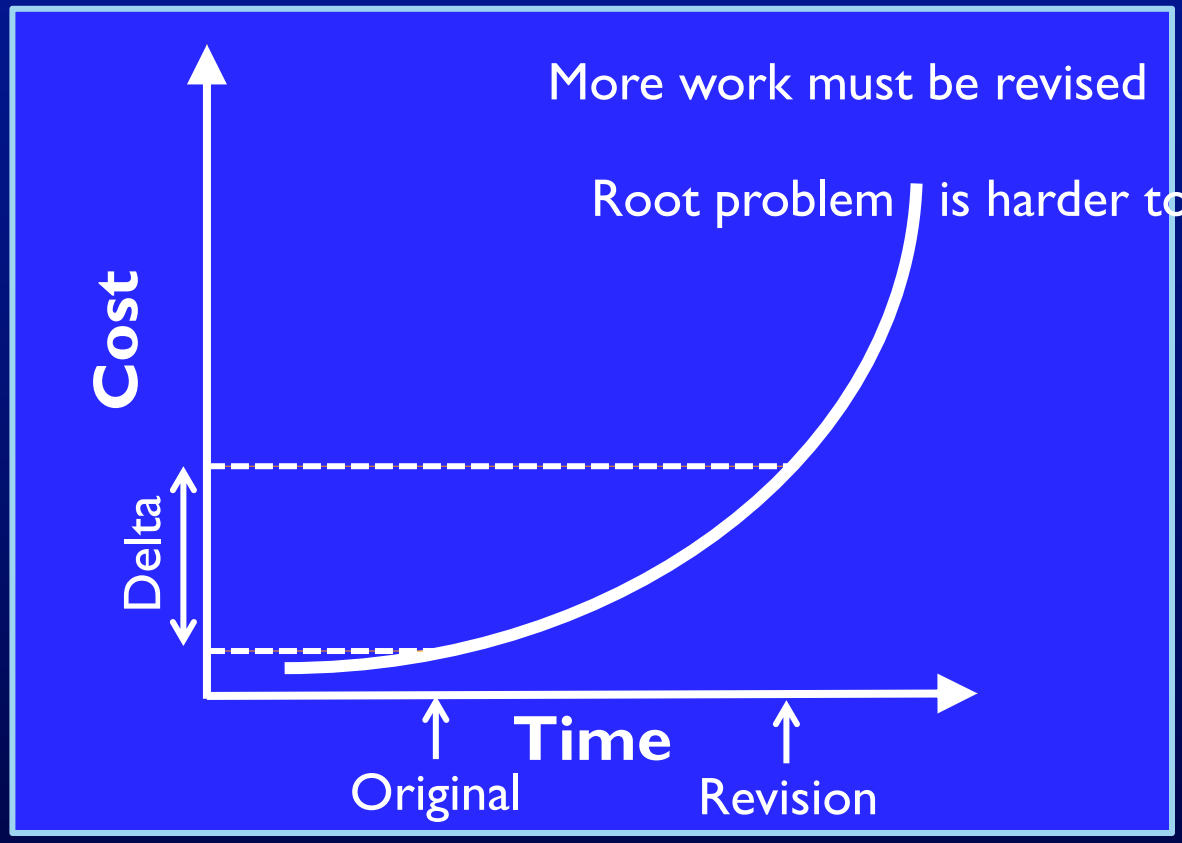
Putting Testing First

Paul Ammann & Jeff Offutt

August 2014

The Increased Emphasis on Testing

- Philosophy of traditional software development methods
  - Upfront analysis
  - Extensive modeling
  - Reveal problems as early as possible



Traditional Assumptions

1. Modeling and analysis can identify potential problems early in development

2. Savings implied by the cost-of-change curve justify the cost of modeling and analysis over the life of the project

- These are true if requirements are always complete and current
- But those annoying customers keep changing their minds!
  - Humans are naturally good at approximating
  - But pretty bad at perfecting
- These two assumptions have made software engineering frustrating and difficult for decades

Thus, agile methods …

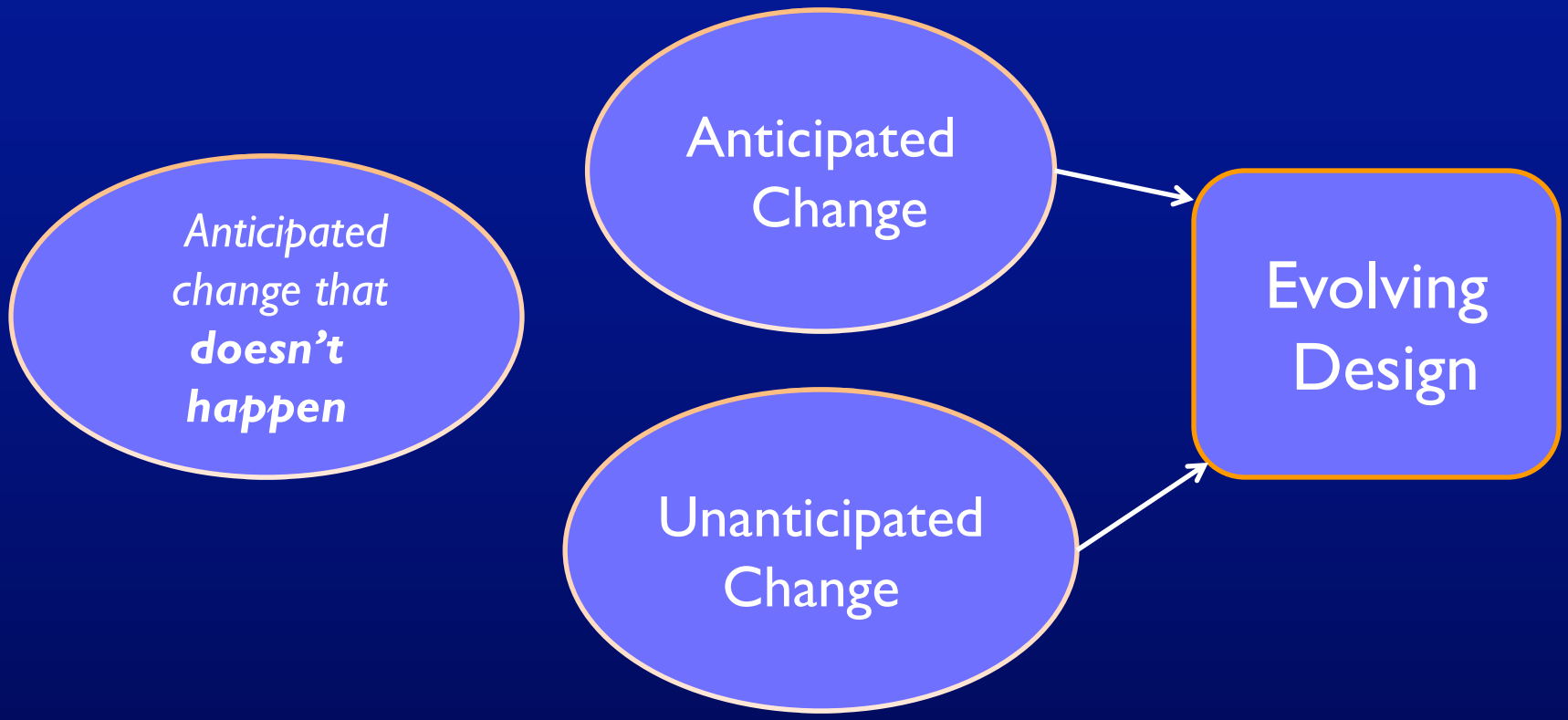
Why Be Agile ?

- Agile methods start by recognizing that neither assumption is valid for many current software projects
  - Software engineers are not good at developing requirements
  - We do not anticipate many changes
  - Many of the changes we do anticipate are not needed
- Requirements (and other "non-executable artifacts") tend to go out of date very quickly
  - We seldom take time to update them
  - Many current software projects change continuously
- Agile methods expect software to start small and evolve over time
  - Embraces software evolution instead of fighting it

Supporting Evolutionary Design

Traditional design advice says to anticipate changes

Designers often anticipate changes that don't happen



Both anticipated and unanticipated changes affect design

A Limited View of Correctness

- In traditional methods, we try to define all correct behavior completely, at the beginning
  - What is correctness?
  - Does "correctness" mean anything in large engineering products?
  - People are VERY BAD at completely defining correctness
- In agile methods, we redefine correctness to be relative to a specific set of tests
  - If the software behaves correctly on the tests, it is "correct"
  - Instead of defining all behaviors, we demonstrate some behaviors
  - Mathematicians may be disappointed at the lack of completeness

But software engineers ain't mathematicians!

Test Harnesses Verify Correctness

A *test harness* runs all automated tests efficiently and reports results to the developers

- Tests must be automated
  - Test automation is a prerequisite to test driven development
- Every test must include a test oracle that can evaluate whether that test executed correctly
- The tests replace the requirements
- Tests must be high quality and must run quickly
- We run tests every time we make a change to the software

Continuous Integration

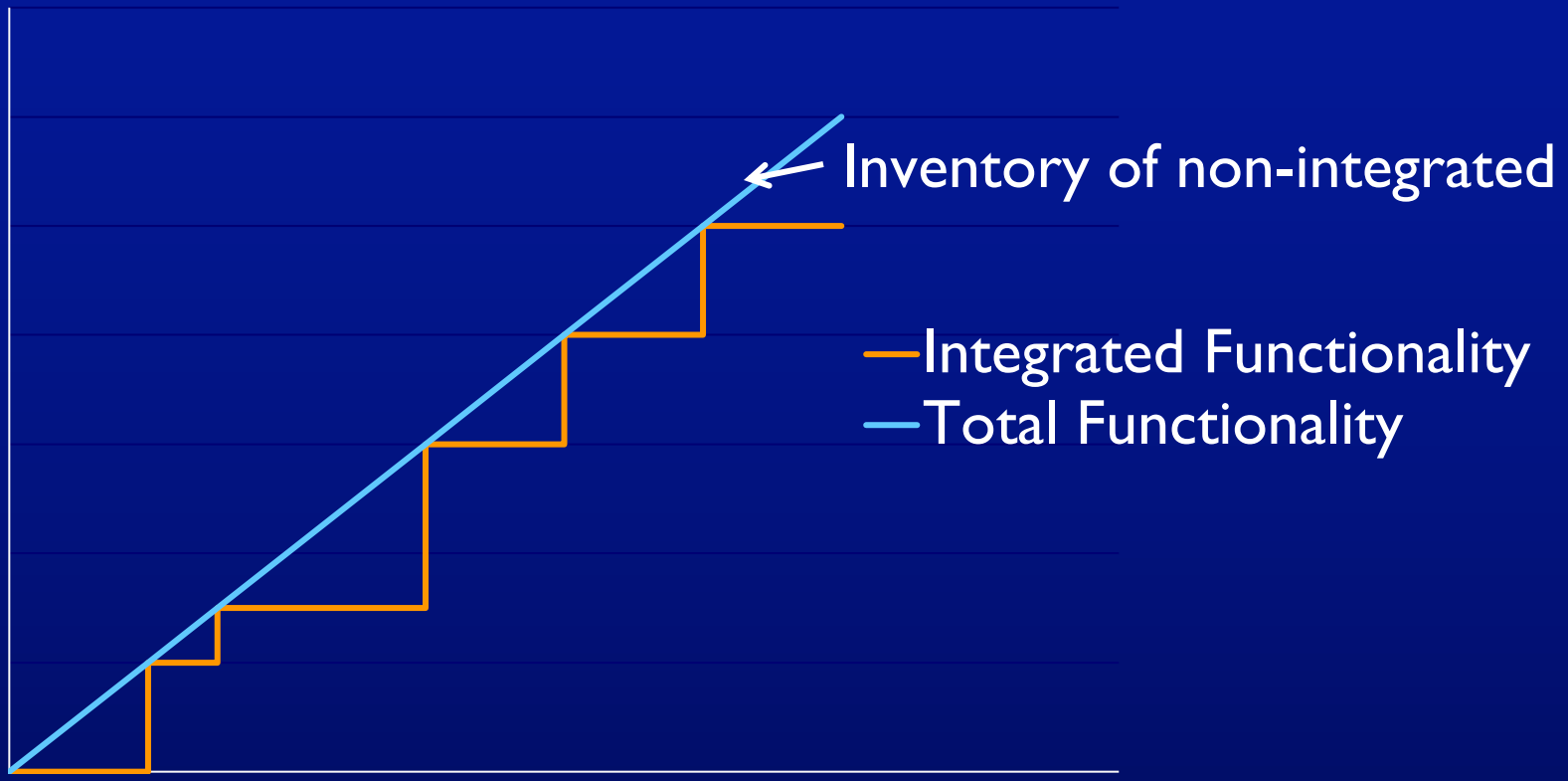
- Agile methods work best when the current version of the software can be run against all tests at any time

A *continuous integration server* rebuilds the system, returns, and reverifies tests whenever *any* update is checked into the repository

- Mistakes are caught earlier
- Other developers are aware of changes early
- The rebuild and reverify must happen as soon as possible
  - Thus, tests need to execute quickly

A *continuous integration server* doesn't just run tests, it decides if a modified system is still correct

Continuous Integration Reduces Risk



Non-integrated functionality is dangerous!

Adding Tests to Existing Systems

- Most of today's software is legacy
  - No legacy tests
  - Legacy requirements hopelessly outdated
  - Designs, if they were ever written down, lost
- Companies sometimes choose not to change software out of fear of failure

How to apply TDD to legacy software with no tests?

- Create an entire new test set? — too expensive!
- Give up? — a mixed project is unmanageable

Incremental TDD

- When a change is made, add TDD tests for just that change
  - Refactor
- As the project proceeds, the collection of TDD tests continues to grow
- Eventually the software will have strong TDD tests

The Testing Shortfall

- Do TDD tests (acceptance or otherwise) test the software well?
  - Do the tests achieve good coverage on the code?
  - Do the tests find most of the faults?
  - If the software passes, should management feel confident the software is reliable?

NO!

Why Not?

- Most agile tests focus on "*happy paths*"
  - What should happen under normal use
- They often miss things like
  - Confused-user paths
  - Creative-user paths
  - Malicious-user paths

The agile methods literature does not give much guidance

What Should Testers Do?

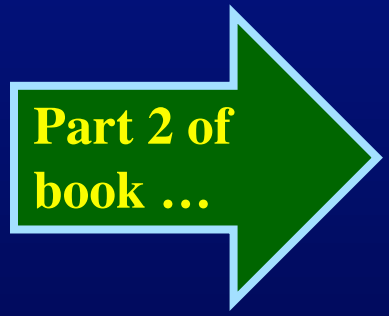
Ummm ... Excuse me, Professor ...

What do I do?

Design Good Tests

1. Use a human-based approach

- Create additional user stories that describe non-happy paths
- How do you know when you're finished?
- Some people are very good at this, some are bad, and it's hard to teach



2. Use modeling and criteria

- Model the input domain to design tests
- Model software behavior with graphs, logic, or grammars
- A built-in sense of completion
- Much easier to teach—engineering
- Requires discrete math knowledge

Summary

- More companies are putting testing first
- This can dramatically decrease cost and increase quality
- A different view of "*correctness*"
  - Restricted but practical
- Embraces evolutionary design
- TDD is definitely not test automation
  - Test automation is a prerequisite to TDD
- Agile tests aren't enough