Software Testing

Why Test?

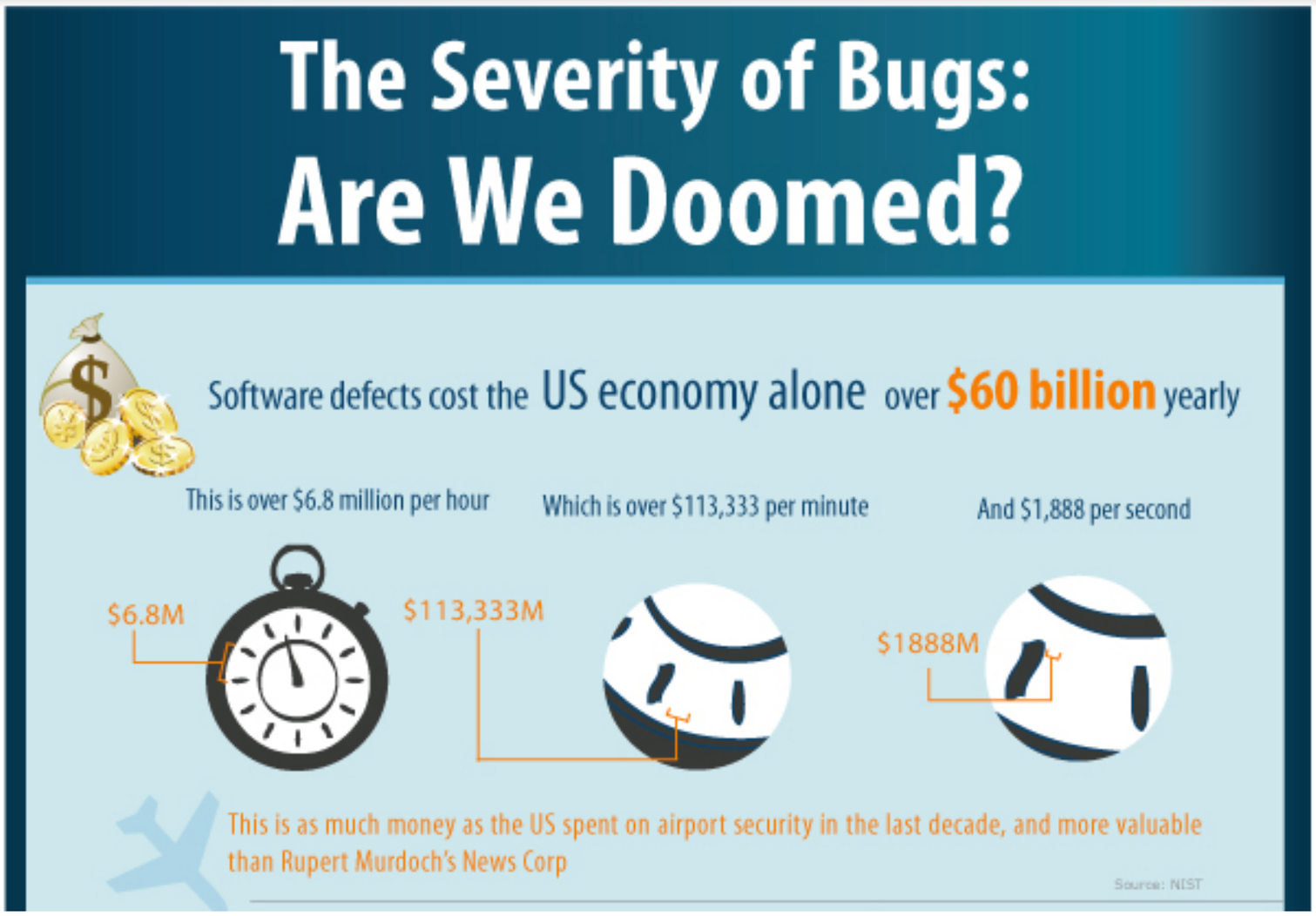
The stuff we call "software" is not like anything that human society is used to thinking about. Software is something like a machine, and something like mathematics, and something like language, and something like thought, and art, and information.... but software is not in fact any of those other things. The protean quality of software is one of the great sources of its fascination. It also makes software very powerful, very subtle, very unpredictable, and very risky.

Some software is bad and buggy. Some is "robust," even "bulletproof." The best software is that which has been tested by thousands of users under thousands of different conditions, over years. It is then known as "stable." This does NOT mean that the software is now flawless, free of bugs. It generally means that there are plenty of bugs in it, but the bugs are well-identified and fairly well understood.

- There is simply no way to assure that software is free of flaws. Though software is mathematical in nature, it cannot by "proven" like a mathematical theorem; software is more like language, with inherent ambiguities, with different definitions, different assumptions, different levels of meaning that can conflict.

Quote by Bruce Sterling, from: A Software Testing Primer, Nick Jenkins

- Not technically true, see Formal Methods



https://www.typemock.com/software-bugs-infographic

“A bug found at design time costs ten times less to fix than one in coding and a hundred times less than one found after launch”

1:6:10:1000
requirements : design : coding : implementation
(Barry Boehm)

Up to 50% of bug fixes actually introduce new errors in the code!

- Testing reduces risk
- Testing is not optional and needs to infuse the development lifecycle
- Test early; Test often
- Testing mindset

Terminology

- QA
- QC
- Testing
- Verification
- Validation

- Black-Box testing
- White-Box testing (glass-box)
- Alpha Testing
- Beta Testing

- Performance testing
- Security testing
- Stress testing
- Recovery testing
- …

- Static Analysis
- Dynamic Analysis

https://www.parasoft.com/

https://scan.coverity.com/o/oss_success_stories

https://developer.apple.com/library/content/documentation/Performance/Conceptual/PerformanceOverview/PerformanceTools/PerformanceTools.html

- Re-testing
- Regression Testing

- For a great example, see SQLite:
- http://www.sqlite.org/testing.html)

- Unit Testing (test case, test harness/suite)
- Integration Testing (stubs, drivers)
- System Testing

Unit Testing

Unit testing verifies the functioning in isolation of software elements that are separately testable. Depending on the context, these could be the individual subprograms or a larger component made of highly cohesive units. Typically, unit testing occurs with access to the code being tested and with the support of debugging tools. The programmers who wrote the code typically, but not always, conduct unit testing.

SWEBOKv3

Integration Testing

Integration testing is the process of verifying the interactions among software components. Classical integration testing strategies, such as top-down and bottom-up, are often used with hierarchically structured software.

Modern, systematic integration strategies are typically architecture-driven, which involves incrementally integrating the software components or subsystems based on identified functional threads.

Integration testing is often an ongoing activity at each stage of development during which software engineers abstract away lower-level perspectives and concentrate on the perspectives of the level at which they are integrating. For other than small, simple software, incremental integration testing strategies are usually preferred to putting all of the components together at once—which is often called “big bang” testing.

System Testing

System testing is concerned with testing the behavior of an entire system. Effective unit and integration testing will have identified many of the software defects. System testing is usually considered appropriate for assessing the non-functional system requirements—such as security, speed, accuracy, and reliability (see Functional and Non-Functional Requirements in the Software Requirements KA and Software Quality Requirements in the Software Quality KA). External interfaces to other applications, utilities, hardware devices, or the operating environments are also usually evaluated at this level.

- Acceptance Testing (User Acceptance Testing, UAT), ATDD
- Behavior Driven Development
- Usability Testing
- Test Automation
- QA Dashboard (e.g. TFS Dashboard)

> I love watching developers who take part as observers in usability studies. As a former developer myself I know the hubris that goes along with designing software. In the throes of creation it is difficult for you to conceive that someone else, let alone a user (!), could offer better input to the design process than your highly paid, highly educated self.
>
> Typically developers sit through the performance of the first evaluator and quietly snigger to themselves, attributing the issues to 'finger trouble' or user ineptitude. After the second evaluator finds the same problems the comments become less frequent and when the third user stumbles in the same position they go quiet.
>
> By the fourth user they've got a very worried look on their faces and during the fifth pass they're scratching at the glass trying to get into to talk to the user to "find out how to fix the problem".

- Code Coverage
- Traceability

- Fault — cause of a malfunction; wrong or missing code
- Failure — undesired effect observed in the system's delivered service; manifestation of a Fault
- Defect
- Error — human mistake that contributed to a fault, or a difference between actual and expected output
- Bug — tester language

Testing can reveal failures, but it is the faults that can and must be removed

Faults cannot always be unequivocally identified

Test Planning

So how do you plan your testing?

At the start of testing there will be a (relatively) large number of issues and these can be uncovered with little effort. As testing progress more and more effort is required to uncover subsequent issues.

The law of diminishing returns applies and at some point the investment to uncover that last 1% of issues is outweighed by the high cost of finding them. The cost of letting the customer or client find them will actually be less than the cost of finding them in testing.

The purpose of test planning therefore is to put together a plan which will deliver the right tests, in the right order, to discover as many of the issues with the software as time and budget allow.

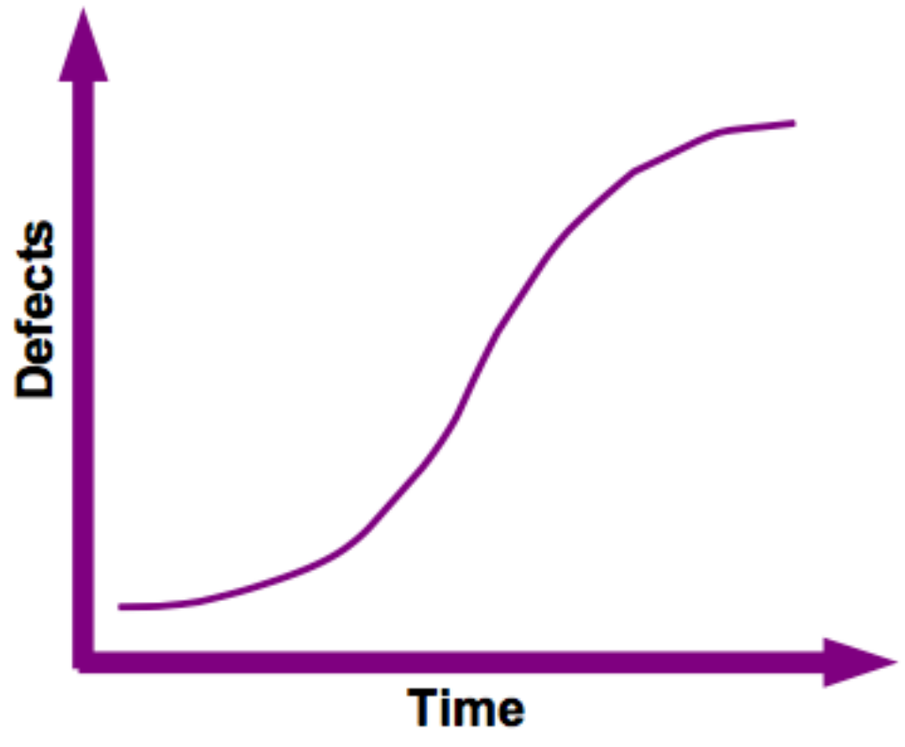


Figure 6: Typical defect discovery rate

Unit Testing

See other slide set

Test Scripting

- Scripted evaluation, or
- Free-form exploration

Test Cases

- Document a test
- Prove a requirement (at least one test case per requirement)
- Should have at least 2: one positive, one negative

Elements of a Test Case

The following table list the suggested items a test case should include:

ITEM	DESCRIPTION
Title	A unique and descriptive title for the test case
Priority	The relative importance of the test case (critical, nice-to-have,etc.)
Status	For live systems, an indicator of the state of the test case. Typical states could include : Design – test case is still being designed Ready – test case is complete, ready to run Running – test case is being executed Pass – test case passed successfully Failed – test case failed Error – test case is in error and needs to be rewritten
Initial configuration	The state of the program before the actions in the "steps" are to be followed. All too often this is omitted and the reader must guess or intuit the correct pre-requisites for conducting the test case.
Software Configuration	The software configuration for which this test is valid. It could include the version and release of software-under-test as well as any relevant hardware or software platform details (e.g. WinXP vs Win95)
Steps	An ordered series of steps to conduct during the test case, these must be detailed and specific. How detailed depends on the level of scripting required and the experience of the tester involved.
Expected behaviour	What was expected of the software, upon completion of the steps? What is expected of the software. Allows the test case to be validated with out recourse to the tester who wrote it.

Testing in Agile

1. User Story with acceptance test
2. Write integration test cases
3. JIT modeling, design code
4. Write unit tests
5. Write code to implement features described in the story that themselves represent requirements

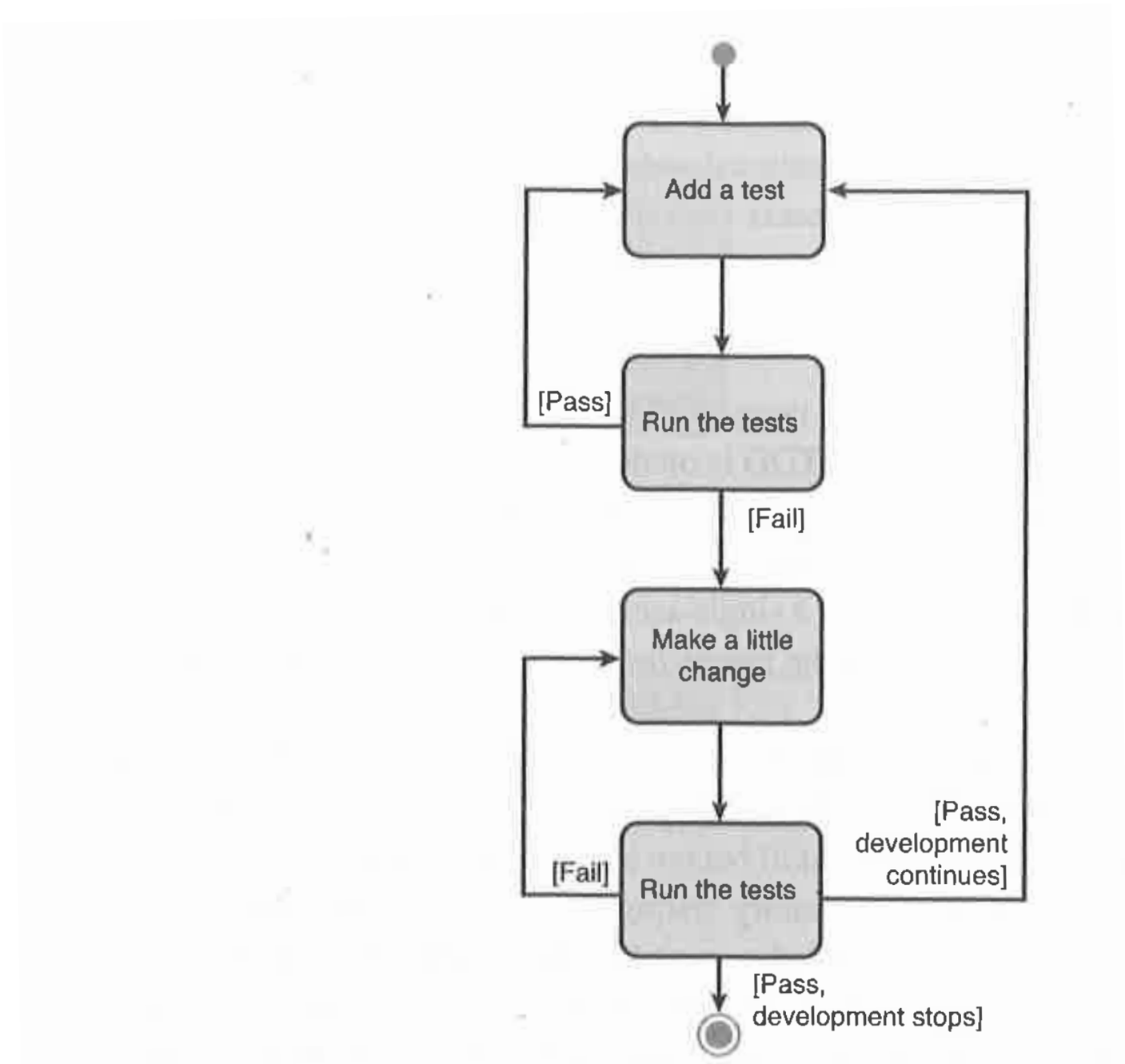


Figure 15.8 The steps of test-first development (TFD)

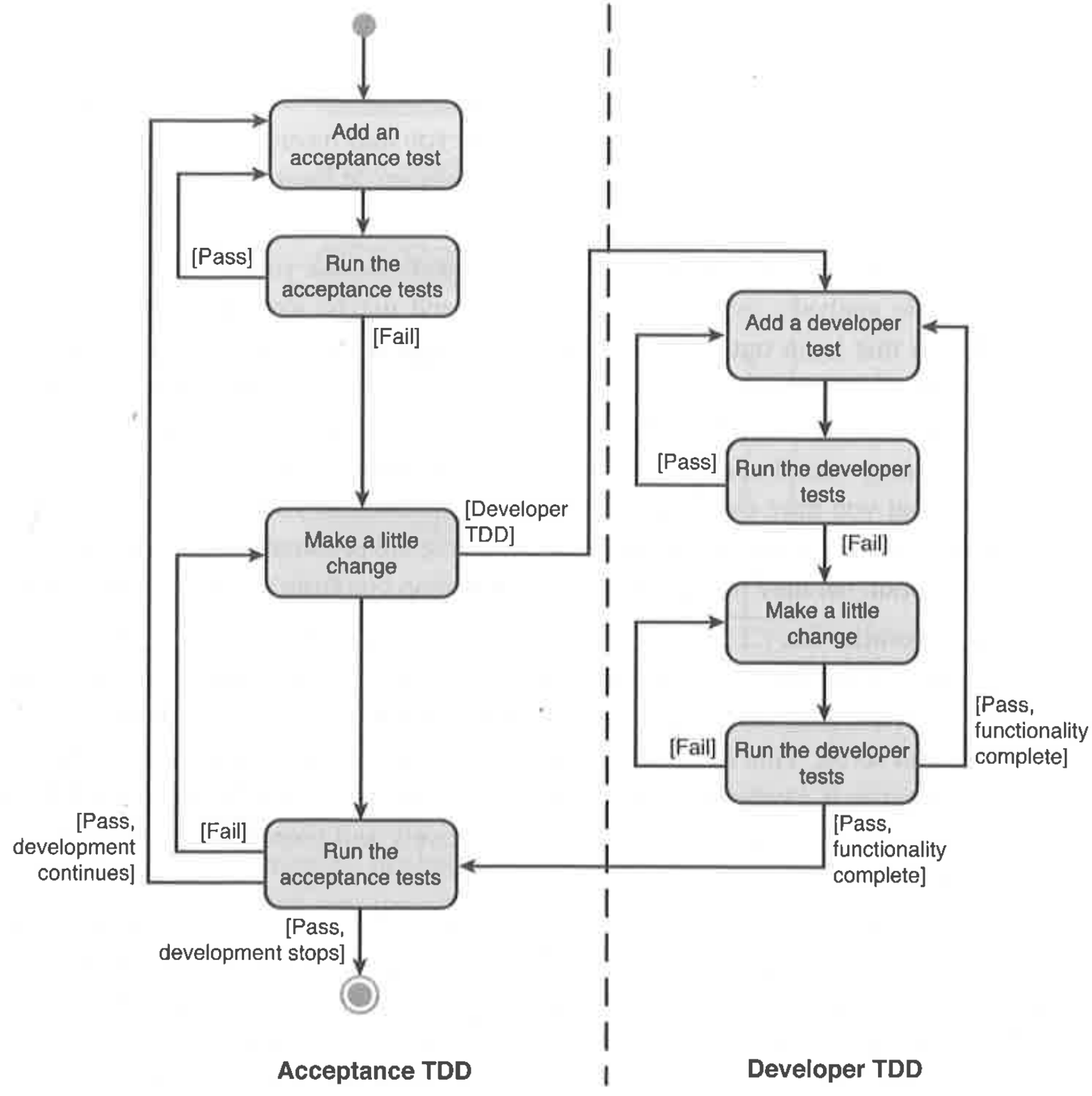


Figure 15.9 Acceptance and developer TDD together

Integration & System Testing

See other slides