

TVVS - Software Testing Verification and Validation

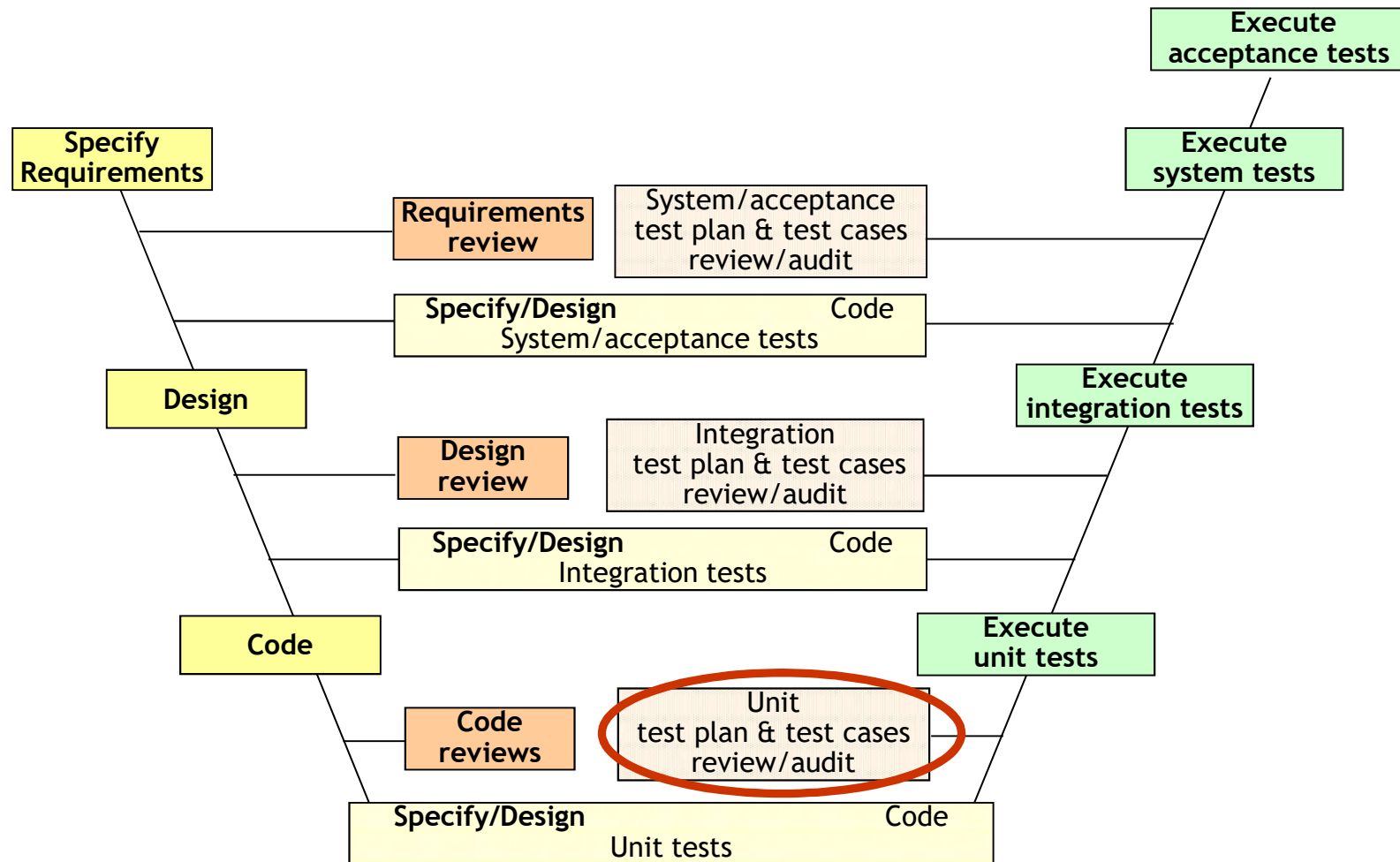
Unit Testing

Ana Paiva

apaiva@fe.up.pt www.fe.up.pt/~apaiva

Unit Testing

Unit testing



The extended V-model of software development [I.Burnstein]

Unit testing - definition

- **Unit testing:** Testing of individual hardware or software units or groups of related units [IEEE 90].
- **Unit testing** is a development procedure where programmers create tests as they develop software. The tests are simple short tests that test functionality of a particular unit or module of their code, such as a class or function.

Unit testing - definition

- **Unit testing** is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinized for proper operation. This testing mode is a component of **Extreme Programming (XP)**, a pragmatic method of software development that takes a meticulous approach to building a product by means of continual testing and revision.
- A **unit test** is a piece of a code (usually a method) that invokes another piece of code and checks the correctness of some assumptions afterward. If the assumptions turn out to be wrong, the unit test has failed. A “unit” is a method or function.

Unit Testing

[<http://glossary.istqb.org/>]

- **component testing:** The testing of individual software components.
- **unit testing:** *See component testing.*
- **unit:** *See component.*
- **component:** A minimal software item that can be tested in isolation.

Benefits of unit testing

- Facilitates change
 - Allows **refactor** code at a later date and make sure the module still works correctly
- Simplifies integration
 - Helps to eliminate uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, **integration testing** becomes much easier
- Documentation
 - Provides a sort of **living documentation** of the system. Developers looking to learn what functionality is provided by a unit and how to use it can look at the unit tests to gain a basic understanding of the unit API

“Good” unit tests

- What properties a unit test should have?
 - It is automated and repeatable
 - It is easy to implement
 - Once it is written, it stays on for the future
 - Anyone can run it
 - It runs at the push of a button
 - It runs quickly
- Unit tests are code ...
 - Maintainability
 - Readability
 - Correctness
 - Documentation
 - ...

“Good” unit tests

- **Runs fast, runs fast, runs fast.** If the tests are slow, they will not be run often.
- **Separates or simulates environmental dependencies such as databases, file systems, networks, queues, and so on.** Tests that exercise these will not run fast, and a failure does not give meaningful feedback about what the problem actually is.
- **Is very limited in scope.** If the test fails, it is obvious where to look for the problem. Use few Assert calls so that the offending code is obvious. It is important to only test one thing in a single test.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

“Good” unit tests (2)

- **Runs and passes in isolation.** If the tests require special environmental setup or fail unexpectedly, then they are not good unit tests. Change them for simplicity and reliability. Tests should run and pass on any machine. The "works on my box" excuse doesn't work.
- **Often uses stubs and mock objects.** If the code being tested typically calls out to a database or file system, these dependencies must be simulated, or mocked. These dependencies will ordinarily be abstracted away by using interfaces.
- **Clearly reveals its intention.** Another developer can look at the test and understand what is expected of the production code.

Separation of interface from implementation

- Because some classes may have references to other classes, testing a class can frequently spill over into testing another class.
 - Ex.: classes that depend on a database: in order to test the class, the tester often writes code that interacts with the database.
- This is a mistake
 - unit test should never go outside of its own class boundary.
- Abstract an interface around database connection and implement it with your own **mock objects**
 - the independent unit can be more thoroughly tested
 - this results in a higher quality unit that is also more maintainable.

Mock objects

- Mock objects are simulated objects that mimic the behavior of real objects in controlled ways.
- Useful when a real object is impractical or impossible to incorporate into a unit test. If an object has any of the following characteristics, it may be useful to use a mock object in its place:
 - supplies non-deterministic results (e.g., the current time or the current temperature);
 - has states that are difficult to create or reproduce (e.g., a network error);
 - is slow (e.g., a complete database, which would have to be initialized before the test);
 - does not yet exist or may change behavior;
 - would have to include information and methods exclusively for testing purposes (and not for its actual task).

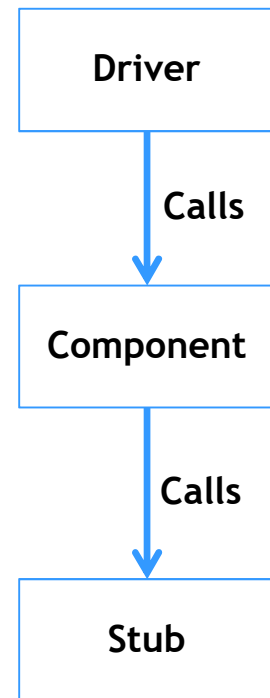
Mock objects (example)

- An alarm clock program which causes a bell to ring at a certain time might get the current time from the outside world. To test this, the **test must wait** until the alarm time to know whether it has rung the bell correctly. If a mock object is used in place of the real object, it can be programmed to provide the bell-ringing time (whether it is actually that time or not) so that the alarm clock program can be tested in isolation.

Definitions

[<http://glossary.istqb.org/>]

- **driver:** A software component or test tool that replaces a component that takes care of the control and/or the calling of a component or system.
- **stub:** A skeletal or special-purpose implementation of a software component, used to develop or test a component that calls or is otherwise dependent on it. It replaces a called component.
- **test harness:** A test environment comprised of stubs and drivers needed to execute a test.



Six rules of unit testing

- Write the test first
- Never write a test that succeeds the first time
- Start with the null case, or something that doesn't work
- Don't be afraid of doing something trivial to make the test work
- Loose coupling and testability go hand in hand
- Use mock objects

[<http://radio.weblogs.com/0100190/stories/2002/07/25/sixRulesOfUnitTesting.html>]

Unit testing - best practices

- Keep unit tests small and fast
 - Ideally the entire test suite should be executed before every code check in.
- Unit tests should be fully automated and non-interactive
 - The test suite is normally executed on a regular basis and must be fully automated to be useful. If the results require manual inspection the tests are not proper unit tests.
- Make unit tests simple to run
 - Configure the development environment so that single tests and test suites can be run by single command or a one button click

Unit testing - best practices

- Keep testing at unit level
 - Unit testing is about testing units. Units should be tested in isolation. Avoid temptation to test an entire workflow using a unit testing framework, as such tests are slow and hard to maintain. Workflow testing may have its place, but it is not unit testing and it must be set up and executed independently
- Start of simple
 - One simple test is infinitely better than no tests at all. A simple test class will verify the presence and correctness of both the build environment, the unit testing environment, the execution environment and the coverage analysis tool, etc.

Unit testing - best practices

- Keep tests independent
 - To ensure testing robustness and simplify maintenance, tests should never rely on other tests nor should they depend on the ordering in which tests are executed
- Name tests properly
 - Make sure each test method test one distinct feature of the unit being tested and name the test methods accordingly. The typical naming convention is test[what] such as testSaveAs(), testAddListener(), testDeleteProperty(), etc.

Unit testing - best practices

- Keep tests close to the unit being tested
 - If you are testing a class Foo, the test class should be called FooTest and keep in the same package (directory) as Foo.
- Think black-box
 - Act as a 3rd party class consumer, and test if the class fulfills its requirements. As try to tear it apart.
- Think white-box
 - After all, the test programmer also wrote the class being tested, and extra effort should be put into testing the most complex logic.

Unit testing - best practices

- Test the trivial cases too
 - It is sometimes recommended that all non-trivial cases should be tested and that trivial methods like simple setters and getters can be omitted. However, there are several reasons why trivial cases should be tested too:
 - Trivial is hard to define. It may mean different things to different people.
 - The trivial cases can contain errors too, often as a result of copy-paste operations
- Focus on execution coverage
 - Ensure that the code is actually executed on some input parameters
- Cover boundary cases
 - Make sure the parameter boundary cases are covered, e.g., test negatives, 0, positive, smallest, larger, etc

Unit testing - best practices

- Provide a random generator
 - When boundary are covered, you may use random generated parameters.
- Use explicit asserts
 - Always prefer `assertEquals(a,b)` to `assertTrue(a==b)` as the former gives more useful information
- Provide negative tests
 - Negative tests intentionally misuse the code and verify robustness and appropriate error handling

Unit testing - best practices

- Design code with testing in mind
 - Writing and maintaining unit tests are costly, so reducing cyclomatic complexity in the code are ways to reduce this cost
- Know the cost of testing
 - Not writing unit tests is costly, but writing unit tests is costly too. There is a trade-off between the two, and in terms of execution coverage the typical industry standard is at about 80%
- Prioritize testing
 - If there is not enough resources to test all parts you should establish a prioritization

Unit testing - best practices

- Prepare test code for failures
 - If a test fails, the remaining tests will not be executed. Always prepare for test failure so that the failure of a single test does not bring down the entire test suite execution
- Write tests to reproduce bugs
 - When a bug is reported, write a test to reproduce the bug (i.e., failing test) and use this test as a success criteria when fixing the code
- Know the limitations
 - Unit tests can never prove the correctness of code.
- ...
- ...

Limitations of unit testing

- Testing, in general, cannot be expected to catch every error in the program. Unit tests can only show the presence of errors; it cannot show the absence of errors.
- It only tests the functionality of the units themselves.
- It may not catch integration errors, performance problems, or other system-wide issues.
- Unit testing is more effective if it is used in conjunction with other software testing activities.

Limitations of unit testing

- Software testing is a combinatorial problem.
 - For example, every **boolean decision** statement requires at least **two tests**:
 - one with an outcome of "true" and one with an outcome of "false".
 - As a result, for **every line of code** written, programmers often need **3 to 5 lines of test code**. Therefore, it is unrealistic to test all possible input combinations for any non-trivial piece of software without an automated characterization test generation tool such as JUnit Factory used with Java code or many of the tools.

Limitations of unit testing

- To obtain the intended benefits from unit testing use of a **version control system** is essential
 - If a later version of the unit fails a particular test that it had previously passed, the version-control software can provide a list of the source code changes (if any) that have been applied to the unit since that time.
- It is also essential to **implement a sustainable process** for ensuring that test case failures are reviewed daily and addressed immediately
 - If such a process is not implemented and ingrained into the team's workflow, the application will evolve out of sync with the unit test suite — increasing false positives and reducing the effectiveness of the test suite.

Unit testing frameworks

- Help simplify the process of unit testing
 - Log test cases that fail
 - Automatically flag and report in a summary these failed test cases.
 - Depending upon the severity of a failure, the framework may halt subsequent testing.
- Examples
 - JUnit
 - JTest
 - NUnit
 - MbUnit
 - ...

[http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks]

Test environment

- **test environment:** An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test. Same as **test bed**.
[According to ISTQB]
- The purpose is to ensure that there is a well known and fixed environment in which tests are run so that results are repeatable.
- Examples:
 - loading a database with a specific, known set of data
 - erasing a hard disk and installing a known clean operating system installation
 - copying a specific known set of files
 - preparation of input data and setup/creation of fake or mock objects

Some more definitions

[<http://glossary.istqb.org/>]

- **testware:** Artifacts produced during the test process required to plan, design, and execute tests, such as documentation, scripts, inputs, expected results, set-up and clear-up procedures, files, databases, environment, and any additional software or utilities used in testing.
- **test basis:** All documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based. If a document can be amended only by way of formal amendment procedure, then the test basis is called a frozen test basis.

Setup and teardown (1)

- You want to avoid duplicated code when several tests share the same initialization and cleanup code.
- ... before and after each test method inside a test class
 - **JUnit:** Use the `setUp()` and `tearDown()` methods. Both of these methods are part of the `junit.framework.TestCase` class.
 - **NUnit:** build methods and annotate them with `SetUp` and `TearDown` tags

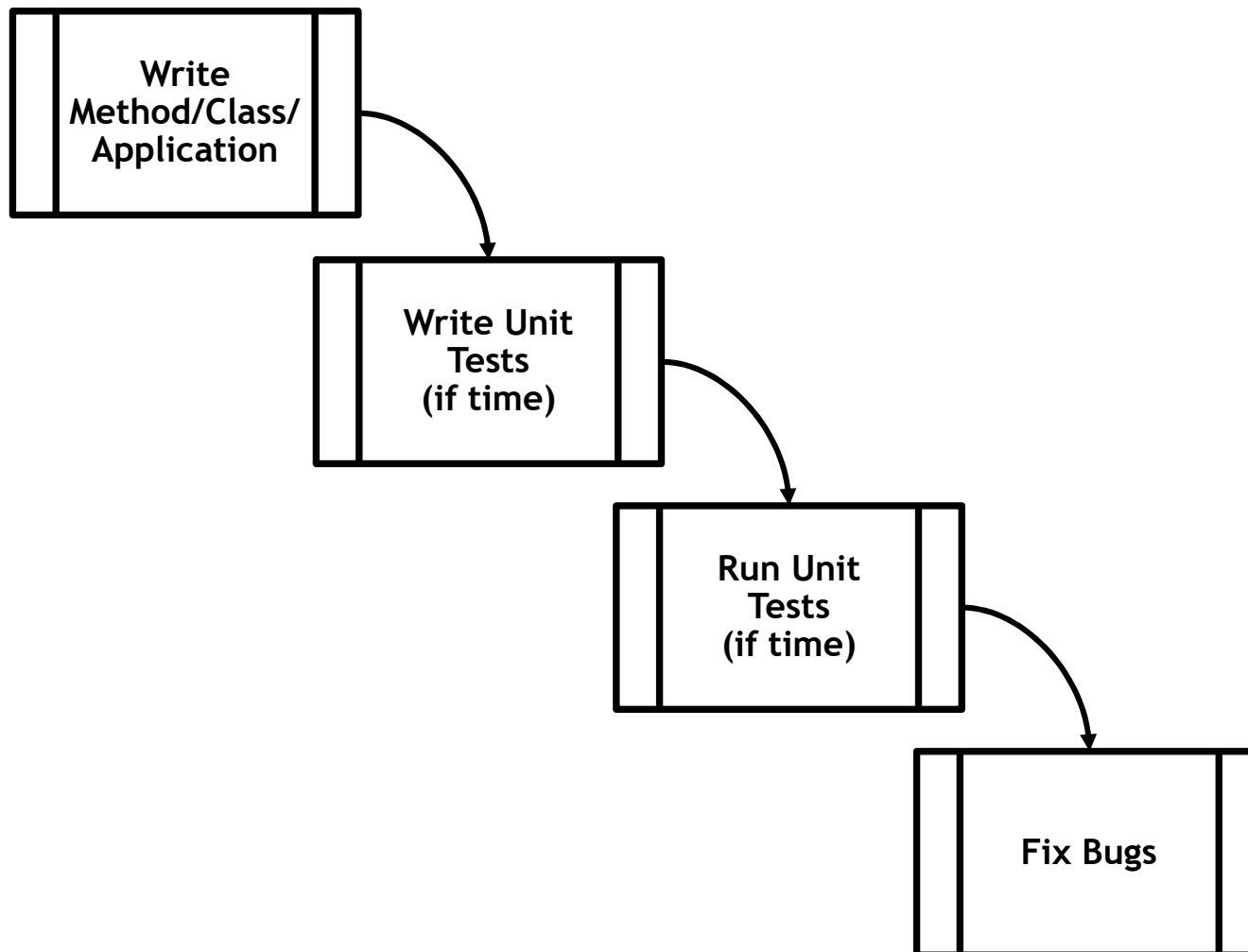
Setup and teardown (2)

■ One-time set up and tear down

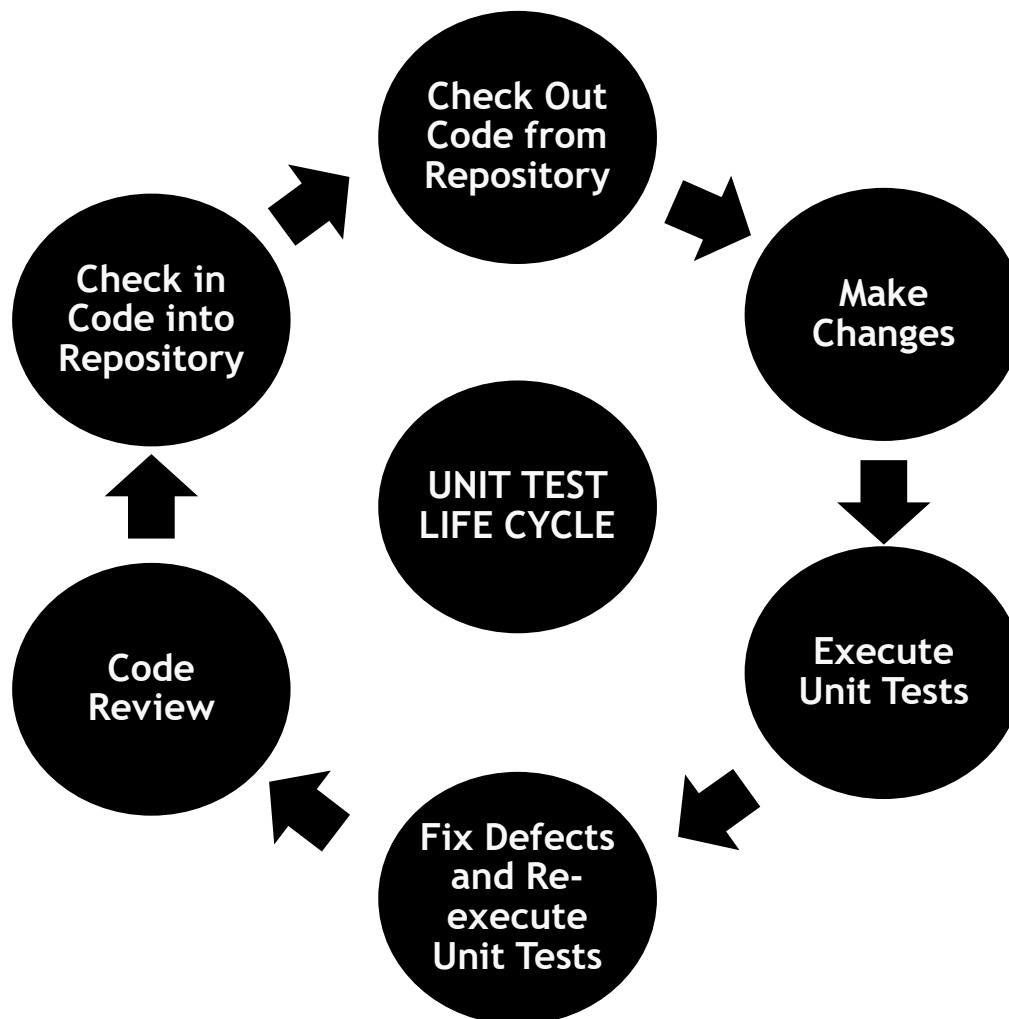
- You want to run some setup code one time and then run several tests. You only want to run your cleanup code after all of the tests are finished, ex.: establish a database connection.
- **JUnit**: Use the `junit.extensions.TestSetup` class to define **test suites**
 - Pass a `TestSuite` to the `TestSetup` constructor. This means that `TestSetup`'s `setUp()` method is called once before the entire suite, and `tearDown()` is called once afterwards.
`TestSetup setup = new TestSetup(new TestSuite(TestPerson.class)) {...}`
- **NUnit**: methods annotated with `TestFixtureSetup` and `TestFixtureTearDown` attributes.

Unit Testing within Development Processes

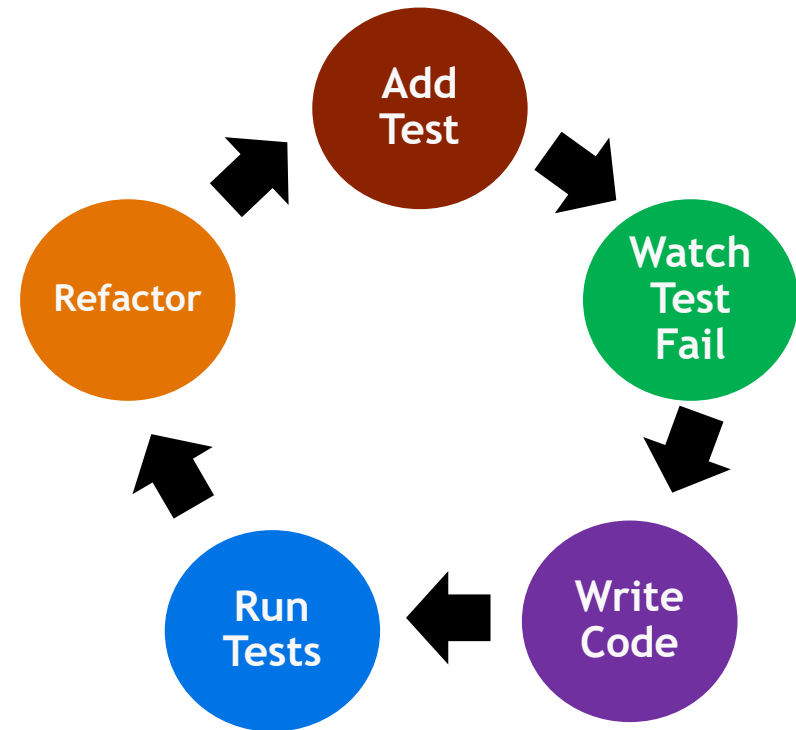
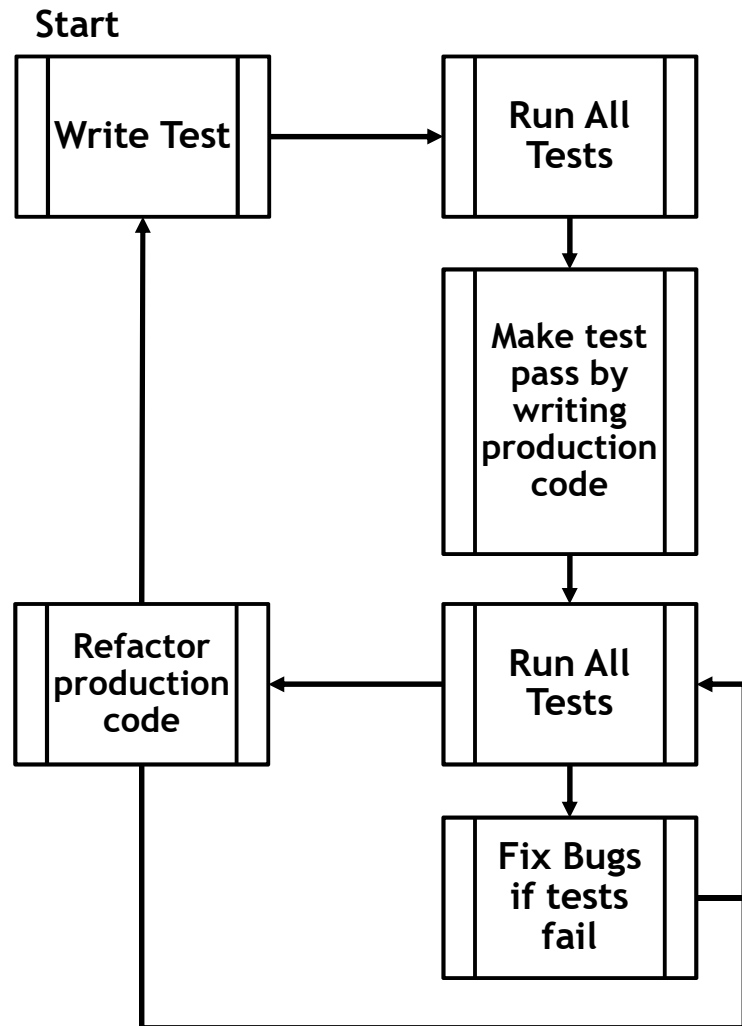
Traditional way of writing unit tests



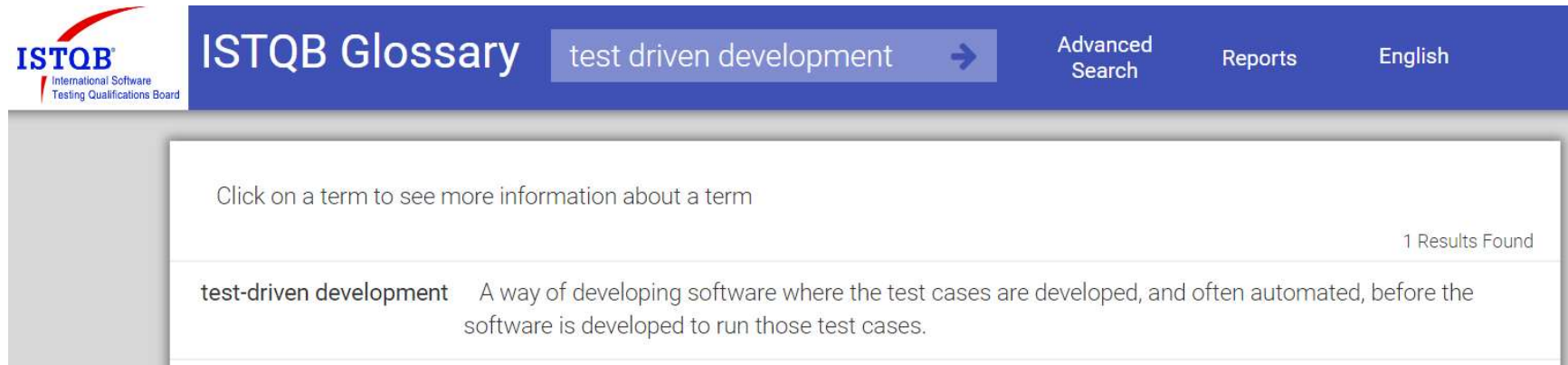
Unit Testing



Test-Driven Development (TDD)



TDD - Test Driven Development



The screenshot shows the ISTQB Glossary search interface. The header includes the ISTQB logo, the title 'ISTQB Glossary', the search term 'test driven development' with a right arrow, and links for 'Advanced Search', 'Reports', and 'English'. Below the header, a message says 'Click on a term to see more information about a term'. On the right, it indicates '1 Results Found'. The result for 'test-driven development' is shown with its definition: 'A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.'

- A way of developing software where the test cases are developed, and often automated, before the software is developed to run those test cases.

Test-Driven Development (TDD)

- Development approach appropriate for unit testing
- The rhythm of Test-Driven Development (TDD) can be summed up as follows:
 1. Quickly add a test.
 2. Run all tests and see the new one fail.
 3. Make a little change.
 4. Run all tests and see them all succeed.
 5. Refactor to remove duplication.

[Kent Beck, Test-Driven Development, Addison-Wesley, 2003]

Test-Driven Development (TDD)

- **Red:** Write a failing test to prove code or functionality is missing from the end product
 - The test is written as if the production code is already working; the test failing means there is a bug in the production code.
- **Green:** Make the test pass
 - By writing production code that fits the reality that your test is expecting. It should be written as simple as possible.
- **Refactor:** Refactor your code
 - When the test passes, you are free to move on to the next unit test or refactor your code to make it more readable, remove code duplication, and more

Refactoring

- **Refactoring** is the act of changing a piece of code without changing its functionality.
 - If you've ever renamed a method, you've done refactoring.
 - If you've ever split a large method into multiple smaller method calls, you've refactored your code. It still does the same thing; it's just easier to maintain, read, debug, and change.

Principles of TDD

- Simplicity
- Minimalist approach
- Test-first
 - *During development* - When you need to add new functionality to the system, write the tests first. Then, you will be done developing when the test runs.
 - *During Debugging* - When someone discovers a defect in your code, first write a test that will succeed if the code is working. Then debug until the test succeeds.
- Focus on needed functionality

Benefits of TDD

- Unit tests “prove” that your code actually works. You get more confidence on the code’s quality
- You get a low-level regression-test suite
- You can improve the design without breaking it. Helps code maintenance
- Unit tests demonstrate concrete progress
- Unit tests are a form of sample code and code documentation
- Test-first forces you to plan before you code
- Test-first reduces the cost of bugs. You find bugs earlier
- Unit tests make better designs
- ...

Benefits of TDD

- The suite of unit tests provides constant feedback that each component is still working.
- The unit tests act as documentation that cannot go out-of-date, unlike separate documentation, which can and frequently does.
- When the test passes and the production code is refactored, it is clear that the code is finished, and the developer can move on to a new test.
- TDD forces critical analysis and design because the developer cannot create the production code without truly understanding what the desired result should be and how to test it.

[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

Benefits of TDD

- The software tends to be better designed (loosely coupled and easily maintainable), because the developer is free to make design decisions and refactor at any time with confidence that the software is still working (by running the tests).
- The test suite acts as a regression safety net on bugs: if a bug is found, the developer should create a test to reveal the bug and then modify the production code so that the bug goes away and all other tests still pass. On each successive test run, all previous bug fixes are verified.
- Reduced debugging time!

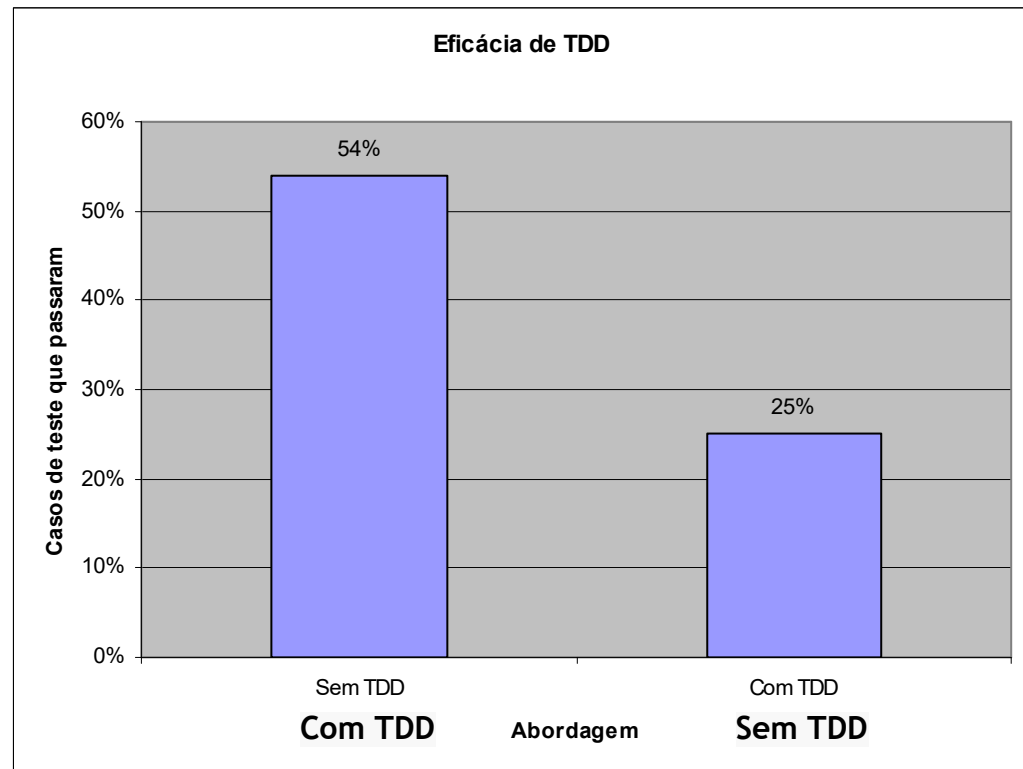
[http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3]

What about architecture and design

- TDD does not replace architecture or design
- You need to do up front design and have a vision
- But, you should not do “Big Up Front Design”
- Defer architectural decisions as long as is “responsibly” possible
- TDD will inform and validate (or invalidate) your design decisions
- TDD will almost undoubtedly uncover weaknesses and flaws in your design.

Is it worth?

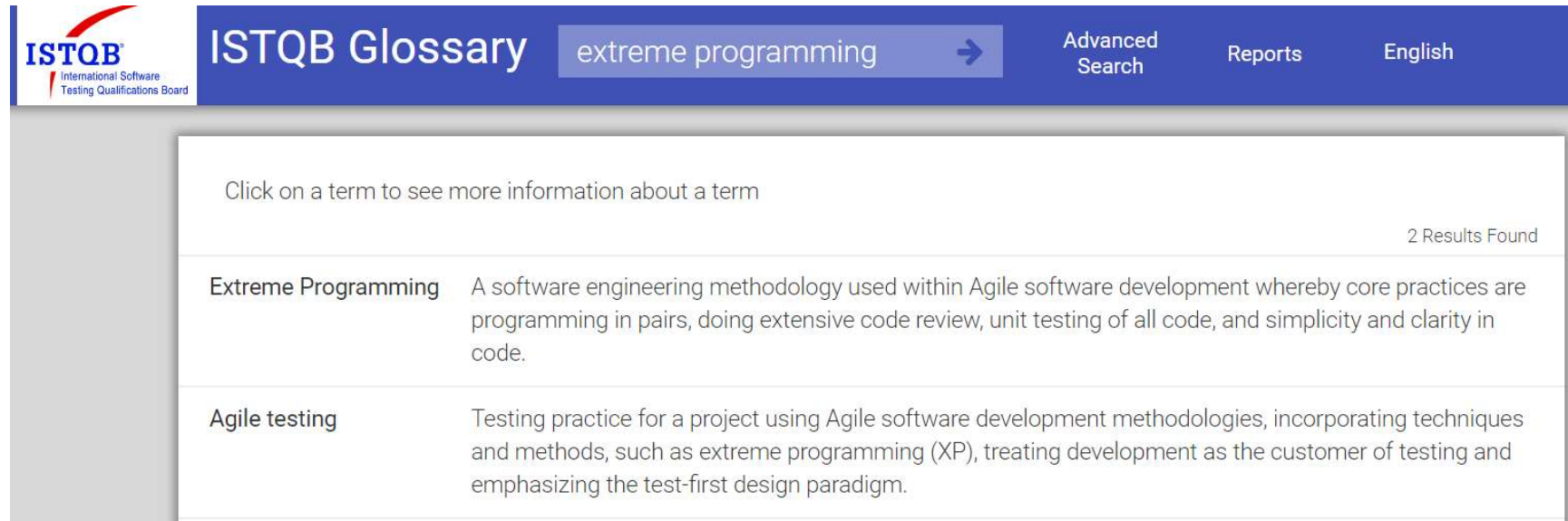
- Experiment with 20 students (TQS, FEUP, 2006/07)
- Convert numbers into words
- Time bounded to one hour
- Half of the students used TDD, the others built tests on the end
- Quality of each solution evaluated with a test suite with 60 tests



With TDD: 54% of the test cases passed

Without TDD: 25% of the test cases passed

Extreme Programming



The screenshot shows the ISTQB Glossary search results for the term 'extreme programming'. The header includes the ISTQB logo, the search term 'extreme programming', and navigation links for 'Advanced Search', 'Reports', and 'English'. Below the header, a message says 'Click on a term to see more information about a term'. The search results show '2 Results Found'. The first result is 'Extreme Programming', described as 'A software engineering methodology used within Agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code.' The second result is 'Agile testing', described as 'Testing practice for a project using Agile software development methodologies, incorporating techniques and methods, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm.'

Term	Description
Extreme Programming	A software engineering methodology used within Agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code.
Agile testing	Testing practice for a project using Agile software development methodologies, incorporating techniques and methods, such as extreme programming (XP), treating development as the customer of testing and emphasizing the test-first design paradigm.

- XP: A software engineering methodology used within Agile software development whereby core practices are programming in pairs, doing extensive code review, unit testing of all code, and simplicity and clarity in code.

eXtreme programming

- Formulated in 1999 by Kent Beck, Ward Cunningham and Ron Jeffries
- Agile software development methodology (other: Scrum)
- Developed in reaction to high ceremony methodologies

eXtreme programming

- XP embrace change knowing that all requirements will not be known at the beginning and will change
- Use tools to accommodate change as a natural process
- Do the simplest thing that could possibly work and refactor mercilessly
- Emphasize values and principles rather than process

eXtreme Programming (XP)

- Extreme Programming (XP) is a set of values, principles, and disciplined practices that focus the entire team on delivery of value and reduction of waste. A typical day for the programming team would be as follows:
 - 15 minute team *stand-up* meeting
 - Get a story from task board
 - Pair-up
 - Talk with the customer sitting with the team
 - Hold programming episodes
 - Monitor the automated build & test system
 - Deliver a few fully tested, integrated business Stories
 - Go home at a reasonable time

eXtreme Programming (XP)

- Although practices used in XP are actually common practices which are shared by other methodologies, XP goes further by doing those practices to an extreme level (hence the “Extreme” Programming). Below is a table on how XP is extreme.

Good practices are	Pushed to the <i>extreme</i>
Code reviews	Review code all the time (pair programming)
Testing	Everybody will test all the time (unit testing) even the customers (functional testing)
Design	Make it part of everybody’s daily business (refactoring)
Simplicity	Always leave the system with the simplest design that supports current functionality (simplest thing that could possibly work)
Architecture	Everybody will work defining and refining the architecture all the time (metaphor)
Integration testing	Integrate and test several times a day (continuous integration)
Short iterations	Make iterations really short-seconds, minutes, hours not weeks, months, years (the planning game)

XP Values

- **Communication** - let everyone know what's happening
- **Simplicity** - don't solve problems you don't have
- **Feedback** - start with the end in mind
- **Courage** - effective Action in the face of fear
- **Respect** - if people don't care, the project will not succeed

[Extreme Programming Explained, by Kent Beck]

References and further reading

- <http://glossary.istqb.org/>
- www.junit.org
- www.nunit.org
- <http://testdriven.net/>
- [IEEE 90] Institute of Electrical and Electronics Engineers. IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries. New York, NY: 1990.
- “The art of unit testing - With Examples in .NET”, by Roy Osherove, ISBN: 1933988274
- xUnit Test Patterns, Gerard Meszaros, Addison-Wesley, 2007
- <http://www.onestoptesting.com>
- Test Driven Development, Kent Beck, Addison Wesley, 2002
- Test-Driven Development in Microsoft.Net, James W. Newkirk, Alexei A. Vorontsov, Microsoft Professional, 2004
- IEEE Software, May/June 2007 (Vol. 24, N. 3), Special issue on TDD
- http://msdn.microsoft.com/en-us/library/aa730844.aspx#guidelinesfortdd_topic3
- “Extreme Programming in the Real World - Java Extreme Programming Cookbook”, by Eric M. Burke & Brian M. Coyner, O'Reilly
- <http://www.umsl.edu/~sauterv/analysis/f06Papers/Hutagalung/#xp>
- <http://www.geocities.com/xtremetesting/TestingDictionary.html>
- Lean, Agile, and Extreme Programm, By Mark Windholtz, ObjectWind Software Ltd, in the RailsStudio.com
- “Extreme Programming Explained: Embrace Change”, By Kent Beck, Published by Addison-Wesley, 2000
- <http://christopher-steiner.com/html/?p=37>