Unit Testing with JUnit

Prof. univ. dr. Catalin Boja
catalin.boja@ie.ase.ro
http://acs.ase.ro

Dep. de Informatică și Cibernetică Economică
ASE București

Other Resources

- Lasse Koskela - *Effective Unit Testing*, Manning, 2013
- Lasse Koskela - *Practical TDD and Acceptance TDD for Java Developers*, Manning, 2007
- Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004
- Clean Code: *Writing Code for Humans* – Pluralsight series
- Robert C. Martin - *Design Principles and Design Patterns*
- Evan Dorn speak on TDD (on youtube.com)
- http://www.junit.org
- https://en.wikipedia.org/wiki/Unit_testing

GOOD CODE

We need a process for software engineering

Understand

Design

Prepare

Write code & test it

GOOD CODERS...

... KNOW WHAT THEY'RE DOING

GOOD CODE

Writing code means also writing the corresponding unit tests – *in the same time*

TESTING SOURCE CODE

Acceptance testing

- Test done by the client. Testing if the design is accordingly with the client needs

System testing

- Testing the entire system (all the components). Is testing if the system works accordingly with the design and the specifications

Integration Testing

- Testing more combined and integrated components

Unit Testing

- Testing the smallest units of the source code (classes or methods)

TESTING SOURCE CODE

Regression Testing

- Testing the app (usually in an automate manner) after implementing changes to the existing code in order to avoid getting old bugs – checking if the changes are affecting the app in a wrong way.

Black-box testing

- Testing the public interface of components without any info regarding their implementation, internal structure, etc

White-box testing (glass-box testing)

- Testing components for which the implementation is known and accessible

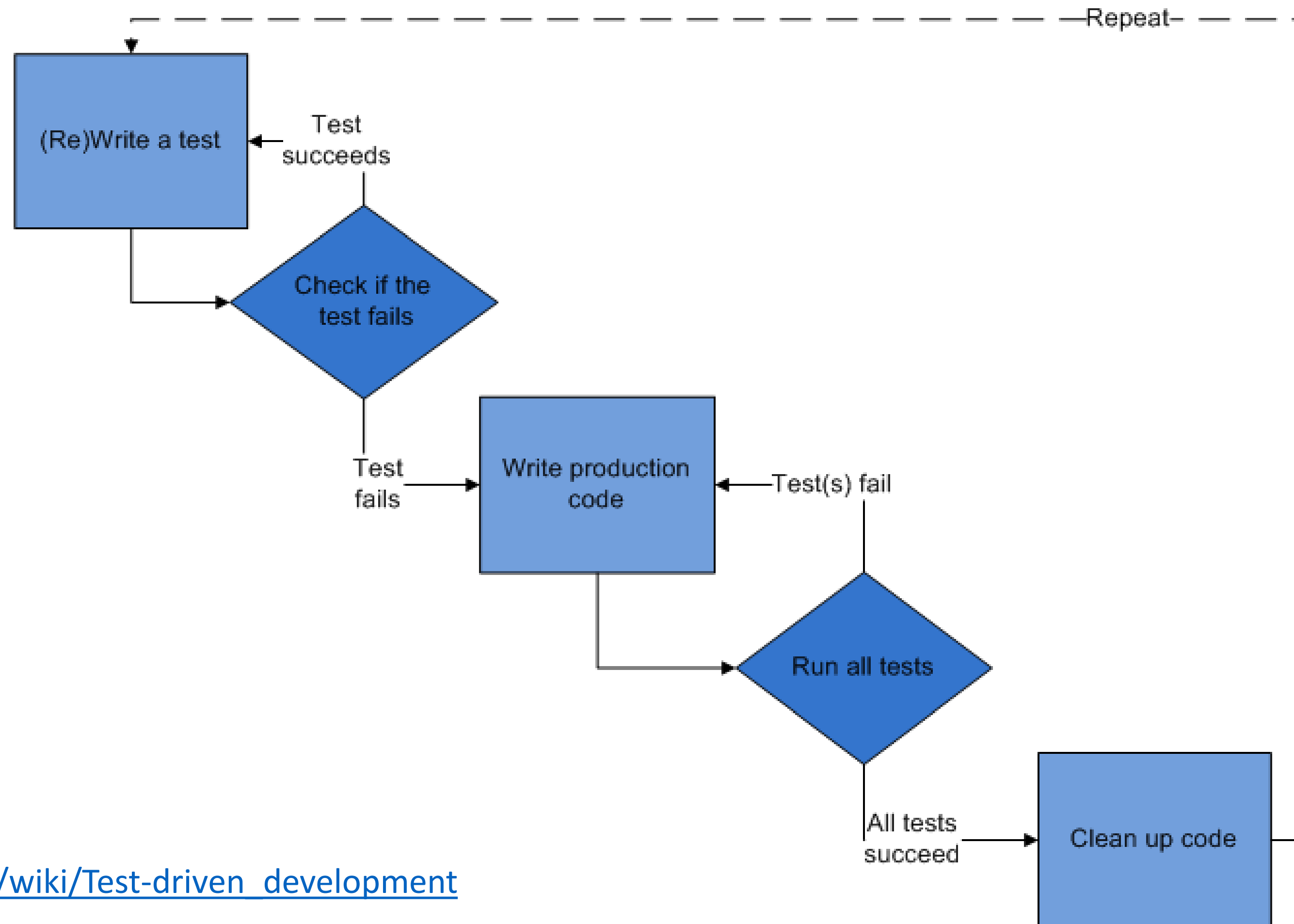
WHAT IS UNIT Testing?

- A simple and rapid method to test source code **for the programmers**
- Used in the development phase
- An instrument **for the programmers**
- A *unit test* is a code sequence written by the programmer to evaluate a reduced size, clear defined code sequence from its tested source code – a class or a method
- A *unit test* assesses how a methods runs in a well defined context (it's all about input and results)
- A *unit test* is the basic block for implementing *Test-Driven Development (Agile Programming, SCRUM)*

What is Test-Driven Development (TDD) ?

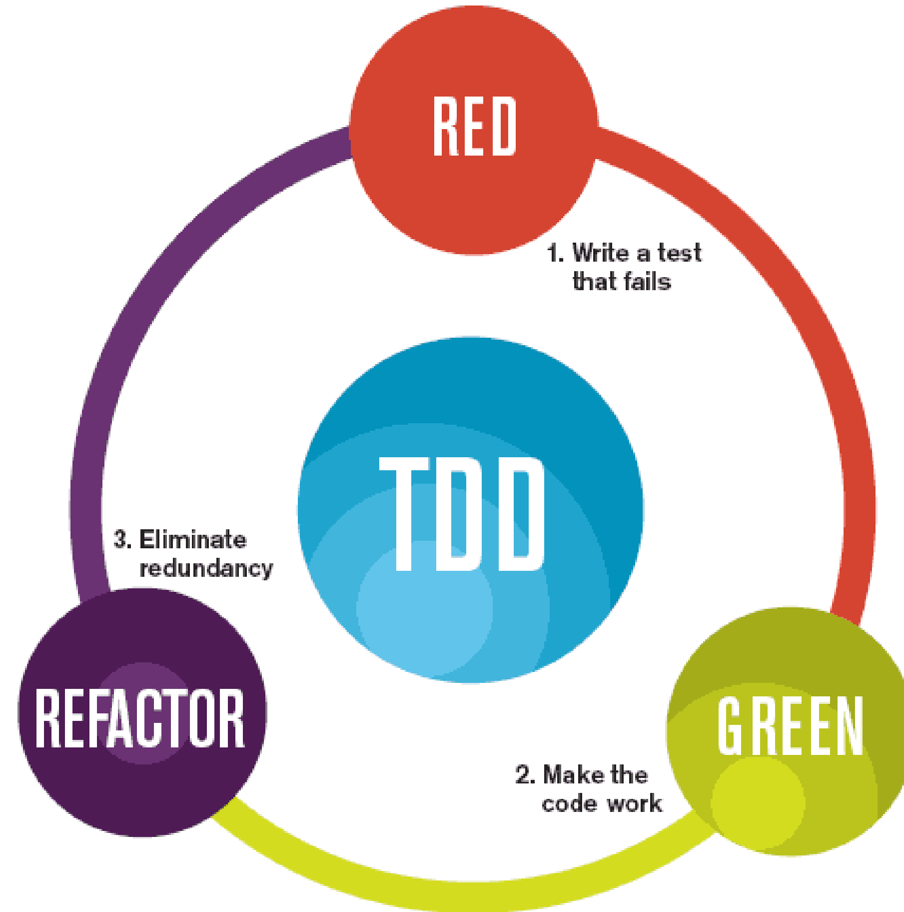
***Test-driven development** (**TDD**) is a software development process that relies on the repetition of a very short development cycle: first the developer writes an (initially failing) automated test case that defines a desired improvement or new function, then produces the minimum amount of code to pass that test, and finally refactors the new code to acceptable standards* [*http://en.wikipedia.org/wiki/Test-driven_development*]

What is Test-Driven Development (TDD) ?



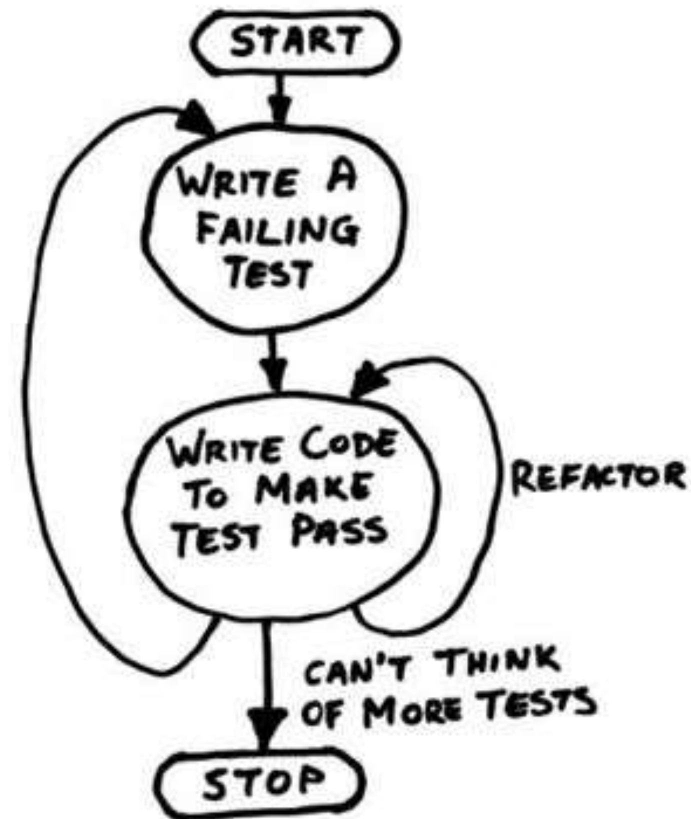
http://en.wikipedia.org/wiki/Test-driven_development

What is Test-Driven Development (TDD) ?



What is Test-Driven Development (TDD) ?

1. Understand the code you need to implement (read specifications)
2. Define the method signature/header
3. Write a unit test for correct conditions
4. Implement the method – dummy body
5. Test the method – it will fail
6. Write the method
7. Test the method – should pass
8. Add more unit tests for extreme conditions



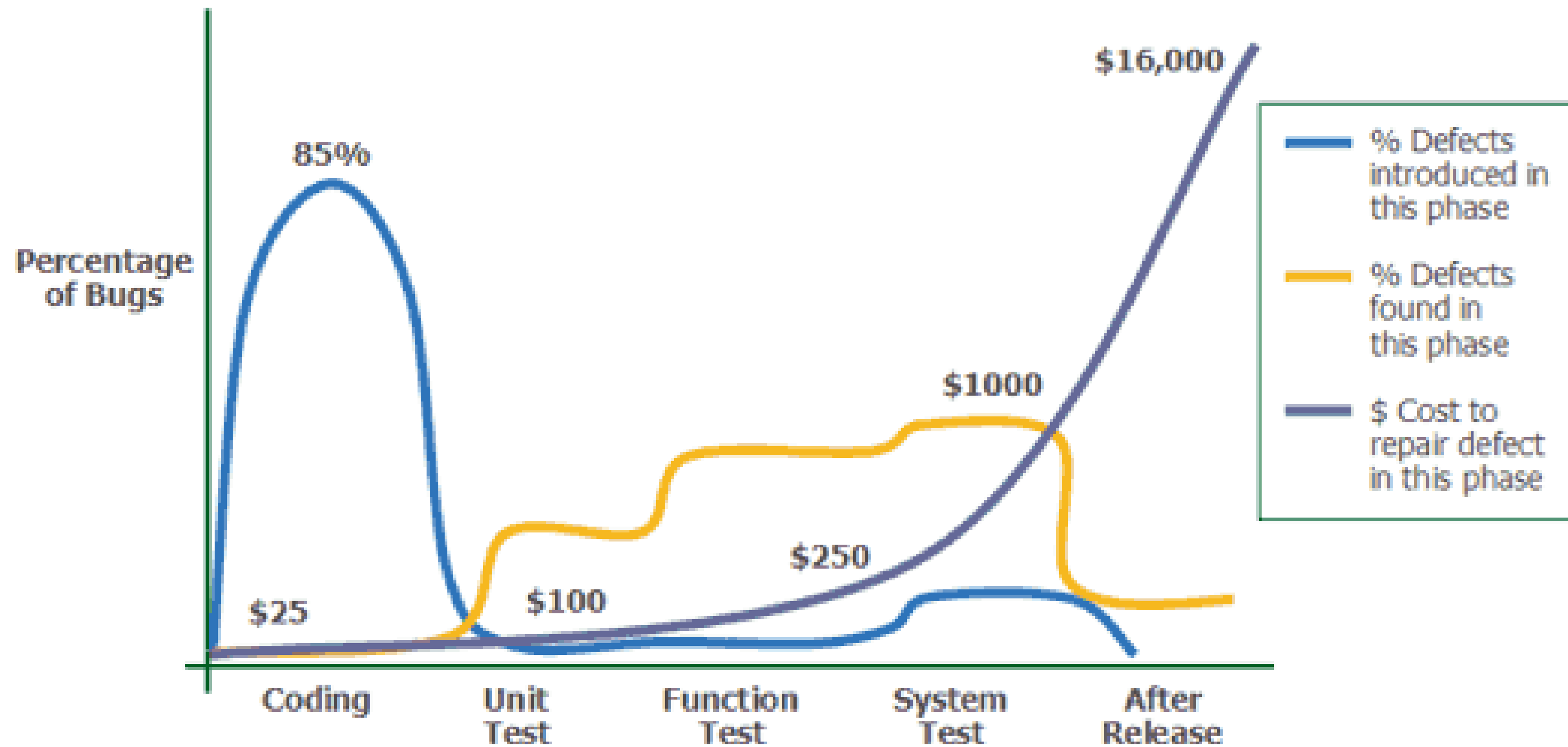
WHY TO USE Unit Testing ?

http://kristianmcfadyen.com/

Source code quality – sources of evil:

- Available time (deadlines) and costs
- Programmer experience
- Programmer know-how
- Clarity and level of detail for Specifications (if ar
- Solution complexity
- Rate of change (specs, requirements, team, etc) – 100% chance to happen

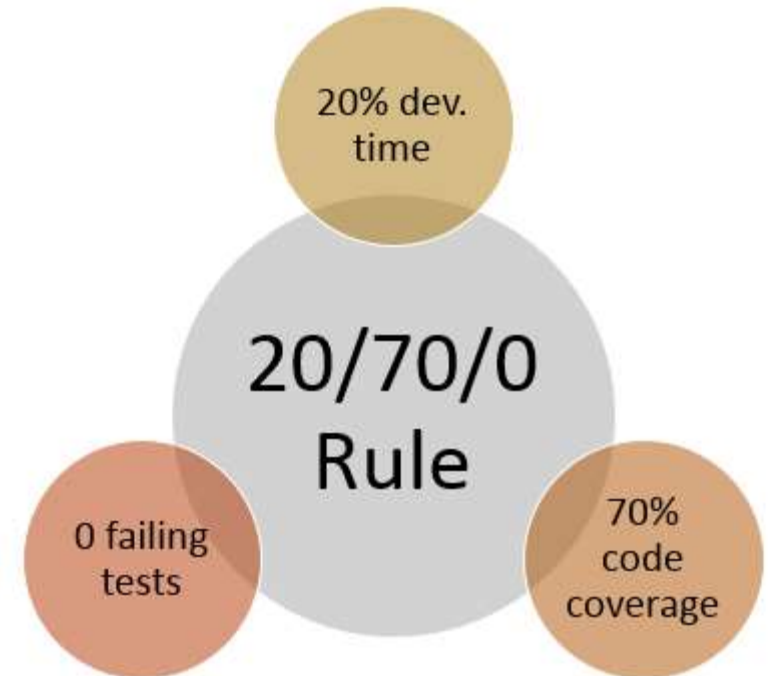
WHY TO USE Unit Testing ?



Source: *Applied Software Measurement*, Capers Jones, 1996

WHY TO USE Unit Testing ?

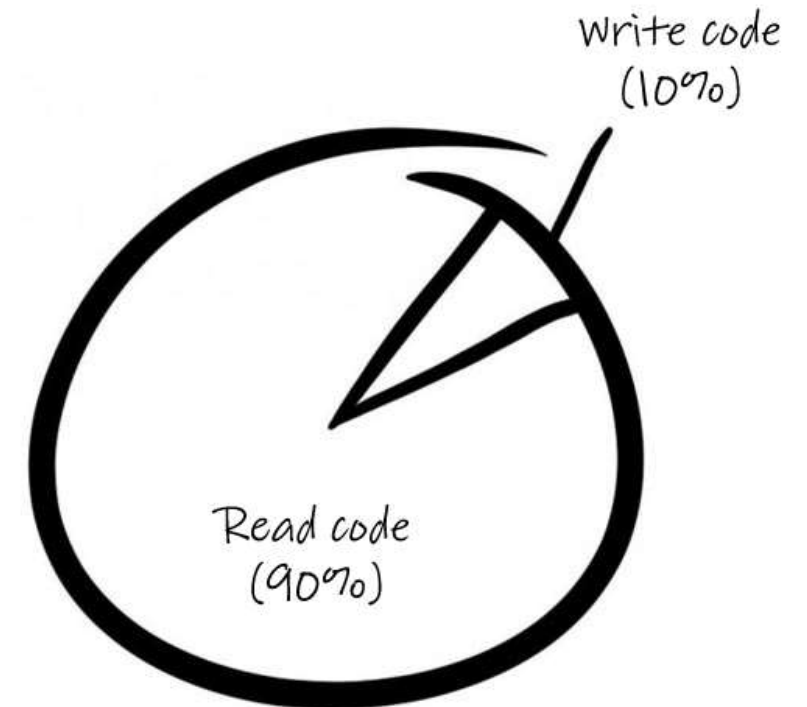
- Easy to write
- Tests can be implemented ad-hoc on a need-now scenario
- Despite there are simple, more complex collections can be defines using **Test Suites**
- Can implement an automate testing process – you can run them anytime you need them (*write once, use many times*)



Source: Anthony Langsworth, Unit Testing: The 20/70/0 Rule

WHY TO USE Unit Testing ?

- There are multiple open-source or proprietary frameworks that can help you write and run them
- Reduce the time for debugging and for manually finding bugs
- Reduce the number of bugs in your delivered or integrated solution
- Increase the rate of finding bugs in the phase of writing the source code



WHY TO USE Unit Testing ?

- Helps documenting the code – reading the tests
- Reduce number of future bugs
- Prevents future modifications that will brake the code
- Prevents writing bad code – *because its hard to test*
- Proves the code works

WHY TO USE Unit Testing ?

Unit testing promotes implementation of clean code principles for writing code

WHY NOT TO USE Unit Testing ?

- Do I really need to test my code ? But is written by me ?
- I don’t have time. I’m busy. I have other things to do (write new code)
- It’s hard
- I don’t know how to do it
- It’s not in the specifications
- It’s not my job. Get a tester

http://www.dylanmeconis.com/how-not-to-write-comics-criticism/

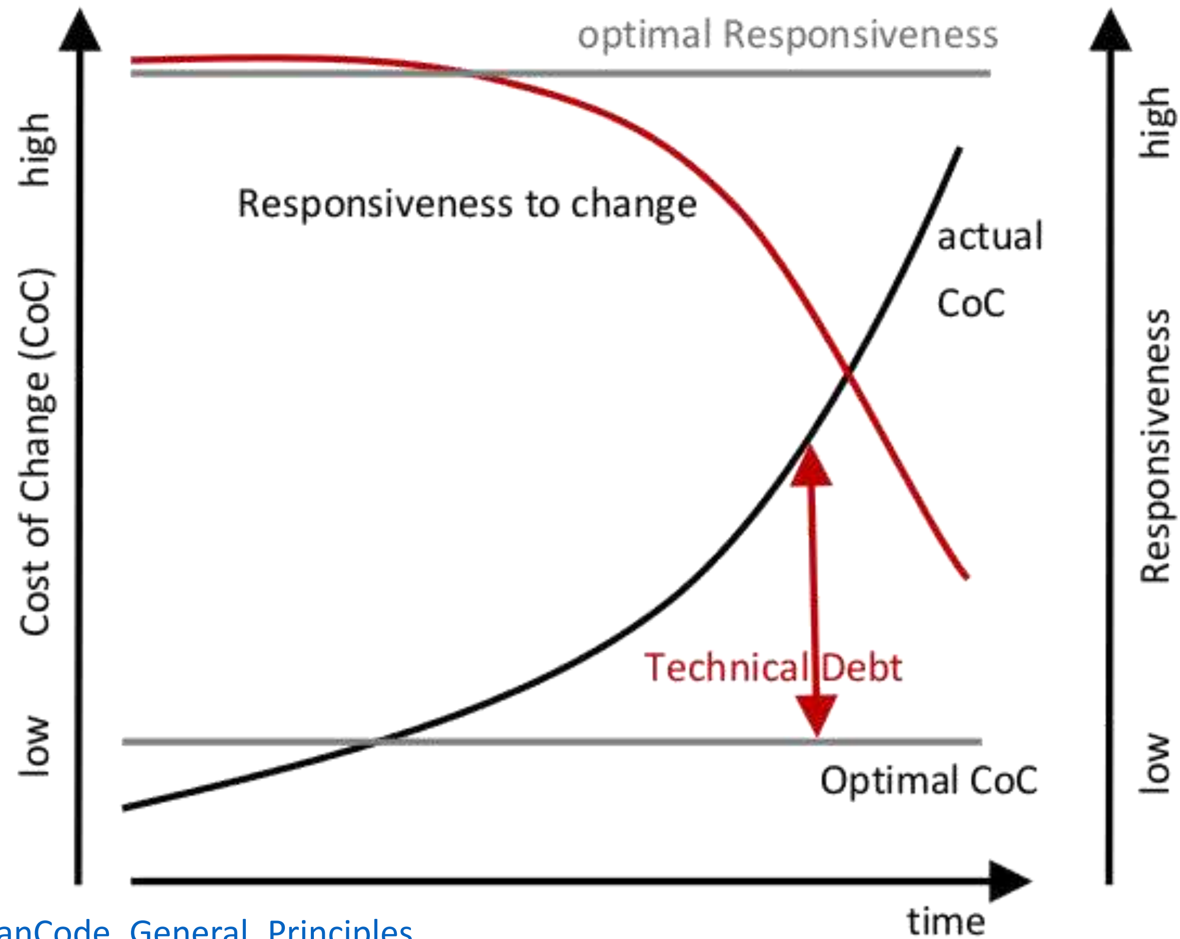
Bad code

- Confusing
- Hard to read and understand
- Breaks stuff when you modify it
- Has dependencies in multiple modules – glass breaking code
- Tight coupled to other stuff

Bad code

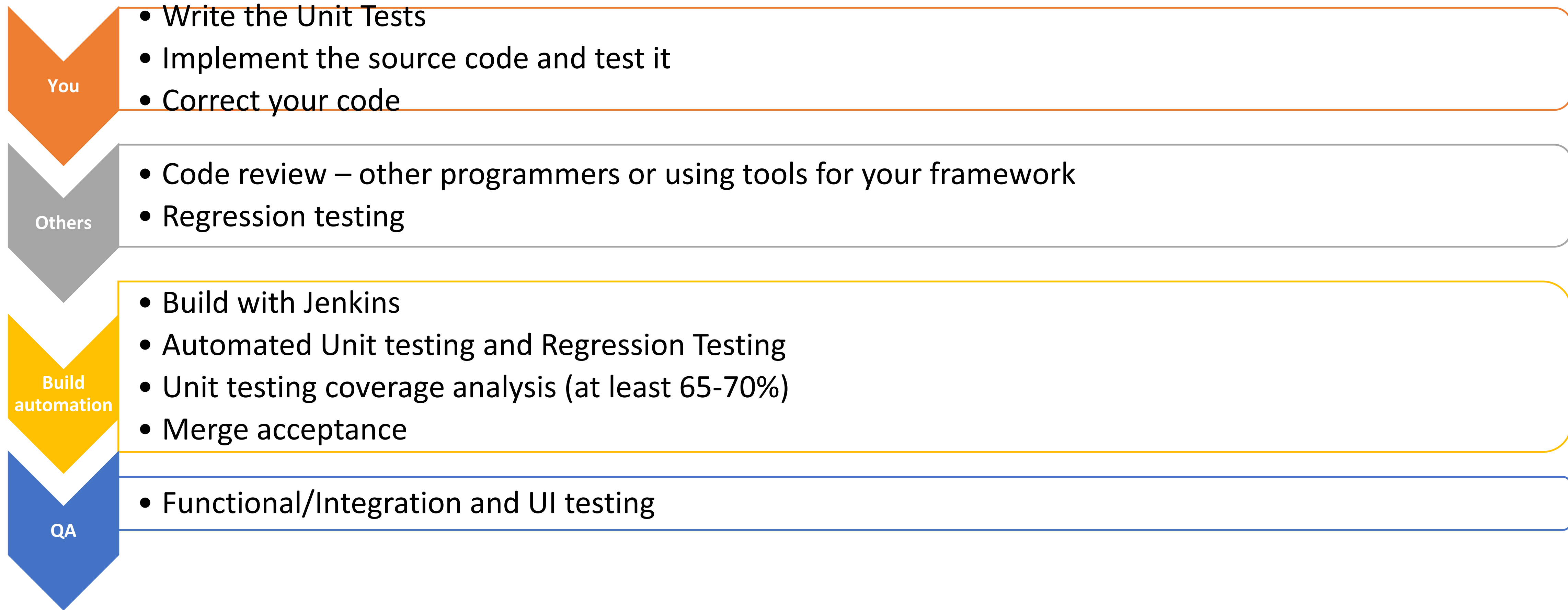
Bad code (fast, rapidly deployed code) slows you down (Bob Martin)

BAD CODE vs. clean code



http://www.chudovo.com/Blog/CleanCode_General_Principles

UNIT TESTING in production – The process



HOW to implement UNIT testing ?

The golden path:

1. Read the specs for a method or a class
2. Understand the problem
3. Define conditions to evaluate (rewrite the specs)
4. Write tests that will fail if the method/class is not implemented correctly
5. Implement the code for the method/class
6. Run the tests
7. If they fail go back to step 1

WHAT TO Test?

- Real scenarios
- Limits of the input values interval
  - Smaller than 0 such as –1, -2, …, -100, …
  - 0
  - Bigger than 0 such as 1, 2, …, 100…
  - Null
  - CORRECT Boundary Conditions [Alasdair Allan - *Pragmatic Unit Testing in Java with Junit*]
- The Right-BICEP [Alasdair Allan - *Pragmatic Unit Testing in Java with Junit*]

CORRECT Boundary Conditions

- *Conformance* – Has the value the correct format/ Is the value valid ?
- *Ordering* – The set of values must be ordered or not ?
- *Range* – Is the value between acceptable limits (maxim and minim) ?
- *Reference* – Is the code referring external components which are NOT directly controlled?
- *Existence* – Does the value exists (ex. non-null, non-zero, parte of a set, etc.)?
- *Cardinality* – Does the set of values has enough values?
- Time (absolute or relative) – Is everything happing in order? At the right moment? In a limited time?

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004

CORRECT Boundary Conditions

- Totally bogus or inconsistent input values, such as a one of "!*W:Xn&Gi/w>g/h#WQ@".
- Badly formatted data, such as an e-mail address without a top-level domain ("fred@foobar.").
- Empty or missing values (such as 0, 0:0, "", or null).
- Values far in excess of reasonable expectations, such as a person's age of 10,000 years.
- Duplicates in lists that shouldn't have duplicates.
- Ordered lists that aren't, and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance or even a reverse-sorted list.
- Things that arrive out of order, or happen out of expected order, such as trying to print a document before logging in, for instance.

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004

The Right-BICEP

- Right . Are the results right for the right input?
- B . Are all the boundary conditions CORRECT?
- I . Can you check inverse relationships?
- C . Can you cross-check results using other means?
- E . Can you force error conditions to happen?
- P . Are performance characteristics within bounds?

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004

Error - conditions

- Running out of memory
- Running out of disk space
- Issues with wall-clock time
- Network availability and errors
- System load
- Limited color palette
- Very high or very low video resolution

TESTING BRANCHES

- Number of test depends on the method cyclomatic complexity value
- Branch Testing/ Condition Coverage – implement enough tests to assure that every branch alternative has been executed at least once under some test; for every branch alternative you provide values that generate true and false results

TESTING EXTERNAL functions

- DON’T test them
- Use mocks or stub methods to replace them
- Use clean code principles (like SOLID) to refactor the tested method
  - Split it in more simples methods
  - Use interfaces and functions pointers to remove external dependencies

Mocks – WHAT ARE?

- A unit test assesses a method execution but this requires objects/methods/conditions external to the method
- It's a *testing pattern*
- It a simple replacer for the real object/method
- It has the same interface as the real method/object
- The are frameworks for different languages that generates mocks: for Java http://easymock.org/

Mocks – when to use them?

- The real object has nondeterministic behavior (it produces unpredictable results; as in a stock-market quote feed.)
- The real object is difficult to set up.
- The real object has behavior that is hard to trigger (for example, a network error).
- The real object is slow.
- The real object has (or is) a user interface.
- The test needs to ask the real object about how it was used (for example, a test might need to check to see that a callback function was actually called).
- The real object does not yet exist (a common problem when interfacing with other teams or new hardware systems).

SCENARIO - A GOOD method

```
int sum(int* values, int no){
        int result = 0;
        for(int i = 0; i<no; i++){
                result+=values[i];
        }
        return result;
}
```

Computing the sum of an array with signed integer values

Implemented in C

It’s simple, right ?

SCENARIO - UNIT Testing

1. unit test for NORMAL conditions with values = {1,2,3}, no = 3; expected 6
2. unit test for EXTREME conditions with values = null, no = 0; expected exception or error code
3. unit test for EXTREME conditions with values = {1,2,3}, no = 0; expected exception or error code
4. unit test for WRONG conditions with values = {1,2,3}, no = -3; expected exception or error code
5. unit test for LIMIT conditions with values = {INT_MIN,-23}, no = 2; expected exception

SCENARIO – UNIT TESTING

6. unit test for LIMIT conditions with values = {INT_MIN,-23}, no = 2; expected exception
7. Cardinality unit test with values = {3}, no = 1; expected result 3
8. Ordering unit test (not necessary) with values = {3,2,1}, no = 3; expected 6
9. Buffer overflow unit test (allowing max 1000 values) with values = {1,2,3…}, no = 1001; expected exception

SCENARIO - A GOOD method

in C you don't have exceptions or you are not allowed to throw exceptions

Method returns an error code each time a condition is not verified

Is the same method

```
int sum_final(int* values, int no, int* result) {
		//testing limits and special cases
		if (values == NULL || no <= 0 || no > 1001) {
				return ERROR_CODE;
		}

		int value = 0;
		for (int i = 0; i < no; i++) {
				//testing for overflow
				if ((value > 0) && (values[i] > INT_MAX - value)){
						//will overflow
						return ERROR_CODE;
				}
				if ((value < 0) && (values[i] < INT_MIN - value)){
						//will overflow
						return ERROR_CODE;
				}

				//do the sum
				value += values[i];
		}

		*result = value;
		return SUCCESS_CODE;
}
```

GOOD methods properties

- Good methods are implemented by writing unit tests
- Are simple to understand
- Have low cyclomatic complexity (optimum is 1) -> less branches
- Have a single responsibility (remember SOLID)
- Don’t depend on external methods (only on interfaces or function pointers)
- Are small – one screen rule

Good vs Bad methods

Good method (all the previous props)

Do more than ONE thing

Add MORE lines of code

Increased COMPLEXITY

More DEPENDENCIES

BAD method

https://ghostlypineapples.wordpress.com

Simple rules for good methods

- Write unit tests while implementing the method
- Single responsibility
- Keep It Simple & Stupid
- Delegate
- Use interfaces

Tools - UNIT TESTING FRAMEWORKS

- **AppleScript:** ASUnit, ASTest
- **C/C++:** Boost Test Library, Google Test, QT Test, unity
- **Java**: JUnit
- **C#**: Nunit
- **Python**: unittest

https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

UNIT TESTs characteristics

- *simple – very simple*
- *fast*
- ***each test should check one condition – one assert rule***
- *independent* and *repeatable (use mockups or stubs)*
- *portable* and *reusable*

UNIT TESTs characteristics

- well *organized*
- reflect the structure of the tested code
- related tests are grouped into Test Cases/Suites
- should provide as much *information* about the problem as possible

Unit testing general concepts

- ***assertions***, which are statements that check whether a condition is true
- a ***test case*** contains one or many tests
- a ***test suite*** is a collection of one or more test cases
- common objects and subroutines are put into a ***test fixture*** class
- when an assertion fails, it prints the file name, line number, and a message that you can customize
- assertion's result can be ***success***, ***nonfatal failure***, or ***fatal failure*** *(*it aborts the current function)

Unit testing general concepts

Test Fixture (objects and resources)

Test Suite(collection of tests)

- setUp

Test Case (unit tests)

- setUp
- tearDown

- tearDown

Tests properties

- **Automatic -** can be run automatically
- **Repeatable -** they can be executed many times with the same results
- **Independent** - independent of performance-specific conditions and independent of other tests
- **Complete -** tests everything that can generate errors at a time
- **Real -** they are just as important as the test code; Are all programs and must be treated as such (NOT coded sequences for a specific moment and then discarded)

Tests properties

FIRST Principles

- Fast
- Isolated/Independent
- Repeatable
- Self-Validating
- Thorough

- “Pragmatic Unit Testing in Java 8 with JUnit” by *Andy Hunt; Jeff Langr; Dave Thomas*.

What is JUnit

- A tool for Test-Driven Development
- JUnit is a class framework that allows writing and executing tests for differer methods / classes in the code

History

- Kent Beck developed the first automated test tool, xUnit, for Smalltalk in the 1990s
- Beck and Gamma (Gang of Four) developed JUnit during a flight from Zurich to Washington, D.C.
- Martin Fowler: "*Never in the field of software development was so much owed by so many to so few lines of code*."
- Junit has become the standard tool for TDD - Test-Driven Development in Java
- Junit is the standard component in multiple Java IDEs (Eclipse, BlueJ, Jbuilder, DrJava, InteliJ)
- Xunit tools have been developed for other languages (Perl, C ++, Python, Visual Basic, C #, ...)

Junit Architecture

- **TestRunner** executes tests and reports **TestResult**s
- Test are implemented in classes that extend abstract ***TestCase***
- To write tests you need to use **Assert**
- Each *assert* method has next arguments: *message, expected-value, actual-value*
- For floating point values *assert* has a supplementary argument – delta (margin error)
- Each *assert* method has an equivalent version without the message – not recommended to use
- The *assert* message is used to document tests and is usefull for debugging failed tests

Assert type methods

- assertEquals(expected, actual)
- assertEquals(message, expected, actual)
- assertEquals(expected, actual, delta)
- assertEquals(message, expected, actual, delta)
- assertFalse(condition)
- assertFalse(message, condition)
- Assert(Not)Null(object)
- Assert(Not)Null(message, object)
- Assert(Not)Same(expected, actual) – verifica daca cele 2 referinte (NU) sunt identice
- Assert(Not)Same(message, expected, actual)
- assertTrue(condition)
- assertTrue(message, condition)
- fail(message)

Junit Concepts

- **Fixture** – Set of objects used in the test
- **Test Case** – A class that defines the fixture set to run multiple tests
- **Setup** – A method / step of defining the set of objects used (fixture) before testing.
- **Teardown** – A method / stage of destroying the fixture after completion of the tests
- **Test Suite** – Collection of test cases
- **Test Runner** – tool for running tests (test suite) and to display results

Junit Characteristics

- The function package is quite simple
- JUnit code is not delivered to the client at build / export
- Tests are written in Java
- If a test fails the others are still executed
- The test has a simple structure:
  - Prepares initial conditions (create objects, resource initialization)
  - Call the methods to be tested
  - Check if the method works by comparing the results generated with the expected ones
  - Release the resources used

Junit Characteristics

- One test method can have several assertions
- Simplifying tests helps to quickly identify bugs
- The objective is to write unsuccessful tests so that errors are corrected in time
- Tests can be combined in collections - Test Suites

TestCase tutorial – JUnit3

1. Define a subclass for *TestCase*

```
import junit.framework.TestCase;
public class TestMath extends TestCase {
} //
```

- The class name is important and must be in the form - ***Test****MyClass* or *MyClass****Test***
- The naming convention allows the *TestRunner* to automatically identify classes used for testing

TestCase tutorial – JUnit3

2. The class defines a constructor by which certain test methods can be selected (Test Suites)

```
import junit.framework.TestCase;

   public class TestMath extends TestCase{

      public TestMath(String name){
      super(name);
      }

   }
```

TestCase tutorial – JUnit3

3. In the class that extends *TestCase* there are defined tests that use *assert* methods to evaluate the way of execution of the methods in the tested class

Recommendations:

A unit test should evaluate maximum one method

- Each unit test should evaluate a well-defined scenario - a maximum assert per unit test
- Unit tests must be independent (can be easily evaluated and can be inserted into *TestSuites*)

TestCase tutorial – JUnit3

4. In the class that extends TestCase tou define needed resources (fixture) and you implement setUp() and tearDown()

```
public class TestStudent extends TestCase {
//fixture pt TestCase
String examen = null;
Student stud= null;

//setUp – executed before each test
public void setUp(){
System.out.println("Apel setUp");
examen = "CTS";
stud= new Student();
}

//tearDown – executed after each test
public void tearDown(){
System.out.println("Apel tearDown");
}
```

TestCase tutorial – JUnit3

5. Tests can be combined by defining external collections that load all test methods or just selected ones.

```
public class TestComposite extends TestCase {
          public TestComposite(String name){
                    super(name);
          }

static public Test suite(){
          TestSuite suite = new TestSuite();

          //loads all tests
          suite.addTestSuite(TestLargest.class);
          //loads only one method
          suite.addTest(new TestMath("testMedie"));

          return suite;
          }
}
```

TestCase tutorial – JUnit3

6. You can define also global **setUp()** and **tearDown()** methods

```
//setUp global
public static void setUpGlobal(){
System.out.println("setUp GLOBAL");
}

//tearDown global
public static void tearDownGlobal(){
System.out.println("tearDown GLOBAL");
}

public static Test suite(){
TestSuite wrapper = new TestSuite();
wrapper.addTestSuite(TestStudent.class);

TestSetup setupGlobal = new TestSetup(wrapper) {
public void setUp(){
        TestStudent.setUpGlobal();
    }
public void tearDown(){
        TestStudent.tearDownGlobal();
    }
};

return setupGlobal;
}
```

TestCase tutorial – JUnit3

7. You can define your own *assert* methods by defining a base test type that will be extended in other tests

Test Case Structure

- setUp()
  - Defines and builds the resources (*fixture*) needed to run the tests
  - Call before each test method
  - In the case of test kits, the method may be a generic, one-time execution if defined within a *TestSetup*
- tearDown()
  - Releases / destroys the resources allocated to the test
  - Call for each test method
  - In the case of test kits, the method may be a generic, one-time execution if defined within a TestSetup

Test Suites – JUnit3

- Collection of tests to be evaluated together
- Unit test defined by total or partial combination of other test cases
- Unit tests must be defined without being related

- Partial collection – are executed in the order there are added to the suite:

```
public static Test suite() {
        suite.addTest(new TestMath("testMedie"));
        suite.addTest(new TestMath("testSuma"));
        return suite;
}
```

- TestSuite that adds all unit tests from other test cases:

```
public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(TestMath.class);
        suite.addTestSuite(TestStudent.class);
        return suite;
}
```

Junit 4

- Uses annotations to mark framework specific methods
- *@Test* – marks a method as a unit test
- *@Before* – marks the setUp method (the method name must be setUp)
- *@After* – marks the tearDown method
- *@BeforeClass* – marks the setUpBeforeClass method
- *@AfterClass* – marks the tearDownAfterClass method

Junit 4

- *@Ignore / @Ignore("Is disabled")* – ignores the unit test
- *@Test (expected = Exception.class)* – assesses if the tested method is throwing an exepction
- *@Test(timeout=100)* – assesses if the unit test finishes before the timeout (used for Performance tests)

Test Suites – Junit 4

@IncludeCategory(NeedTest.class) - marks the class or method to be included in the partial suite

@IncludeCategory({Type1.class, Type2.class, }) – you can include multiple categories in the suite

@ExcludeCategory(NeedTest.class) - marks the class or method to be excluded from the partial suite

NeedTest.class – a class defined only to be used to define a category (the class does not have a specific implementation)

TestSuitePartial – should not include @Test methods

Test Suites – Junit 4

```
@RunWith(Categories.class)
@IncludeCategory(NeedTest.class)
@Suite.SuiteClasses({
  TestCase1.class,
  TestCase2.class
})
public class TestSuitePartial {

}
```

Other resources JUnit4

- http://www.tutorialspoint.com/junit/junit_suite_test.htm
- http://www.vogella.com/tutorials/JUnit/article.html
- http://junit.org/junit4/
- https://junit.org/junit5/

Recommended reading

Andrew Hunt, David Thomas - *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2004