09 - XUnit Testing Patterns

Prof. Peter Sommerlad
IFS Institute for Software
HSR Rapperswil
Learning goals

The student...

- appreciates the importance of automated builds and testing
- applies automated tests at different levels (unit, functional, system)
- knows four phases of an automated test and create test accordingly
- understands how to improve test coverage by using test stubs and mock objects
- can design software for greater testability
- can detect "test smells" that imply refactoring tests for greater simplicity, robustness, and execution speed.
Is that Testing?

- “it compiles!”
  - no syntax error detected by compiler
- “it runs!”
  - program can be started
- “it doesn’t crash”
  - ... immediately with useful input
- “it runs even with random input”
  - the cat jumped on the keyboard
- “it creates a correct result”
  - a single use case is working with a single reasonable input
What is Testing?

- All on the previous slide, but much more!
  - **Manual Testing**
    - sometimes useful and needed
      - UI testing, usability testing, user testing with a plan
    - but automation is much better!
      - no ad-hoc testing!
  - **Automated Testing**
    - unit tests
    - functional tests
    - integration, load and performance tests
    - code quality tests (lint, compiler, code checkers)
Unit Testing

• Is not “Testing” in the classic sense:

Program testing can be used to show the presence of bugs, but never to show their absence! - E.W. Dijkstra

But

• Is Built-In Quality Assurance
• Allows Regression Testing
• Enables Refactoring
• Is Change Insurance
• Improves Built Automation
Vicious Circle: Manual Testing - Stress

- Automate tests and run them often!
Test Automation

- **Advantages**
  - repeatability - regression
    - insurance for change, portability, extension
    - no (or very low) cost for re-testing
  - well-defined specification given by executable tests
    - test-code is program code with well defined semantics
  - repeatability, repeatability, repeatability, ...

- **Drawbacks**
  - need to write and maintain also test code
    - tests also require refactoring
  - test code is program code
    - is the right thing tested? (instead of implemented?)
Why and When?

- Become “test-infected”. Once you are used to unit testing your code, you get addicted.
  - That’s a fact I observed many times.
  - You’ll regret every piece of code you want to change where you don’t have tests for

- Write your tests close to writing your code!
  - Some say: Test-First or Test-Driven Design (TDD)
  - modern: Behavior-Driven Design (BDD)
  - Retrofitting existing code with tests will show you its design deficiencies
    - hard to write tests -> entangled design, too complex
    - easy to write tests -> orthogonal design, simpler

- At least write tests before you change code!
How do I write good Unit Tests (GUTs)?

- Ask yourself the following questions:
  (among others about your coding)

- If the code is correct, how would I know?
- How can I test this?
- What else could go wrong?
- Could a similar problem happen elsewhere?

- Code coverage tools help for seeing what code is really tested --> e.g. eclEMMA
Writing good automated tests is hard.

Beginners are often satisfied with “happy-path” tests

- error conditions and reactions aren’t defined by the tests
  - coverage tools help here!

Code depending on external stuff (DB, IO, etc) is hard to test. How can you test it?

Will good tests provide better class design?

How can tests be designed well?
Principle of Automated Tests
Triple-A (AAA)

1. **Arrange**
   - initialize object(s) under test

2. **Act**
   - call functionality that you want to test

3. **Assert**
   - assert that results are as you expect

Remember: "Triple-A: arrange, act, assert"
SUT system under test
**Test Case Structure: Four Phase Test**

- Setup
- Exercise
- Verify
- Teardown

- compare that to AAA ---> another similarity
- **Source:** xunitpatterns.com
How many unit tests should I write?

- Test anything that might break
  - don’t write tests for code that cannot break
- Test everything that does break
  - for every bug, write a test demonstrating it
- New code is guilty until proven innocent
- At least as much test code as production code
- Run local tests with each compile
  - don’t write new code when tests are failing
- Run all tests before check-in to repository
  - also run them after check-in on your build server
Use your Right-BICEP [PragProg]

- Are the results right?
  - ASSERT_EQUAL(42, 7*6)

- Are all boundary conditions CORRECT?
  - 0, 1, 0xffffffff

- Can you check inverse relationships?
  - sqrt(x)*sqrt(x) == x

- Can you cross-check results using other means?

- Can you force error conditions to happen?
  - y/x, x=0

- Are performance characteristics within bounds?
CORRECT
Boundary Conditions

• Conformance
  o e.g., check email address: foo@bar.com

• Ordering
  o is sequence relevant? what if out of order?

• Range
  o is the domain range correct

• Reference
  o expectations on environment

• Existence
  o is some parameter/variable defined, null, existent

• Cardinality
  o off-by one errors, 0,1, many

• Time
  o sequencing of actions, concurrency
Test Fixtures

- Often several test cases require identical arrangements of tested objects

- Reasons
  - "expensive" setup of objects
  - no duplication of code (DRY principle)

- Mechanisms
  - JUnit 3 provides setup() and teardown() methods
  - JUnit 4 corresponding @Before @After annotations for these fixture methods
    - and @BeforeClass, @AfterClass for static methods before/after all tests in the current class
Test-Driven Development

Exploiting Unit Tests...
There are several books on test-driven design (or TDD)
- Kent Beck, Dave Astels, Gerard Meszaros

TDD is not a testing technique, but a coding and design technique
- nevertheless TDD patterns help you writing tests, regardless if you follow TDD or not

TDD relies heavily on Refactoring
- we at IFS try hard to provide you with such Refactoring automation for C++ as well as you might be used to with Java or Ruby. (plus Refactoring for Groovy, Python (PyDev), PHP, JavaScript)
TDD
[Kevlin Henney]

- TDD has emerged from the many practices that form Extreme Programming's core
  - Focused on code-centric practices in the micro process rather than driving the macro process

- TDD can be used in other macro-process models
  - TDD is not XP, and vice versa
  - TDD is not just unit testing

- BDD (Behavior Driven Design)
  - Follow-up to TDD
  - since TDD is not about Testing but specifying behavior
TDD Practices and Characteristics

**Essential Test-Driven Development Practices**
- Test-bounded design increments
- Programmer testing responsibility
- Active test writing
- Refactoring
- Automated tests
- Example-based test cases

**Build and Release Practices**
- Fine-grained versioning
- Continuous integration
- Defined stable increments

**Team-Related Practices**
- Pair programming
- Shared coding guidelines

provided by [Kevlin Henney]
TDD Patterns
Writing Tests & Habits

- **Isolated Tests**
  - write tests that are independent of other tests

- **Test List**
  - use a list of to-be-written tests as a reminder
  - only implement one failing test at a time

- **Test First**
  - write your tests before your production code

- **Assert First**
  - start writing a test with the assertion
  - only add the acting and arrangement code when you know what you actually assert
"Red-bar" Patterns
Finding Tests to write

- **One Step Test**
  - solve a development task test-by-test
    - no backlog of test code, only on your test list
    - select the simplest/easiest problem next

- **Starter Test**
  - start small, e.g., test for an empty list
  - refactor while growing your code

- **Explanation Test**
  - discuss design through writing a test for it

- **Learning Test**
  - understand existing code/APIs through writing tests exercising it
“Green Bar” - Patterns
Make your Tests succeed

- **Fake It ('Til You Make It)**
  - It is OK to “hack” to make your test succeed.
  - Refactor towards the real solution ASAP

- **Triangulate**
  - How can you select a good abstraction?
  - Try to code two examples, and then refactor to the “right” solution

- **Obvious Implementation**
  - Nevertheless, when it’s easy, just do it.

- **One to Many**
  - Implement functions with many elements first for one element (or none) correctly
"Red Bar" Patterns (2)

- **Regression Test**
  - For every bug report write tests showing the bug

- **Break**
  - Enough breaks are essential. When you are tired you lose concentration and your judgement gets worse. This results in more errors, more work, and makes you more fatigue. (vicious circle!)

- **Do Over**
  - If you recognize your design and tests lead nowhere, DELETE your code! A fresh start earlier is often better.
• **generate roman numbers as strings from an integer representation**
  o start with the following list of tests
  o write test, implement function, refactor, repeat
  o make up new tests as you go and see need

---

**THE LIST FOR ROMAN NUMBERS (V0)**

<table>
<thead>
<tr>
<th>Integer</th>
<th>Roman</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>I</td>
</tr>
<tr>
<td>0</td>
<td>EMPTY STRING</td>
</tr>
<tr>
<td>2</td>
<td>II</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>
Demo/Exercise TDD V2

(3+4)*6 → 42

- Expression Evaluator for simple Arithmetic
- Test-First Development with CUTE
- Incremental Requirements Discovery

The List for Eval (V0)

"" → error
"0" → 0
"2" → 2
"1+1" → 2
TDD Patterns Habits

- **Child Test**
  - If a test case gets too large, “remove” it, redo the core, get “green-bar”, and then introduce the “full” case again, get “green-bar”

- **Broken Test**
  - If you have to stop programming or take a break, leave a broken test to remind you where you left.
    - ➢ but only do Clean Check-in!

- **Clean Check-in**
  - Do only (and may be always) check-in your code and tests when you have a green bar.
Test-Doubles

Testing the untestable ...
Test Double Pattern

xunitpatterns.com

- How can we verify logic independently when code it depends on is unusable?
- How can we avoid Slow Tests?
Principle of Test Doubles

- A unit/system under test (SUT) depends on another component (DOC) that we want to separate out from our test.

- Reasons
  - real DOC might not exist yet
  - real DOC contains uncontrollable behavior
  - want to test exceptional behavior by DOC that is hard to trigger
  - using the real DOC is too expensive or takes too long
  - need to locate problems within SUT not DOC
  - want to test usage of DOC by SUT is correct
Why the need for Test Doubles?

- **Simpler Tests and Design**
  - especially for external dependencies
  - promote interface-oriented design

- **Independent Testing of single Units**
  - isolation of unit under testing
  - or for not-yet-existing units

- **Speed of Tests**
  - no external communication (e.g., DB, network)

- **Check usage of third component**
  - is complex API used correctly

- **Test exceptional behaviour**
  - especially when such behaviour is hard to trigger
There exist different categories of Mock objects and different categorizers.

- **Stubs**
  - substitutes for “expensive” or non-deterministic classes with fixed, hard-coded return values

- **Fakes**
  - substitutes for not yet implemented classes

- **Mocks**
  - substitutes with additional functionality to record function calls, and the potential to deliver different values for different calls
Interface-oriented Test
Double introduction

- classic inheritance based mocking
  - extract interface for DOC -> IDOC
  - make SUT use IDOC
  - create MOCK implementing IDOC and use it in UT
Test Double Patterns
[Beck-TDD]

- **Mock Object**
  - Decouple a class under test from its environment

- **Self Shunt**
  - Use the test case class itself as a Mock Object

- **Log String**
  - Test temporal dependencies of calls by concatenating call info in a string, e.g., using Self Shunt

- **Crash Test Dummy**
  - How do you test exceptions that are hard to force, but might occur during production?
  - Use a dummy/Mock Object that throws an exception instead of the real object.
Example for a Crash Test Dummy

- like a Mock Object, but an inner class for it can be an elegant solution

```java
public void testFileSystemError() {
    File f = new File("foo") {
        public boolean createNewFile() throws IOException {
            throw new IOException();
        }
    };
    try {
        saveAs(f);
        fail();
    } catch (IOException e){}
}
```

how to test Exceptions!
Mock Object xunitpatterns

- How do we implement Behavior Verification for indirect outputs of the SUT?
- How can we verify logic independently when it depends on indirect inputs from other software components?
Difference Test-Stub and Mock-Object

xunitpatterns.com
Why Test Doubles and Mock Objects? [PragUnit]

- The real object has **nondeterministic behavior** (it produces unpredictable results, like 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 confirm 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).
Benefits of Test Doubles

- **tests run faster and are simpler**
  - test really the component in isolation not its environment

- **software design is improved**
  - less tight coupling
  - programming against interfaces instead of concrete objects
  - Parameterize from Above
    - e.g., PrintStream parameter instead of System.out
    - output can be tested automatically

- **Refactoring can be necessary for getting these benefits**
import java.util.Calendar;
public class Checker {
    public Checker(Environmental anEnv) {
        env = anEnv;
    }

    /**
     * After 5 o'clock, remind people to go home
     * by playing a whistle
     */
    public void reminder() {
        Calendar cal = Calendar.getInstance();
        cal.setTimeInMillis(env.getTime());
        int hour = cal.get(Calendar.HOUR_OF_DAY);
        if (hour >= 17) { // 5:00PM
            env.playWavFile("quit_whistle.wav");
        }
    }

    private Environmental env;
}

public interface Environmental {
    public long getTime();
    // Other methods omitted...
    public void playWavFile(String name);
}

public class SystemEnvironment implements Environmental {
    public long getTime() {
        return System.currentTimeMillis();
    }
    // other methods ...
    public void playWavFile(String name) {
        // Left as an exercise
    }

    public class MockSystemEnvironment implements Environmental {
        public long getTime() {
            return current_time;
        }
        public void setTime(long aTime) {
            current_time = aTime;
        }
        private long current_time;
        public void playWavFile(String filename) {
            playedWav = true;
        }
        public boolean wavWasPlayed() {
            return playedWav;
        }
        public void resetWav() {
            playedWav = false;
        }
        private boolean playedWav = false;
        // ...
    }
import junit.framework.*;
import java.util.Calendar;

public class TestChecker extends TestCase {
    public void testQuittingTime() {
        MockSystemEnvironment env = new MockSystemEnvironment();

        // Set up a target test time
        Calendar cal = Calendar.getInstance();
        cal.set(Calendar.YEAR, 2004);
        cal.set(Calendar.MONTH, 10);
        cal.set(Calendar.DAY_OF_MONTH, 1);
        cal.set(Calendar.HOUR_OF_DAY, 16);
        cal.set(Calendar.MINUTE, 55);
        long t1 = cal.getTimeInMillis();
        env.setTime(t1);

        Checker checker = new Checker(env);

        // Run the checker
        checker.reminder();

        // Nothing should have been played yet
        assertFalse(env.wavWasPlayed());

        // Advance the time by 5 minutes
        t1 += (5 * 60 * 1000);
        env.setTime(t1);

        // Now run the checker
        checker.reminder();

        // Should have played now
        assertTrue(env.wavWasPlayed());

        // Reset the flag so we can try again
        env.resetWav();

        // Advance the time by 2 hours and check
        t1 += 2 * 60 * 60 * 1000;
        env.setTime(t1);

        checker.reminder();
        assertTrue(env.wavWasPlayed());
    }
}
Summary Test Doubles and Mock Objects

- Test Doubles/Mock Objects are important for isolating unit tests
  - or speeding them up
- They can lead to better, less-coupled design
  - separation of concerns
    - danger with auto-generated mock objects
- Overdoing mocking can be dangerous
  - go for simplicity!
  - test against interfaces, do not over-specify a specific implementation
Questions?
Outlook

- Even though unit tests relieve the burden of interactive debugging you will learn a bit more about that later.
- Refactoring your code is relying on GUTs and is even more important to make your design better and simpler.
- Look forward to conscious debugging and bug tracking in the last week of the semester.
References

- [Gerard Meszaros] - xUnit Test Patterns
  - http://xunitpatterns.com
  - very good overview of the problems of and with test automation and their solutions

- [Beck-TDD]
  - Kent Beck: Test-Driven Design

- [PragUnit]
  - Andy Hunt, Dave Thomas: Pragmatic Unit Testing

- [Kevlin Henney]
  - JUTLAND:
    - Java Unit Testing: Light, Adaptable 'n' Discreet

- [Dave Astels] - TDD
  - Test Driven Development: A Practical Guide

- [Dan North] - Behaviour Driven Development
  - http://dannorth.net/introducing-bdd/
  - http://behaviour-driven.org/