Developing, Testing, Debugging

Some slides adapted from FSE’98 Tutorial by Michal Young and Mauro Pezze'

Tools, Testing, Debugging

• Goal: write reliable, correct software
  – And do it without too much pain!
• The development process
  – Analyze requirements (what am I trying to do?)
  – Write code (how am I doing it?)
  – Test code (does my code work?)
  – Debug code (nope—what’s wrong?)
  – then iterate! (has what I’ve written met the requirements?)
Holistic Development: IDE’s

- Interactive Development Environments
  - Typically simplify writing of code (editor + compiler hook)
  - May help with debugging (syntax checking, debugger interface)
  - May help with testing (interface to test tool)
  - May help with search, transformation, etc.

- Many available
  - Eclipse JDT
  - Dr. Java
  - Visual Studio
  - ...

Dr. Java

- Editing
  - Syntax coloring
  - Auto-indent
  - Brace matching

- Testing
  - Integrates with Junit testing framework
  - Interaction panel allows interactive method invocations

- Debugging
  - Integrates with Java debugger
  - Interactions panel also useful
Testing

How do I know my program works?

• Identify properties of interest
• Formally prove they are upheld, or
• Test them by running with various inputs

• But which properties? How do I do it? What if my program changes? When am I done? Ahhh!!!

• It gets worse …
You can’t always get what you want

- Correctness properties are undecidable
  - the halting problem can be embedded in almost every property of interest

Getting what you need ...

- We must make the problem of verification “easier” by permitting some kind of inaccuracy

Sampling the input space

Optimistic inaccuracy (testing)

Pessimistic inaccuracy (analysis, proofs)

Perfect verification

“Easier” properties

Folding the input space
Easier Properties - Example: Unmatched Semaphore Operations

```java
if ( .... ) {
    ...
    lock(S);
}...

if ( .... ) {
    ...
    unlock(S);
}
```

Static checking for match is necessarily inaccurate ... ... so Java prescribes a more restrictive, but statically checkable construct.

Testing

- Executing the program on a sample of input data
- *Dynamic technique*: programs must be executed
- Optimistic inaccuracy
  - the program runs on a (very small) *subset* of input data
  - the behavior of the program on *every* input is assumed to be consistent with the examined behaviors
**Goals of Testing**

- Find faults ("Debug" Testing):
  - a test is successful _iff_ the program fails\(^1\)
- Provide confidence (Acceptance Testing)
  - of reliability
  - of (probable) correctness
  - of detection (therefore absence) of particular faults

\(^1\) Goodeneough, Gerhart, "Toward a Theory of Test Data Selection", IEEE Transactions on Software Engineering, Jan. 85

---

**Testing Activities**

- Test case execution is only a part of the process
- Must also consider
  - Test case generation
  - Test result evaluation
- Planning is essential
  - To achieve early and continuous visibility
  - To choose appropriate techniques at each stage
  - To build a testable product
  - To coordinate complementary analysis and testing
The Test Case Generation Problem

• What tests will show that my program works?
  – Must consider “operational scenarios”
  – What is legitimate input?
  – What is the correct action or output?
• How can I make sure that all of the important behaviors of my program have been tested?

Granularity of Tests

• Whole program
  – Test case inputs to whole program, and outputs examined
• Piece-meal
  – Individual components of a program are tested
    • Methods
    • Classes/packages
    • Processes of a distributed system
Styles of Tests

- Functional (black box)
  - based on specifications ("external behavior")
- Structural (white box)
  - based on code
- Fault based
  - based on classes of faults

Black Box Testing

- Pick a subcomponent of the program
  - Internals of component not considered
- Give it inputs
- Compare against correct outputs
White Box Testing

- Pick a subcomponent of the program
- Give it inputs
  - Inputs determined based on component code
- Compare against correct outputs

```
inputs ➔ subcomponent being tested ➔ outputs ➔ Is it correct?
```

White vs. Black box

- Black box
  - depends on spec
  - scales up
    - different techniques at different granularity levels
  - it cannot reveal code coverage problems
    - same specification implemented with different modules
- White box
  - depends on control or data flow coverage
  - does not scale up
    - mostly applicable at unit and integration testing level
  - cannot reveal missing path errors
    - part of the specification that is not implemented
The Testing Environment

• Want to create a scaffold for executing tests
  – Code infrastructure that allows tests to be run easily and the results checked for correctness

• Many benefits
  – Can automate testing process
  – Useful for regression testing

• But, can take some time to implement

Testing Environment Components

• A *user* to generate input for tested component
• An *oracle* for verifying the results are correct

• These two may be combined into a single system
Unit Testing with **JUnit**

- Testing environment for writing black-box tests
  - Write special **TestCase** classes to test other classes
  - Several ways to use/set up test cases
- Can be downloaded from
  - [http://www.junit.org](http://www.junit.org)
- Simple version included in Dr. Java
  - Implements only a subset of Junit functionality

**JUnit Philosophy**

- Iterative, incremental process
  - Write small black-box test cases (as needed)
  - Test-as-you-go
    - i.e., after changes, when new method added, when bug identified
- **JUnit** test cases must be completely automated
  - No human judgment
  - Easy to run many of them at the same time
- Goal: lots of bang for the buck
  - Even simple tests can find many bugs quickly
JUnit Components

- Test cases (class `TestCase`)
  - Individual tests
  - Can reuse test case setup (optional)
- Test suites (class `TestSuite`)
  - Test case container
- Test runner (various classes)
  - Executes test suites and presents results

Each test has three 3 parts

- Code that creates test objects
  - Create a subclass of `junit.framework.TestCase`
- Code that executes the test
  - Override the method `runTest()` (which executes the test)
- Code that verifies the result
  - e.g., use `junit.framework.assertTrue()` to check results
    (throws exception is test fails)
public class LogRecordTest extends TestCase {
    protected String event1 = "test";

    public void testEquals1() {
        LogRecord tmp1 = new LogRecord(event1+"1");
        LogRecord tmp2 = new LogRecord(event1+"2");
        assertTrue(tmp1.equals(tmp1)); // should pass
        assertTrue(tmp1.equals(tmp2)); // should fail
    }

    public void runTest() { testEquals1(); }

    public static void main(String[] args) {
        TestResult result = (new LogRecordTest("testLogRecord")).run();
        if (!result.wasSuccessful()) { System.out.println("Doh!"); }
    }
}

---

### Setup/Teardown

- Creating objects for each test too simple
  - Setup overhead grows as number of tests grows
  - Instead, group setup (and teardown) code in one place and reuse
- `junit.framework.TestCase.run()` executes test case:
  - `public void run() { setUp(); runTest(); tearDown(); }`
  - Put setup code in `setUp()` method
  - Put cleanup code in `tearDown()` method
public class LogRecordTest extends TestCase {
    protected String event1;
    LogRecord tmp1, tmp2;

    public void setUp() {
        event1 = "test";
        tmp1 = new LogRecord(event1+"1");
        tmp2 = new LogRecord(event1+"2");
    }

    public void testEquals1() {
        assertTrue(tmp1.equals(tmp1)); // should pass
        assertTrue(tmp1.equals(tmp2)); // should fail
    }
    // other cases here will reuse tmp1, tmp2, event1, etc. …
}