Tools

Testable programs

Large Scale Programming, 1DL410, autumn 2009
Cons T Åhs
Tools overview

- Having and knowing a set of tools will make you more productive, so spend time learning your tools.
- Go beyond the basics from the start. Take time to learn the tools and the features of it, before you need a specific feature.
- There is no excuse for not knowing your tools, be it an IDE, a language or methods.
- A good craftsman has an extensive toolbox and knows how to use the tools.
- One tool (language, method, principle, pattern, ..) is never enough.
- **IDE**
  - Eclipse - widely used, available on the department unix systems.
  - IntelliJ - widely used, not available at the department.
  - **Note:** BlueJ is not good enough for our needs.
- Most IDEs have integration with other tools, such as version control systems (svn, cvs, ..), build and automation (ant), test (JUnit) etc.
- Don’t rely *only* on the IDE - you need to be able to do all standard tasks, such as building, running tests etc, outside of the IDE as well.
  - [See the section on Evil Wizards in *The Pragmatic Programmer*]
JavaDoc

- JavaDoc is a standard method of commenting source code (interfaces, classes, methods, instances variables).
- The source code can then be run through a tool which extracts the comments and generates, initially, HTML.
- Comments of the format /** ... */ are parsed by JavaDoc.
- The comment applies to the next element such as a class or method.
- Inside the comment you can use special tags, which will be formatted specially.
- Eclipse can help generate templates for JavaDoc comments (see Source>Generate Element Comment) with relevant tags.
- Eclipse can also show generated JavaDoc interactively, even for your own classes, so write it from the start. It helps! (in more ways than one)
- More info on JavaDoc can be found on the web or with man javadoc or inside Eclipse.
/**
 * Read a stream in PPM format and return an internal image.
 *
 * @param stream stream to read PPM from.
 * @param imageFactory a supplied factory that can create an suitable (empty) image, which the PPM is written to.
 * @return internal image read from the stream.
 *
 * TODO Rewrite to a finite state machine using hyperbolic indexing.
 * TODO Don’t believe everything you read.
 */

public void readStream(InputStream stream, IImageFactory imageFactory) {
    ...
}
Testing in Practice

- How often should tests be done?
  - Ideally after *every* change of source code.
  - Regression testing ensures that new changes haven’t broken anything that passed the tests before. We can use this to have a *known quality* of the software we’re producing.
- Is it always practical to run tests after every code change?
- What is the value of manual tests?
  - Essentially zero after a code change.
  - Manual tests are no tests.
- Testing should thus be automated as far as possible.
- Code that is written with testing in mind will not only be testable and tested; by some kind of magic it will actually be easier to reuse as well.
Testing in Practice

- No amount of testing can guarantee that a program is completely correct, i.e., that all results are according to the specification and that it produces everything the specification requires. [There is a formal definition of this]
  - The set of test cases is finite, so they can therefore not capture and fully describe a general behaviour.
  - A descriptive view is to see the tests as fences, trying to enclose the erratic behaviour of a program.

- What's does a bug reveal?
  - A hole in the fence!

- What's the first thing you should do when you discover a bug?
  - Write a test case to cover it!
  - Apparently it was missing.

- Testing is important and you should spend time, not on the actual testing, but on formulating and writing test cases.
- Given a good requirement specification you should write your tests first!
- This gives a way to test your interface before you spend time on implementation.
  - Test driven development.
Testing in Practice

- The lowest level of testing is unit testing, i.e., testing individual parts.
- A rather simplistic view is that you are testing a function in a box, i.e.,
  
  \[ y = f(x) \]

  Test cases are thus a set of pairs \(<x, e>\) that can be tested against your implementation by comparing \(f(x)\) against \(e\).

- For components that are leaves in the (ideal) tree structure of your system, this is fine and easily testable.

- What happens when you want to test a component that is not a leaf, i.e., something that depends on other parts?

  \[ y = g(u), \text{ where } u = f(x) \]

  If you compose your test cases as a set \(<x, e>\) and compare \(g(x)\) with \(e\), what is that you are actually testing?

  - **Both** the implementation of \(f\) and \(g\), i.e., if you find a fault, you really don’t know if the fault is in \(f\) or \(g\) or both.

  - This is actually *integration testing*, which is also needed, but we want a way of testing **only** \(g\).

  - Test cases should be \(<u, e>\), i.e, not include the implementation of \(f\).
Testing in Practice

- Two ways of implementation
  - $h(x) = g(u)$, where $u = f(x)$
  - $h(f, x) = g(f(x))$
- What’s the difference?
  - In the first case, the dependency to $f$ is fixed inside the implementation of $h$. We can’t change it or test $g$ in isolation.
  - In the second case, we pass in $f$ as a dependency. It can be changed and it is easy to test $g$ in isolation (by using test cases where we already know the values without depending on the implementation of $f$).
- Note how we pass in behaviour as an argument rather than having it fixed.
- In functional programming this is called higher order programming.
- This can be done in object oriented programming as well and will serve us well for both testing and creating components that are easy to reuse.
- Lifting behaviour breaks long chains of dependence, which make components difficult to reuse.
- We’ll get smaller building blocks that are easy to reuse.
Testing in Practice

- How can we do this with an object oriented programming language?
- Interfaces are the key!
- An interface specifies public methods, i.e., abstract behaviour.
- Actual behaviour is delivered by implementations of the interface.
- For testing purposes, we use implementations that implement a known behaviour that is easy to test against.
- Say we want to implement formatting of time stamped log messages.
  - log messages are written (presumably) to a file.
  - we want a method to log a single message.
Bad Implementation

- Always writes to file - a test must look at the file.
- Time stamps are created at each call - the “same” test run at different times will give different results in the file.
- We are locked in to logging to a file. If we want to log somewhere else, we have to rewrite the code, thus exposing it to risk.
- Difficult to test.
- Difficult to reuse.
- Difficult to modify.

```java
public class BadLogger {
    private PrintStream ostream;

    public BadLogger(String filename) throws FileNotFoundException {
        FileOutputStream out = new FileOutputStream(filename);
        ostream = new PrintStream(out);
    }

    public void log(String message) {
        Date now = new Date();
        ostream.print(now.toString() + ":" + message);
    }
}
```
Good Implementation

- Not locked to logging to a file - we create a writer outside (that implements the interface) and pass to the constructor.
- We can pass the date and thereby write tests that are comparable even when run at different times.
- Tests are written by providing a writer that just stores the messages.
Good Implementation

- Testing that the logger does what it should is just then a matter of comparing strings without having to look at specific files.
- We want to test the relevant component here, i.e., the creation of log messages, not all the mechanics of handling files.
- Easy to test.
- Easy to reuse.
- Easy to extend, without modifying any source code.
  - We just provide other implementations of `IStringWriter`.
- Note that our logger provides or owns the interface `IStringWriter` - it’s easy to wrap something that can do string output so it implements this interface.
  - The specification of this interface is part of the reusable package we provide.
  - Another user wanting to reuse our logger will thus need to import the logger and the interface.
JUnit

- **JUnit** is a framework for writing tests.
- You extend the class `junit.framework.TestCase`.
- Tests are written as public methods starting with `test`, e.g., `testEmptySort`. These are found by reflection by the framework when running the tests.
- Different forms of assertions are used to express what should hold in each test case, i.e., to describe the expected result or outcome.
- There are special methods such as `setUp()`, that are run before each test so you can provide a clean set up for each test without having to repeat code.
- Tests are run and a report is written.
- Eclipse has direct support for using JUnit.
- Spend some time to learn more about JUnit - you will need it.
Testing it.

```java
private class MockLogger implements IStringWriter {
    List<String> messages = new ArrayList<String>();

    public void writeString(String message) {
        messages.add(message);
    }

    public int length() {
        return messages.size();
    }

    public String getMsg(int n) {
        return messages.get(n);
    }
}
```

- Standard practice to call something a mock when it pretends to be something.
- This mock implements the `IStringWriter` interface so we can pass an instance to our logger.
- The implementation just stores the messages we want to log.
- Additionally it provides methods for obtaining the number of messages we have logged and retrieving a specific message by index.
The `setUp()` method is run before every test.

This ensures that we can run each test in isolation from each other and that they have the same initial environment in a simple way.

Additionally, there are methods that are run before and after a test suite.
Testing it.

public void testOneMessage() {
    final String message = "even adders multiply on a log table";
    Date now = new Date();
    String expected = now.toString() + ":" + message;

    logger.log(now, message);

    assertEquals(1, destination.length());
    assertEquals(expected, destination.getMessage(0));
}

public void testOrder() {
    final String mesg1 = "first message";
    final String mesg2 = "second message";
    Date now = new Date();
    String exp1 = now.toString() + ":" + mesg1;
    String exp2 = now.toString() + ":" + mesg2;

    logger.log(now, mesg1);
    logger.log(now, mesg2);

    assertEquals(2, destination.length());
    assertEquals(exp1, destination.getMessage(0));
    assertEquals(exp2, destination.getMessage(1));
}

- Test cases are run in an unspecified order.
- Each test should test something specific.
- Don’t test many unrelated things at once - when it fails you don’t know why.

tisdag 15 september 2009
Best practices

- Given requirements, do the following
  - What test cases do I want for testing?
  - How can I construct an interface that allows me to test it simply?
  - Write test cases and see the interface growing from a concrete need that forces it to be general.
  - Since the interface supports testability from the start, it is probably rather complete.
  - Implement your class or method to satisfy the tests.
  - Observe that the component is easy to reuse since it has a general interface.
- Use interfaces - they are mighty and powerful!
- Look at the principles called DIP (Dependent Inversion Principle) and OCP (Open Closed Principle) that can be found in *Agile Software Development* by Robert C. Martin or on the net.
Misc

- Graphical components are more difficult to test since the “result” is graphics behaviour or an image.
  - Not impossible, though, but a corollary is that you should make your GUI components really stupid.
  - The complexity should thus be in the controller, which you can test using the outlined methods.
- Static analysis with **pmd, checkstyle** - more later.