Software Testing, Quality Assurance and Maintenance Winter 2010

Lecture 28 — March 15, 2010

Patrick Lam *version 1*

Test Plans and Documentation

Today we'll talk about test plans and discuss coming up with test cases for existing large legacy systems.

It is easy to go overboard generating documentation and thinking that you are doing work—filling out paperwork is not progress. My main resource for today's lecture is the book by Cem Kaner, Jack Falk, and Hung Quoc Nguyen, *Testing Computer Software* (2nd ed); I also used notes by Kaner and Bach on "Requirements Analysis for Test Documentation", 2005.

> A test plan is a valuable tool to the extent that it helps you manage your testing project and find bugs. Beyond that, it is a diversion of resources.
>
> (Cem Kaner et al.)

Benefits of test plans and documentation. A test plan is analogous to a requirements document for test cases: to some extent,

1. it facilitates technical tasks of testing;
2. it improves communication about testing tasks and process; and
3. it provides structure for organizing, scheduling and managing testing.

Mechanics of Building a Test Plan

We can build the test plan in parallel with building tests. I'm going to also refer to the following document that I found on the Internet, which talks about developing test cases for jEdit.

`http://www.ibm.com/developerworks/java/library/j-legacytest.html`

These tests are emphatically not unit tests, even if written for JUnit.

Initial Test Plan

The initial test plan is an outline for future work.

- Enumerate fundamental test requirements (get the program working);
- Create test requirements from program documentation (if it exists);
- Create test documents (e.g. feature lists) and TRs from them; and
- Identify limits of the software system and corresponding TRs.

Keywords: "superficial but broad".

jEdit example. Let's examine the jEdit example. It begins with a test of the `main()` method (definitely not a unit test!), first calling `main()` and then ensuring that all focussed frames are instances of jEdit `Views`. The author then implements a hack to prevent calling `main()` more than once and tests that the menubar contains the correct menus (of dubious utility, if you ask me.)

```
org.gjt.sp.jedit.jEdit.main(new String[0]);
// ...
if (f.isFocused()) {
   assertTrue(f instanceof org.gjt.sp.jedit.View);
}
```

The jEdit example then continues with a test of the cut feature; it drives the UI through a simulation of user input, select-all, and cut, and checks that the clipboard contains the desired text:

```
public void testCut() {
    JEditTextArea ta = view.getTextArea();
    ta.setText("Testing 1 2 3");
    JMenu edit = menubar.getMenu(1);
    JMenuItem selectAll = edit.getItem(8);
    selectAll.doClick();
    JMenuItem cut = edit.getItem(3);
    cut.doClick();
    assertEquals("", ta.getText());
    assertEquals("Testing 1 2 3", getClipboardText());
}
```

Why is this test case hard to maintain?

Refining the Test Plan

Use your intuition to decide where additional test requirements belong; for instance:

- add various types of coverage criteria (e.g. one test per package/class);
- most likely errors (for instance, where errors already exist: bugs clump);
- most visible errors (showstoppers/brown paper bag errors);
- most often used program areas;

- the program area which distinguishes your program from its competitors;
- program areas which are hardest to fix (start there); and
- parts of the program which you understand best.

jEdit. The jEdit document continues by proposing different ways to refine the test plan, e.g. imposing test requirements on a per-package, per-class, and per-method basis. These test requirements may eventually be met by tests that are actually unit tests.

Types of Test Plans

We can talk about different levels of test plans, which contain varying levels of technical content.

- Mission Plan
- Strategic Plan
- Tactical Plan

Test Planning Documents

- List of test cases;
- High-level designs of test cases and suites;
- Descriptions of platforms to test on, plus variations of platforms;
- Descriptions of interactions of tested software with other applications;
- Testing Project Plan: classes of tasks, plus resource allocation;
- etc.

Overdocumenting testing has a number of pitfalls.

Components of Test Plans

- Lists
- Outlines
- Tables
- Matrices
- Models

Two Matrix Examples

- System configuration variables.
- Configurations versus test cases.

When to Start Testing

Leaving aside test-driven development and developer-based testing, we can talk about when the QA team ought to start testing a release. It wastes everyone's time for QA to test half-baked "releases".

- Enforce rules for committing changes to repository ("if it doesn't build, don't commit it!" or donut rule; don't forget to `svn add`);
- Testers might pre-test a release before starting the full test process.

Another example: Debian "testing" repository.

Test Plan Composition

There is an IEEE standard for test plans, 829-1998.