xp123.com

The Test/Code Cycle in XP, Part 2: GUI

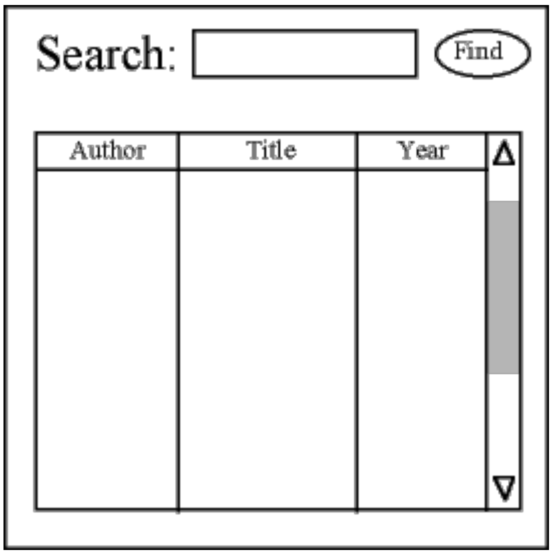
Catch Themes

16-20 Minuten

People who unit-test, even many who unit-test in Extreme Programming, don't necessarily test the user interface. You *can* use JUnit to assist in this testing, however. This paper will work through a small but plausible example, giving the flavor of testing and programming using JUnit. This paper is part 2, but can be read on its own; part 1 developed the model.

Example

Suppose we're developing a simple search engine. We'd like the user interface to look something like this:



We'll develop it in the XP style, working back and forth between testing and coding. The code fragments will reflect this: tests will be on the left side of the page, and application code on the right.

Model First

When you're creating a GUI (graphical user interface), you should develop and test the model first. We'll assume this has been done, and that it has the following interface:

```
public class SearcherFactory {
    public static Searcher
get(String s) throws IOException
{...}
}

public interface Searcher {
    public Result find(Query q);
}

public class Query {
    public Query(String s) {...}
    public String getValue()
{...}
}

public interface Result {
    public int getCount();
    public Document getItem(int
i);
}
```

```
public interface Document {
    public String getAuthor();
    public String getTitle();
    public String getYear();
}
```

In testing and development of the GUI, I don't mind depending on the *interfaces* of the model; I'm less happy when I have to depend on its concrete classes.

The GUI Connection

What we'd like to happen:

- a searcher is associated with the GUI
- a query is entered
- the button is clicked
- the table fills up with the result

We want to make this happen and unit-test the result.

Testing Key Widgets

We proposed a screen design earlier. The first thing we can test is that key widgets are present: a label, a query field, a button, and a table. There may be other components on the panel (e.g., sub-panels used for organization), but we don't care about them.

So, we'll create `testWidgetsPresent()`. To make this work, we need a panel for the overall screen

("SearchPanel"), the label ("searchLabel"), a textfield for entering the query ("queryField"), a button ("findButton"), and a table for the results ("resultTable"). We'll let these widgets be package-access, so our test can see them.

```
public void testWidgetsPresent() {
    SearchPanel panel = new
SearchPanel();

assertNotNull(panel.searchLabel);

assertNotNull(panel.queryField);

assertNotNull(panel.findButton);

assertNotNull(panel.resultTable);
}
```

The test fails to compile. (Of course – we haven't created SearchPanel yet.) So, create class SearchPanel with its widget fields, so we can compile. Don't initialize the widgets yet – run the test and verify that it fails. (It's good practice to see the test fail once; this helps assure you that it captures failures, and lets you ensure that the testing is driving the coding.) Code enough assignments to make the test pass.

Things to notice:

- The test helped design the panel's (software) interface.
- The test is robust against even dramatic re-arrangements of the widgets.
- We took very small steps, bouncing between test, code,

and design.

- Our panel might not (and in fact, does not) actually display anything – we haven't tested that.
- The panel still doesn't *do* anything (e.g., if the button were clicked).

We can make another test, to verify that the widgets are set up correctly:

```
public void
testInitialContents() {
    SearchPanel sp = new
SearchPanel();
    assertEquals("Search:",
sp.searchLabel.getText());
    assertEquals("",
sp.queryField.getText());
    assertEquals("Find",
sp.findButton.getText());
    assert("Table starts empty",
sp.resultTable.getRowCount() ==
0);
}
```

Run this test, and we're ok.

At this point, our SearchPanel code looks like this:

```
public class SearchPanel extends
JPanel {
    JLabel searchLabel = new
JLabel("Search:");
    JTextField queryField = new
JTextField();
    JButton findButton = new
JButton("Find");
    JTable resultTable = new
JTable();

    public SearchPanel() {}
}
```

We could go in either of two directions: push on developing the user interface, or its interconnection with searching. The urge to "see" the interface is strong, but we'll resist it in favor of interconnection.

Testing Interconnection

Somehow, we must associate a Searcher with our GUI, and verify that we display its results.

We'll give our panel two methods, `getSearcher()` and `setSearcher()`, that will associate a Searcher with the panel. This decision lets us write another test:

```
public void testSearcherSetup()
{
    Searcher s = new Searcher()
{
        public Result
search(Query q) {return null;}
    };

    SearchPanel panel = new
SearchPanel();
```

```
    assert ("Searcher not set",
panel.getSearcher() != s);
    panel.setSearcher(s);
    assert("Searcher now set",
panel.getSearcher() == s);
}
```

The compile fails, so bounce over to SearchPanel, add the methods, run the tests again, they fail; implement the set/get methods, and the test passes.

The panel still can't do much, but now we can associate a Searcher with it.

Testing with a Fake Searcher

A search returns a set of results. When something returns a list of values, I'm always interested to see how it will behave when it returns 0, 1, or an arbitrary number.

Because this is a unit test, I don't want to depend on the real Searcher implementations: I'd rather create my own for testing purposes. This lets me control behavior in a fine-grained way. Here, I'll create a new Searcher called TestSearcher. We'll have the query string be an integer, which will tell how many items to return. We'll name the items "a0" (for first author), "t1" (second title), etc.

But first… a test. (Notice this is a test of our testing class, not of our GUI.)

```
public void testTestSearcher() {
    assertEquals(new
Query("1").getValue(), "1");

```

```
    Document d = new
TestDocument(1);
    assertEquals("y1",
d.getYear());

    Result tr = new
TestResult(2);
    assert(tr.getCount() == 2);
    assertEquals("a0",
tr.getItem(0).getAuthor());

    TestSearcher ts = new
TestSearcher();
    tr =
ts.find(ts.makeQuery("2"));
    assert("Result has 2 items",
tr.getCount() == 2);
    assertEquals("y1",
tr.getItem(1).getYear());
}
```

Go through the usual compile/fail cycle, and create the test classes, starting with TestDocument:

```
public class TestDocument
implements Document {
    int count;
    public TestDocument(int n)
{count = n;}
    public String getAuthor()
{return "a" + count;}
```

```
    public String getTitle()
{return "t" + count;}
    public String getYear()
{return "y" + count;}
}
```

The TestResult class has a constructor that takes an integer telling how many rows should be present:

```
public class TestResult
implements Result {
    int count;
    public TestResult(int n)
{count = n;}
    public int getCount()
{return count;}
    public Document getItem(int
i) {return new TestDocument(i);}
}
```

TestSearcher uses the number value of the query string to create the result:

```
public class TestSearcher
implements Searcher {
    public Result find(Query q) {
        int count = 0;
        try {count =
Integer.parseInt(q.getValue());}
        catch (Exception ignored)
{}

        return new
TestResult(count);
    }
}
```

Run the test again, and it passes.

0, 1, Many

We'll build tests for the 0, 1, and many cases:

```
public void test0() {
    SearchPanel sp = new
SearchPanel();
    sp.setSearcher (new
TestSearcher());
    sp.queryField.setText("0");
    sp.findButton.doClick();
    assert("Empty result",
sp.resultTable.getRowCount() ==
0);
}
```

At last, we're using the GUI: setting text fields, clicking buttons, etc.

We run the test – and it passes! This means we already have a working solution – if our searcher always returns 0 items.

We move on:

```
public void test1() {
    SearchPanel sp = new SearchPanel();
    sp.setSearcher (new TestSearcher());
    sp.queryField.setText("1");
    sp.findButton.doClick();

    assert("1-row result",
sp.resultTable.getRowCount() == 1);
    assertEquals(
        "a0",

sp.resultTable.getValueAt(0,0).toString());
}
```

Now we fail, because we don't have any event-handling code on the button.

When the button is clicked, we want to form the string in the text field into a query, then let the searcher find us a result we can display in the table. However, we have a problem in matching types: the Searcher gives us a Result, but the table in our GUI needs a TableModel. We need an adapter to make the interfaces conform.

Record our Mental Stack

We have several things in progress at the same time, so it's a good time to review them – and write them down – so we don't lose track of anything.

- Write the button code
- Test and develop a TableModel adapter
- Get `test1()` to pass
- Write `testN()` and get it to pass
- Test the "look" of the GUI

Adapter Implementation

Let's write the button code as if a ResultTableAdapter class existed:

```
findButton.addActionListener(new
ActionListener() {
    public void actionPerformed(ActionEvent
e) {
        Query q = new
Query(queryField.getText());

        resultTable.setModel(
            new
ResultTableAdapter(getSearcher().find(q)));
    }
});
```

When this fails to compile, stub out a dummy implementation:

```
public class ResultTableAdapter
extends DefaultTableModel {
    public
ResultTableAdapter(Result r) {}
}
```

`Test0()` still passes, and `test1()` still fails.

The adapter is straightforward to write, but we begin by writing a test.

```
public void
testResultTableAdapter() {
    Result result = new
TestResult(2);
    ResultTableAdapter rta = new
ResultTableAdapter(result);
    assertEquals("Author",
rta.getColumnName(0));
    assertEquals("Title",
rta.getColumnName(1));
    assertEquals("Year",
rta.getColumnName(2));
    assert("3 columns",
rta.getColumnCount() == 3);

    assert("Row count=2",
rta.getRowCount() == 2);
    assertEquals("a0",
rta.getValueAt(0,0).toString());
    assertEquals("y1",
rta.getValueAt(1,2).toString());
}
```

The test fails because the dummy implementation doesn't do anything.

Bounce over and implement the ResultTableAdapter. Change it to be a subclass of AbstractTableModel (instead of DefaultTableModel), then implement the column names, column and row counts, and finally `getValueAt()`.

```
public class ResultTableAdapter
        extends
AbstractTableModel implements
```

```
TableModel {
    final static String
columnNames[] = {"Author",
"Title", "Year"};
    Result myResult;

    public
ResultTableAdapter(Result r)
{myResult = r;}

    public String
getColumnName(int i) {return
columnNames[i];}

    public int getColumnCount()
{return columnNames.length;}

    public int getRowCount()
{return
myResult.getItemCount();}

    public Object getValueAt(int
r, int c) {
        Document doc =
myResult.getItem(r);

        switch(c) {
        case 0: return
doc.getAuthor();
        case 1: return
doc.getTitle();
        case 2: return
doc.getYear();
        default: return "?";
        }
    }
}
```

This test (`testResultTableAdapter`) should pass, and so should `test1()`.

TestN and More

Write `testN()`, with say 5 items. It will also pass.

What else can give you problems? One possible problem occurs when we do a sequence of queries – can we get "leftovers"? For example, a query returning 5 items followed by a query returning 3 items should only have 3 items in the table. (If the table were improperly cleared, we might see the last two items of the previous query.)

We can test a sequence of queries:

```
public void
testQuerySequenceForLeftovers() {
    SearchPanel sp = new
SearchPanel();
    sp.setSearcher (new
TestSearcher());

    sp.queryField.setText("5");

    sp.findButton.doClick();

assert(sp.resultTable.getRowCount()
== 5);

    sp.queryField.setText("3");
    sp.findButton.doClick();

assert(sp.resultTable.getRowCount()
== 3);
}
```

This test passes.

Testing for Looks

We have a properly connected panel. We can check the widgets' relative locations:

- label left-of queryField
- queryField left-of findButton
- queryField above table

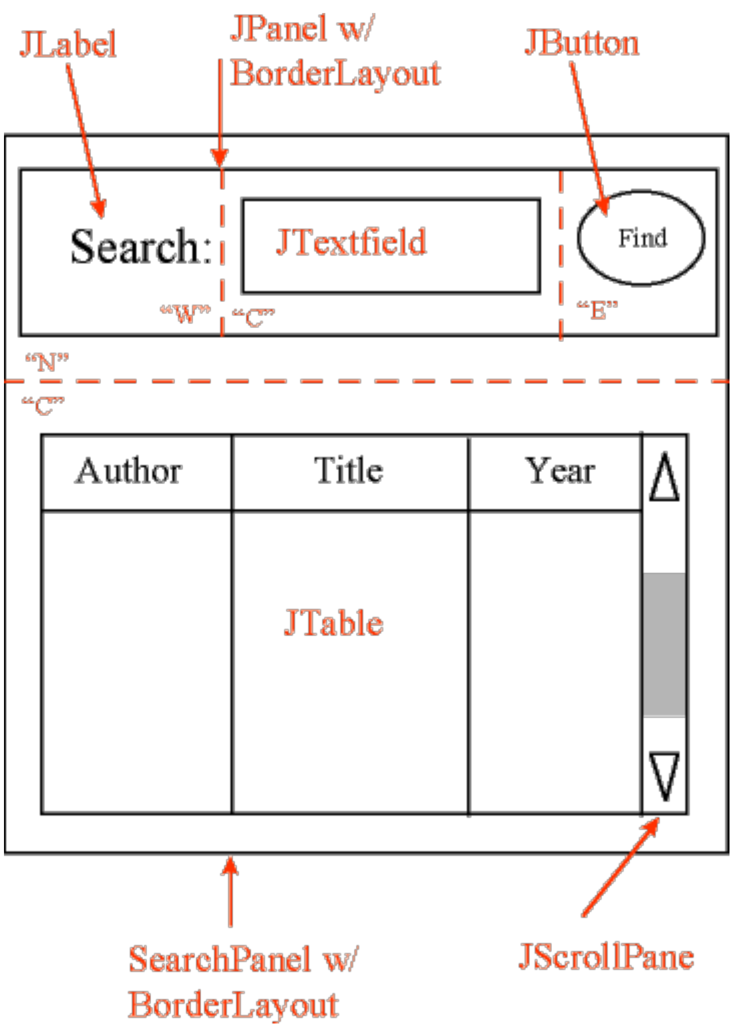
(Would we bother with these tests? Perhaps not, we might just put the panel on-screen and deal with its contents manually. There are times when such tests would definitely be appropriate: perhaps when we're working against a style guide, or when the window format is expected to be stable.)

To make this test run, we need to put our panel in a frame or window. (Components don't have their screen locations set until their containing window is created.)

```
public void testRelativePosition() {
    SearchPanel sp = new SearchPanel();

    JFrame display = new JFrame("test");
    display.getContentPane().add(sp);
    display.setSize(500,500);
    display.setVisible(true);

    //try {Thread.sleep(3000);} catch
(Exception ex) {}

    assert ("label left-of query",

sp.searchLabel.getLocationOnScreen().x
        <
sp.queryField.getLocationOnScreen().x);

    assert ("query left-of button",

sp.queryField.getLocationOnScreen().x
        <
sp.findButton.getLocationOnScreen().x);

    assert ("query above table",

sp.queryField.getLocationOnScreen().y
        <
sp.resultTable.getLocationOnScreen().y);
}
```

The test fails, as we haven't done anything to put widgets on the panel. (You can un-comment the sleep() if you want to see it on-screen.)

To implement panels, I usually do a screen design that shows the intermediate panels and layouts:



Now we can lay out the panel:

```
public SearchPanel() {
    super (new BorderLayout());

    findButton.addActionListener(new
ActionListener() {
        public void
actionPerformed(ActionEvent e) {
            Query q = new
Query(queryField.getText());
            resultTable.setModel(
                new
ResultTableAdapter(getSearcher().find(q)));
        }
    });

    JPanel topPanel = new JPanel(new
BorderLayout());
    topPanel.add(searchLabel,
BorderLayout.WEST);
    topPanel.add(queryField,
BorderLayout.CENTER);
    topPanel.add(findButton,
BorderLayout.EAST);

    this.add(topPanel, BorderLayout.NORTH);
    this.add(new JScrollPane(resultTable),
BorderLayout.CENTER);
}
```

Compile, test, and it works.

We've successfully implemented our panel!

Main

To complete the system, we'll create a `main()` routine:

```
public class Main {
    public static void
main(String[] args) {
        if (args.length == 0) {
            System.err.println(
                "Arg - file
w/tab-delimited author/title
/year");
            System.exit(1);
        }

        Searcher searcher = null;
        try {
            searcher =
SearcherFactory.get(args[0]);
        } catch (Exception ex) {
            System.err.println(
                "Unable to open
file " + args[0] + "; " + ex);
            System.exit(1);
        }

        SearchPanel sp = new
SearchPanel();
        sp.setSearcher(searcher);

        JFrame display = new
JFrame("Bibliographic System - "
+ args[0]);

display.getContentPane().add(sp);
        display.setSize(500,500);
        display.setVisible(true);
    }
```

```
}
```

Conclusions

We've completed development of our user interface. Not every aspect of a GUI can be unit-tested through the approach we've used, but we've identified a number of useful techniques:

- Even GUI development can maintain the cycle-in-the-small of test-code-design.
- GUI tests can be robust against changes in how the widgets are arranged on-screen.
- Fields and buttons can be simulated with `getText()`, `setText()`, `doClick()`, etc.
- Stub out the services provided by the model, to get fine-grained control over what the GUI test shows.
- We can test relative positioning using `getLocationOnScreen()`.

Unit tests can be tedious to write, but they save you time in the future (by catching bugs after changes). Less obviously, but just as important, is that they can save you time now: tests focus your design and implementation on simplicity, they support refactoring, and they validate features as you develop.

Resources and Related Articles

- xp0001.zip contains gui.jar (Java code for the GUI) and search.jar (Java code from part 1, the model).

- "The Test/Code Cycle in XP: Part 1, Model", William Wake. (Chapter 1 of *Extreme Programming Explored*.)
- *Extreme Programming Explained: Embrace Change*, Kent Beck.
- *Refactoring: Improving the Design of Existing Code*, Martin Fowler.
- JUnit home
- Test-First Challenge

Translations

- Japanese: Part 1, Part 2. Courtesy of Shinichi Omura.

[Written 1-3-2000; revised 2-1-2000; re-titled and revised 2-4-2000. Linked to xp0001.zip, 10-26-2000.]