Proceedings of

Workshop on Testing in XP
WTiXP 2002

May 27, 2002
Alghero, Sardinia, Italy

`http://www.cwi.nl/wtixp2002/`

Workshop in conjunction with the Third International Conference on
eXtreme Programming and Agile Processes in Software Engineering (XP2002)

Edited by Leon Moonen (Leon.Moonen@cwi.nl)
Printed by CWI, the national research institute for Mathematics and Computer Science in the Netherlands.

Contents

Workshop on Testing in XP

Background

"If there is a technique at the heart of extreme programming (XP), it is unit testing" [1]. As part of their programming activity, XP developers write and maintain (white box) unit tests continually. These tests are automated, written in the same programming language as the production code, considered an explicit part of the code, and put under revision control. The XP process encourages writing a test class for every class in the system. Methods in these test classes are used to verify complicated functionality and unusual circumstances. Moreover, they are used to document code by explicitly indicating what the expected results of a method should be for typical cases. Last but not least, tests are added upon receiving a bug report to check for the bug and to check the bug fix.

On the other hand, there are a lot of issues surrounding testing that are not that well understood:

- What part of the code do you actually test? How much testing is enough? How to determine "everything that could possibly break"?
- How can we recognize and reuse testing patterns (such as, for example, mock objects)?
- What happens to the tests when code is refactored? On the one hand, we cannot change them since we need them to validate correctness after the changes. On the other hand, refactoring can move functionality between classes, so we need to update our tests or they will fail.
- What about GUI testing, performance testing, and distribution testing?

And there is more than unit testing: *Acceptance tests* (also known as *functional tests* in XP terms) are used to prove that the application works as the customer wishes. They help the customers to gain confidence that the whole product is progressing in the right direction. Acceptance tests operate from the customer perspective, they don't test every possible path in the code (unit tests take care of that), but demonstrate that the business value is present. Furthermore, they allow the programmer to track the state of implementation in relation to the user written story cards.

This kind of testing should also be done automatically to allow short testing cycles while programming. While programming acceptance tests for calculation parts of a software system is easy, acceptance testing for more interactive applications or embedded systems is much harder and more complicated to realize. The community could benefit from a discussion on the different acceptance testing techniques and additional ideas aimed at solving the difficulties in this area. Interesting acceptance testing topics that need more discussion include:

- Programmed (automatic) acceptance tests for highly interactive applications
- Programmed (automatic) acceptance tests for graphical applications
- Programmed (automatic) acceptance tests for web applications
- Programmed (automatic) acceptance tests in the presence of embedded systems
- Acceptance tests for performance-critic applications

Objectives

The purpose of this workshop is to bring together practitioners, researchers, academics, and students to discuss the state-of-the-art of testing in extreme software development projects. The goal is to share experience, consolidate successful techniques, collect guidelines, and identify open issues for future work.

Topics of interest

Workshop topics include, but are not limited to:

- Bad smells in testing
- Testing patterns
- Refactoring test code
- Test first design
- Acceptance testing
- Managing your test suite
- Dealing with testing conflicts
- Experience reports

Organization

Leon Moonen (CWI, the Netherlands)
Martin Lippert (University of Hamburg & Apcon WPS, Germany)

Program Committee

Arie van Deursen (CWI, the Netherlands)
Steve Freeman (M3P, United Kingdom)
Tim Mackinnon (Connextra, United Kingdom)
Gerard Meszaros (ClearStream Consulting, Canada)
Joseph Pelrine (MetaProg, Switzerland)
Stefan Roock (University of Hamburg & Apcon WPS, Germany)
Shaun Smith (ClearStream Consulting, Canada)

References

[1] K. Beck. Embracing change with extreme programming. *IEEE Computer*, 32(10):70–77, October 1999.

One suite of automated tests: examining the unit/functional divide

Geoffrey Bache
Carmen Systems AB
Odinsgatan 9
411 03 Göteborg, Sweden
+46 (0) 31 720 8137
geoff@carmen.se

Emily Bache
(independent)
Flunsåsliden 25
418 71 Göteborg, Sweden
+46 (0) 31 779 35 14
emily_bache@goteborg.utfors.se

ABSTRACT
Extreme Programming (XP) as written [1] prescribes doing and automating both unit and functional testing. Our experiences lead us to believe that these two sorts of testing lie at two ends of a more or less continuous scale, and that it can be desirable to instead run a XP project with just one test suite, occupying the middle ground between unit and functional. We believe that this testing approach offers most of the advantages of a standard XP testing approach, in a simpler way. This report explains what we have done, and our theory as to why it works.

Keywords
XP, Automated testing, Functional testing, Unit testing, Test First Development

1. INTRODUCTION

When we introduced XP at Carmen Systems, the worst problem with our development process was not our testing procedures being out of control. We already had automated testing, though not along the lines outlined by Beck, Jeffries et al [1, 2]. Following the advice to "Solve your worst problem first", we began introducing other aspects of XP, expecting that at some point testing would become our "worst problem" and we would start needing separate unit and functional test suites. That never seemed to happen - we have been doing all the other XP practices in 2 projects for 18 months or so, and our style of automated testing has not only not become a problem, but in fact a great success that seems to fit very well with the rest of XP.

The automated tests we have are perhaps best explained as "pragmatic acceptance tests" - we run the system as closely as possible to the way the customer will run it, while being prepared to break it into subsystems in order to allow fast, easily automatable testing. The overall effect is that the tests are owned by the customer, while being just about fast enough to be run by the developer as part of the minute by minute code-build-test cycle.

2. THE CARMEN TEST SUITE

What we have created is an application independent automatic testing framework written in Bourne shell and Python. The framework allows you to create and store test cases in suites, runs them in parallel over a network, and reports results. For each test case the framework provides stored input data to the tested program via options or standard input redirects. As it runs, the tested program produces output as text or text-convertible files. When it has finished, the testing framework then compares (using UNIX "diff") this output to version-controlled "standard" results. Any difference at all[1] is treated as a failure. In addition, the framework measures the performance of the test, and if it strays outside pre-set limits, (for example if it takes too long to execute) this is also recorded as failure.

New tests are added by providing new input options and running the system once to record the standard behaviour against which future runs will be measured. This behaviour is carefully checked by the customer, so that s/he has confidence the test is correct. Once verified, the new test case (ie input and expected results) is checked into version control with the others.

Of course, not all differences in system behaviour are undesirable, and it sometimes happens that a test failure is registered even though the new system behaviour seems as good as or better than the old. If this happens, it is up to the developer who made the code change that caused the test to fail to confirm with the customer that the change is desirable, and then check in the new standard results of the test(s). They must also add a comment explaining why the new behaviour is an improvement on the old. In this way the behaviour of the system can evolve in a fully controlled way.

---

1 [1] except for run-dependent output such as times and process IDs, which the framework ignores.

We have been very successful using this technique at Carmen Systems to test the decision making middle layer of a larger application - that is the bit between the user interface and the data storage. Since we are not testing the system end to end, we are not really doing Acceptance Testing from the customer's point of view. Since we are not writing tests in the same language as the code, and are not writing tests for individual classes, we are not doing Unit Testing. However, we do get enough of the advantages of both kinds of testing to support XP.

3. STRENGTHS AND WEAKNESSES

The most important ways the testing practices support the rest of XP are by providing developer confidence to refactor and customer confidence in progress being made. The testing we do provides both of those:

- Most of the tests can be run in a matter of minutes, (the tests run in parallel across a network), so they can be run at nearly every build, and can provide fast enough feedback to enable merciless refactoring.
- Every test corresponds to real input and customer-verified output, so the list of passing tests is an accurate measure the customer can use to assess progress.

This way of testing has other advantages, too. Adding a new test is very straightforward, all it requires is finding suitable input data then having the customer confirm that the output is correct. There is no application- or feature-specific code to maintain and refactor, only the generic testing framework itself. Another useful feature is the ability to run tests in parallel, using 3rd party load balancing software to make maximal use of the computing resources available on the network. This means that the speed of the test suite is only limited by network resources and the time it takes the longest test to run.

One criticism that has been levelled at this style of testing is that without unit tests, Test First Development (TFD) as such is not really possible. Beck describes TFD as a design technique [3], and it has been reported as such by many practitioners of XP [4]. However, despite not doing TFD, we have not had difficulty creating a system composed of objects exhibiting high cohesion and loose coupling. We have also not had difficulty evolving the design via merciless refactoring as new user stories are implemented. In short, our experience suggests that TFD is not the only way to evolve a good design within an XP project.

4. FURTHER WORK NEEDED

The applications with which we have so far used this testing technique all operate in batch mode, and do not need to deal with the problem of simulating interactive input. However, we have been able, on a trial basis, to integrate the test suite with a third-party GUI playback testing tool (QCReplay[5]). The playback tool simulates a user session in a repeatable way, and in effect makes an interactive application into a batch application. We hope that future XP projects with a GUI-focus will be able to build on this trial work. We also believe that other kinds of applications can usually be made to run in batch mode with a bit of effort and ingenuity.`

5. CONCLUSION

In this practitioners report we have outlined our experiences with automated testing in the middle ground on the scale between unit and acceptance testing. Our main conclusions are that since the customer is far better qualified than the developers to specify tests for the system, they should specify the tests. On the other hand, the power of placing testing in a very tight feedback cycle within development is essential to enable refactoring and agile design, so the tests must run quickly. If we can have one suite of tests that is both customer owned and fast to run, we have a powerful tool to support a simpler process than XP as written - with one type of testing rather than two.

REFERENCES

1. Beck, "Extreme Programming Explained"
2. Jeffries et al, "Extreme Programming Installed"
3. Beck, "Aim, Fire"
http://www.computer.org/software/homepage/2001/05Design/
4. Community discussion, for example
http://www.c2.com/cgi/wiki?TestDrivenProgramming
5. http://www.centerline.com/productline/qcreplay/qcreplay.html

Is GUI Testing Difficult?

Andrew Swan

andrews@owl.co.uk

It is claimed that it is difficult to test GUIs, either at all, or using test first techniques. One argument is that the user interface changes too often and that simple changes in the GUI can cause a large number of tests to break. This is usually based on testing GUIs using input recorders and comparing screen grabs. Another problem can be testing code which has been auto generated by a GUI builder tool. Often there will not be the hooks exposed to allow testing of this code.

I would argue that GUI testing is no more difficult than testing any other code. It is even possible to use "strict" test first, i.e. always have a failing test before changing implementation code. This is based on my experience writing GUI applications in Java using Swing.

I have collected 4 techniques that I believe are very helpful in testing GUI code, both for unit testing and acceptance testing.

1. First test

How do you write a failing test to prove you need a GUI?

```
public void testMain() {
        assertEquals( 0, Frame.getFrames().length );
        main( null );
        assertEquals( 1, Frame.getFrames().length );
}
```

The simplest code to make the test pass is:

```
public static void main( String[] args ) {
        new Frame();
}
```

2. Separate model and view

Model-view-controller is often given as a pattern to simplify GUI testing. The model referred to is usually the domain model, but for effective GUI testing you need to remove as much code as possible from the GUI. Even the simplest of user interfaces has a requirement for logic to control selection, focus traversal, enabling of controls, etc. There should be a model to represent this logic so it can be tested non-visually.

Within the Swing framework there are classes to represent these elements, for example, Action and ListSelectionModel. GUI testing should concentrate on testing these models.

3. Name components

Some of the most frequently changing aspects of a user interface can be the layout, and text on controls. If you have tests which rely on components being in a certain location, or having some particular text on them, you are likely to spend a great deal of time updating tests.

A simple solution is to associate a logical name with the control. This also provides a good first test to prove that a control exists.

```
public void testConstructor() {
        JDialog d = new MyDialog();
        assertNotNull( "OK button exists", findChildNamed( d, "OK" ) );
}
```

The simplest code to make the test pass is:

```
public MyDialog() {
        JButton b = new JButton();
        b.setName( "OK" );
        add( b );
}
```

The method findChildNamed recursively searches the child components of a given container for a component with the given name.

Another major advantage of naming components is using it as the basis of acceptance test scripts.

For example:

```
public void testSaveDialog() {
        findItem( findMenu( frame.getJMenuBar(), "File" ), "Save" ).doClick();
        assertEquals( 1, frame.getOwnedWindows().length );
        Dialog save = frame.getOwnedWindows()[ 0 ];
        assertTrue( save.isVisible() );
        findButton( save, "OK" ).doClick();
        assertFalse( save.isVisible() );
        assertEquals( 0, frame.getOwnedWindows().length );
}
```

This would test that a dialog becomes visible when the Save item on the File menu is clicked, and becomes hidden when the OK button in the dialog is clicked. Using getOwnedWindows ensures that the dialog has the frame as its owner.

4. Modal dialogs

How do you test a modal dialog when it will block the test as soon as it's shown?

```
public void testDialog() {
        MyDialog d = new MyDialog();
        try {
                d.show(); // blocks!
                assertTrue( d.isVisible() );
                assertTrue( d.isModal() );
        } finally {
                d.dispose();
        }
}
```

The assertion will never be reached until dialog is hidden, and then it will fail!

To solve this problem we can execute the show on another thread, the thread executing the Swing event queue.

```
public void testDialog() {
        MyDialog d = new MyDialog();
        try {
                SwingUtilities.invokeLater( new Runnable() { public void run() {
                        d.show(); } } );
                SwingUtilities.invokeAndWait( new NoOpRunnable() );
                assertTrue( d.isVisible() );
                assertTrue( d.isModal() );
        } finally {
                d.dispose();
        }
}
```

Firstly, show is called, the invokeLater will return as soon as the show has been queued on the event thread. Next a no-op is queued, which will only execute once the show has been executed. This has the effect of blocking the testing thread until the show has been called. It is then safe to make any assertions about the shown state of the dialog.

In conclusion, GUI testing is not difficult. XP testing is focused on using a lightweight coding framework, this can easily be extended to GUI testing. In addition these techniques can also be used as the basis for automated acceptance tests.

Dierk König

Canoo Engineering AG

Kirschgartenstr. 7

CH 4051 Basel, Switzerland

Dierk.Koenig@canoo.com

WebTest Position Paper

XP2002 Testing Workshop Submission

Testing is an important part of any serious development effort. For web applications it is crucial.

Defects in your corporate website may be only annoying at one time but they can cost you real money at other times, they can lower your market value and may even put you out of business.

Canoo WebTest helps you to reduce the defect rate of your web application.

What our customers care about

• Quality	Quality improvements are hard to achieve if you cannot see the the effects of your measures. Canoo WebTest measures the externally observable quality of your application.
• Development Risk	Is the development team on track? What progress did it achieve? What does it mean, if they say that 80% is working? Is it really? Canoo WebTest reports the real progress in terms of running Use Cases.
• Operations Risk	Can we put our application into production safely? Will it work? Will it not do any harm when running? Canoo WebTest tells you whether it will work.
• Delivery	Did the development team really deliver everything they promised? Canoo WebTest tells you what was delivered and whether it works as expected.
• Costs	The costs for testing must not exceed its benefits. Canoo WebTest is free of charge, tests are easy and quick to write. They can be run countless times unsupervised and automatically. In fact it is cheaper and faster than testing manually.

What programmers care about

As programmers we want to be sure that our web application works as expected. We want to validate our work. We need some backing so that we can boldly say: "Yes, we have done it correctly. Yes, it works. Yes, we are finished with this. No, we have not broken any old functionality."

If we apply the full set of tests to the system every day then it is be easy to find the cause of any reported defect, because it must be something we checked in yesterday.

If testing finds a defect, we want to solve it quickly. Therefore, we need to reproduce the unexpected behavior. What were the steps that led to this error? What was the sequence? What were the intermediate results? How much easier would it be to track down the error if we only had this information!

No matter how hard we try, there will always be defects that slip through our testing. They get reported by our users. We want to make sure that their feedback does not get lost, that the defect really gets solved, that it never appears again in future releases. The best solution is to write an automated test that exposes the bug. It will fail as long as the bug is unsolved. It will stay forever in our suite of tests.

We have to read a lot of documentation every day. Bad experience made us suspicious about the correctness of any external documentation. We don't really like writing documentation ourselves because we know that it is only a matter of time until it is out of sync with the system and all our effort will be wasted. If the documentation is done via automated tests, it is assured to be up to date, making it a reliable source of information. We are much more motivated to invest our time for this.

The same holds true for requirements specifications. It would be really convenient if we could automatically prove that we comply with the requirements spec. Therefore the spec needs to be formal enough to allow automated compliance tests. It must still be easy to understand so that the customer, the requirements analyst and the development team can all easily understand the spec. The specification language needs to be flexible enough to express page contents, workflow and navigational structures.

You may claim that all the above would be really helpful but impossible to implement under the constraints of real projects. We have done it ourselves and we have helped others doing it. The effect is tremendous: to the quality of the system, to the satisfaction of the customer and to the motivation and self-esteem of the development crew.

Testing is not for free, but it pays off.

How Canoo WebTest works

Canoo WebTest lets you specify test steps like

- get the login page

- validate the page title to be `Login Page`
- fill `scott` in the username text field
- fill `tiger` in the password field
- hit the ok button
- validate the page title to be `Home Page`

The example steps above make up a sequence of steps that only make sense if executed in exactly this order and within one user session. We call this a *use case* or a *scenario*. Canoo WebTest offers the appropriate abstraction for this. Refer to the Syntax Reference and the API Doc for a complete list of step types.

Converting the textual description into a Canoo WebTest is easy, as you see below. Note how close it is to the textual description.

The example as a Canoo WebTest

```
<target name="login" >
  <testSpec name="normal" >
    &config;
    <steps>
      <invoke       stepid="get Login Page"
        url="login.jsp"   />
      <verifytitle   stepid="we should see the login title"
        text="Login Page" />
      <setinputfield stepid="set user name"
        name="username"
        value="scott"     />
      <setinputfield stepid="set password"
        name="password"
        value="tiger"     />
      <clickbutton   stepid="Click the submit button"
        label="let me in" />
      <verifytitle   stepid="Home Page follows if login ok"
        text="Home Page"  />
    </steps>
  </testSpec>
</target>
```

This is XML and you will get all the support from your preferred XML editor, including syntax highlighting and code completion based on the `WebTest.dtd`. Canoo WebTest leverages the advantages of XML even further. You may have noticed the line `&config;`. This is an XML entity that refers to the content of a file. The XML parser inlines the file at test execution time. It

is one of the possible ways to share common settings for all test steps. Here the settings for protocol, host, port and webapp name are shared.

If you are familiar with the ANT build automation tool you will have recognized that Canoo WebTest makes use of this. If ANT is totally new to you, we recommend having a look at the ANT description at The Jakarta Project. Canoo WebTest exploits ANT's ability to structure a "build" into modules that can either be called separately or as a whole. That way, you can run any WebTest in isolation. You can also group tests into a testsuite that again can be part of a bigger testsuite. In the end you have a tree of testsuites, where each node and subtree can be executed.

The execution of the several test steps is currently implemented by using the HttpUnit API, again an Open Source package. Test results are reported in either plain text or in XML format for later presentation via XSLT. Standard reporting XSLT stylesheets come with the Canoo WebTest distribution. They can easily be adapted to your corporate style and reporting requirements.

A sidebar: Do you think that the above example is so easy that you do not need an automatic test for this? Consider the following variations:

- Bookmark — What if I try to get the Home Page directly without login?
- Other pages — We have to test that no page is shown without proper login and that we get the requested page after proper login.
- Bad Login — Bad login should keep us on the login page.

This is quite a number of scenarios to be tested. Now imagine a manual tester checking all this. Very soon he will get bored and unobservant, not to mention that resetting his session for every single test requires a lot of work. Is he *really* checking again all the possible variations at every full test?

Pragmatic Considerations

Test automation is key to better quality. Manual checks are more flexible and less expensive to do *one time*. They are more expensive and less reliable when tests need to be done over and over again. We advise to do manual checks for everything that cannot break after it worked once. Everything else should be automated if the automation can be done without excessive costs. We feel that testing with Canoo WebTest reaches the break-even point for 90% of our tests after only a few test runs.

We want to use what we already know. We don't want to learn a new language for the test automation. We want to rely on standard formats.

Functional testing can be classified as being either data driven or record/replay. Canoo WebTest follows the data driven approach. Record/replay is appealing at first, because you can create a lot of tests in a short time. A proxy logs what pages you request and stores the results. It can then replay the requests and compare the results against the stored ones. You typically have to tweak this procedure to tell the program what parts of the page are expected to change. The actual date and time are the most obvious examples. Every small change to your webapp causes a lot of these

tests to fail. These failures must be manually processed to separate the "real" failures from the "false negatives". Doing this is almost as tedious and error prone as the manual testing and is therefore discouraged e.g. by the Automated Testing Specialists group.

" Record/Playback is the least cost-effective method of automating test cases. "

Zambelich

Any automated test should fit snugly into your build process. If you are already using ANT for your build automation, it is no effort to integrate Canoo WebTest. An Example of this is Canoo WebTest itself. It contains a selftest that is written with Canoo WebTest. Every new build of Canoo WebTest triggers that selftest. You can explore this behavior online, starting at the Build Info link of the Canoo WebTest distribution page. Note that this is very convenient for nightly builds and even for use with a continuous integration platform like CruiseControl.

If your build process is not ANT based, calling Canoo WebTest is still easy. It means starting a Java Application. This can easily be done with every build script language that we know.

"Regression tests" is the concept of testing that asserts that everything that worked yesterday still works today. To achieve this, our tests must not be dependent on random data. Also, the expected result must be clear in advance as opposed to the "guru checks output" approach, where a specialist validates changing results. Tests must give a thumps up indication when successful and a detailed error indication otherwise. Well, this is pretty much like compiler messages.

Functional tests do not replace unit tests. They work together hand-in-hand. Consider the following example: Your Webapp displays an html table that is filled with data from the database. The maximum number of rows should be 20 and if there is more data available, a link should be shown that points to the page that contains the next 20 entries. If there is no data, no table should be shown, but the message "sorry, no data". We would test this with a) no data b) one row c) 5 rows d) exactly 20 rows e) 21 rows f) 40 rows g) 41 rows. A naive way of testing this would be to manipulate the database (maybe by using an administration servlet that we can call via "invoke") prior to calling the page. But this is not only very slow but also a little dangerous. What if two tests run concurrently against the same test database? They will mutually destroy their test setup. What if the test run breaks? Is the state of the test database rolled back? The whole job is difficult to do for a functional test, but easy and quick for a unit test. A unit test can easily call the table rendering and assert the proper "paging" without even having a database! What is left for the functional test is to assert that the table rendering logic was called at all.

There is a lot more to say about unit testing. Refer to JUnit and the annotated references for further information.

Canoo WebTest is an Open Source Java project and totally based on Open Source packages. If you are not satisfied with any of the functionality, you can adapt it to your requirements. Having the sources, you even gain the ability to start the test in the debugger, revealing everything that

goes on.

Canoo WebTest is free of charge. The downside is, that there is no guaranteed support. However, you can ask Canoo for special support incidents, a support contract and on-site help for introducing automated testing in your project.

Canoo WebTest is not restricted to any special technology on the server side. It makes no difference if you use Servlets, JSP, ASP, CGI, PHP or whatever as long as it produces html. Client side JavaScript will not get executed, but you can check for the expected JavaScript code to be delivered.

Browser dependencies are the menace of web programming. One possibility is to check manually against all the "supported browsers". Our approach is to validate our html to comply with the specification. A full and pedantic validation is outside the scope of Canoo WebTest, but every validation step calls the JTidy parser (part of HttpUnit) and will warn you on improper html. That has proven to be very helpful. If your manual tests reveal that certain html constructions produce different behavior in your supported browers (like empty table cells in IE and Netscape), you can set up a test that checks against the usage of these constructs.

Advanced Topics

We found Canoo WebTests to be easy to understand, maintain and create even for non-developers. We had testers, assistants, novice programmers, business-process analysts and even managers and customers writing tests. This opens another opportunity: if the customer is able to understand or even write the tests, than they can serve as a requirements collection. Our preferred way of dealing with requirements is: "Whatever you write in a test, we will make it run. We promise nothing else but this."

If we get the tests written in advance, they serve as a requirements specification. While implementing, they give feedback how far we are. After Implementation, they document what we have done. That documentation is always up to date, as we can prove by the click of a button. The format of this documentation may be unfamiliar (as it is not MS-Word) but it has "the power of plain text" (cf. The Pragmatic Programmer). It can easily be transferred into other formats, e.g. by using XSLT.

It is good practice to care for the quality of your tests no less than you do for the quality of your production code. The first point here is to avoid duplication. Canoo WebTest combines the options of XML and ANT for helping you with this.

Canoo WebTest allows defining modules that can be reused in a number of tests. A common example is a sequence of validation steps that you apply to almost every page. These steps check against error indications like http errors, java stack traces, "sorry, we cannot...", etc. It may also contain a check for the copyright statement that is supposed to appear on every page. The samples that come with Canoo WebTest show how to do this.

Sometimes we have to test the same scenario for a number of different languages, each with different classes of users and each of these combinations with different user settings, etc. That can easily lead to so many test combinations that copy/paste would make the tests unmaintainable. Canoo WebTest uses the ANT mechanics to allow calling tests with overriding parameters. Again, the distribution contains a comprehensive example. Although all the test combinations get tested, the test description contains the scenario only once plus the information about the variation of calling parameters.

Canoo WebTest can be used to do automated tracking of your project. If your tests capture all the requirements, then every test run gives you feedback on how much you have achieved so far. The history of test reports reflects your team's productivity in terms of delivered functionality. The last report always shows the current state of your project in the most reliable metric we know: running and tested use cases.

Quotes and Success Stories

Canoo WebTest has been used successfully in a number of organizations ranging from small internet startup companies up to global players, for intranet and internet sites, for portals and B2B applications. Needless to say that we use it for our own Canoo Online Services as well.

more to come here...

Annotated References

ANT
http://jakarta.apache.org/ant
• The leading build automation tool.
• The platform independent replacement for "make".

Automated Testing Specialists
http://www.sqa-test.com/
• Points to a huge set of resources about automated testing.
• Answers a lot of questions about testing.
• Homepage of an independent consultants' community.

Canoo Online Services
http://www.canoo.net/
• The Canoo Online Services for german language exploration.
• Includes hundreds of pages with static, dynamic and mixed content.

Canoo WebTest
http://webtest.canoo.com/webtest
• The Canoo WebTest distribution.
• An Open Source tool to facilitate automatic functional testing of html-bound web applications.

CruiseControl
http://cruisecontrol.sourceforge.net/
• The Open Source Continuous Integration facilitator.
• The site also points to more information about continuous integration.
• CruiseControl uses ANT to trigger new builds on any repository change and reports the build result as email and on a website using JSP, XML and XSLT.

HttpUnit
http://httpunit.sourceforge.net/
• The Open Source Web Site Testing tool for programmers.
• Captures web site testing in JUnit TestCases.

JUnit
http://www.junit.org/
• Home of the unit test community.
• Points to articles, downloads and other on-line resources.
• Unit test tools for other languages than Java are also available, including Smalltalk, C++, Perl, Python, JavaScript and even VisualBasic.

The Jakarta Project
http://jakarta.apache.org/
• The leading Java open source software site.
• Includes the Apache web server, Tomcat, ANT, Log4J, Cactus, ORO, Struts and many more.

The Pragmatic Programmer
http://www.pragmaticprogrammer.com/
• Addison-Wesley Oct 1999 ISBN: 020161622X
• It covers topics ranging from personal responsibility and career development to architectural techniques for keeping your code flexible, easy to adapt and reuse.

XML
http://www.w3.org/xml
• Extensible Markup Language

XSLT
http://www.w3.org/TR/xslt
http://www.w3.org/TR/xsl/
• XSL Transformations
• Uses the Extensible Stylesheet Language (XSL) for transformations of XML trees into other tree structures like html, formatting objects (that can be serialized as PDF) or XML again.

Automated Acceptance and System Testing Frameworks

Peter Kelley
Project Architect
Sentillion, Inc.

Abstract

This paper describes a mechanism for automating acceptance (functional) and system level testing in a distributed computing system. In addition to automating tests, the frameworks provide an easy to use mechanism for developers to create functional and system level tests for features they are implementing.

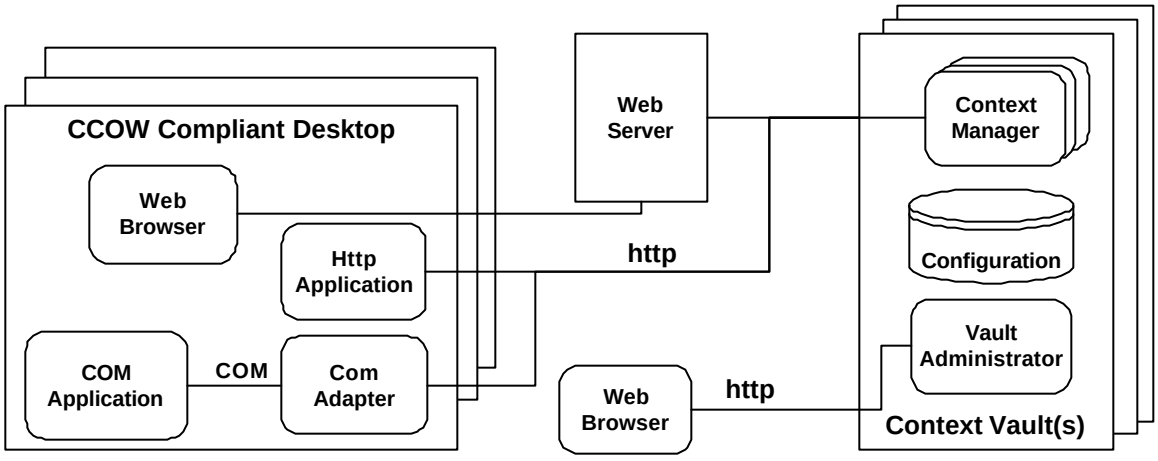
The system under test consists of a cluster of network appliances (Vaults). The Vaults incorporate Context Managers that coordinate the context of a number of disparate applications working on a single computing device. The Context Managers implement interfaces specified by the Health Level Seven (HL7) CCOW standard for context management. The Vaults provide load balancing, configuration replication and fail over capabilities. They are administered through a web interface. A fully CCOW compliant desktop version of the Vault with limited system capabilities is also provided.

Two frameworks have been created to automate the testing of this system. The Functional test framework uses JUnit to manipulate simulated applications that interact with each other and which exercise the Context Manager interfaces. The second framework, System test, uses HttpUnit to perform administrative functions on the Vaults. It also uses a distributed system of Functional test frameworks to verify the proper operation of the system capabilities.

CCOW

The CCOW standard establishes the basis for ensuring consistent access to patient information from heterogeneous sources by coordinating applications. CCOW-compliant applications coordinate by communicating with a Context Manager using a defined transaction for setting the context. They also implement a Context Participant interface to receive asynchronous notifications from the Context Manager.

Simplified System Diagram



Functional Test Framework

The Functional Test framework was designed to verify the external behavior of the Context Manager. A test interface was defined that abstracts the Context Manager interface to provide a means of exercising its full range of normal and abnormal behavior. Two test applications that implement the test interface were created in Java, one using the Http interface to the Context Manager directly, the other using the COM interface to the COM Adapter, which in turn communicates to the Context Manager. Each application maintains its own state and implements the Context Participant interface for communications originating from the Context Manager.

The test interface provides a method for each Context Manager interface method, but instead of providing fixed values for parameters, e.g. couponValue, booleans are supplied that instruct the applications to behave correctly or incorrectly, e.g. useInvalidCoupon. All Context Manager exceptions are caught and transposed into test application exceptions.

System Test Framework

The System Test framework was designed to verify the correct behavior of a complete context system. It consists of a system controller and a set of distributed slaves running on the various clients supported by the system (Win 9X, NT 2K). The controller communicates with the Vault Administrator using HttpUnit to verify the correct behavior of the system administrative functions (adding and removing Vaults, updating configurations, etc).

The controller then orchestrates a series of tests to verify system functionality in response to the administrative changes. It does this by manipulating the test environment, e.g. turning off a port on a programmable hub, and then coordinating the activity of the slaves. The slaves make use of the Functional Test framework to instantiate and operate various test applications. Communication between the master and slaves is accomplished using the same Http based protocol that the Context Manager uses.

Test Development

Functional and system tests are written in Java using JUnit. The functional tests instantiate applications to exercise the Context Manager. In addition, a GUI was created that interacts with the test applications to provide troubleshooting. System administrative tests are written in HttpUnit, which are controlled by the JUnit tests. Both frameworks provide a base class that extends junit.framework.TestCase to provide common functionality and template methods for all tests.

Test reporting

The system test framework provides test-reporting capabilities. A test summary class was created that encapsulates a collection of test result classes. These classes extend junit.framework.TestResult and provide additional detail including: build number, as well as pre and post-test system configuration. Test results are logged to files by build number and date of execution.

Test Execution

Functional tests may be run on any developer's workstation running against either a networked Vault or a Desktop Vault. Run time for the functional test suite is less than 10 minutes. It provides complete coverage of the Context Manager interface.

The functional tests are also incorporated into the nightly build process and are run against the Desktop Vault. Results are emailed to the build coordinator. The build process can be run manually at any time. It performs a clean build, refreshed from source control and runs the functional tests as a regression suite. This process takes less than an hour.

The automated system tests are launched manually. Currently we have automated 15% of these tests. The original manual system test procedure was in excess of 400 pages and required an engineer month to execute. The system tests that we currently have automated execute in 30 minutes.

Conclusion

Our efforts to date in automating functional and system testing have been very successful. Both the system and functional tests save substantial amounts of testing time, while empowering the organization in the practices of agile software development.

Furthermore, both frameworks have proven to be very usable by developers for creating new tests. During a recent code drop that incorporated changes to several subsystems (web server, database, JVM) the lead developer used the automated system tests to verify the changes, and he easily wrote new tests using the framework. Developers refactoring existing sections of code routinely use the functional test framework. And, it has proven easy to use for creating tests for new functionality. A typical test is less than 10 lines of code, while complex tests run to 20 lines of code.

Test First Design With UML / “A Picture is Worth a Thousand Programmers”

David Hussman
Edison Ed Inc.
4327 Garfield Ave South
Minneapolis, Minnesota 55409 USA
01-612-743-4923
david@edisoned.net

ABSTRACT
As a developer and a coach, I am continually surprised by the number of developers still trying to solve problems with an endless stream of words. Why is it still rare that developers use UML to communicate design? Is it not self evident that when a good tool exists, and is simple to use, we should use it? Also, why do so many developers and managers carry the misguided notion that XP and other agile processes are mutually exclusive with the use of UML diagrams? I have starting incorporating 1) a story or task for a story, 2) one or more hand written sequence diagrams, and 3) a collection of test classes and methods together into a process I informally call “sequence testing.” I do not present this process as new or of my design. Like XP, it is a conglomeration of best practices and tools into a simple process that can be used to help teams embrace test first design.

1 INTRODUCTION

Too often I find junior and senior developers gathered around a 21st century software Rosetta stone, speaking in tongues, and not communicating in the least. As soon as someone in the group draws a picture, even if it is a bad picture, people start talking, and the problem definition starts to materialize. Once all parties are speaking the same language, and a common discussion vehicle exists, solutions begin to surface.

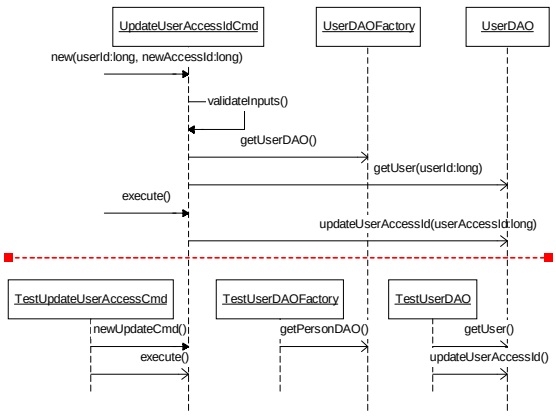
As a coach, I still hear developers refer to the notion that doing XP means not doing designs. I am not sure how this notion came to exist, but I do not agree, nor is this how I see XP and other agile methodologies evolving. I have struggled with pedagogies that help teams truly embrace test first design. Sure, there are those developers that simply take to test first because, in most cases, it is a formalization of the way in which they visualize problem solving and implementation issues. On the same team, there may be many programmers that either do not understand test first design, do not like the experience, believe test first to be unnecessary, or believe that test first is slowing progress.

Similar to the struggle to help developers truly embrace test first, I, and many others, have struggled to explain real OO design in a meaningful way, showing developers simple skills that can be put to use immediately, as well as skills that help create code that can be easily refactored. The collection of practices and tools that incorporate unit testing, simple sequence diagramming, and good OO design practices that I call sequence testing are the outcome of my struggles.

2 SEQUENCE TESTING

While working on XP projects, I find that quickly creating s simple sequence diagram (5- 10 minutes) helps provide a visual road map for a programming pair. Often times we are either displaying and processing data before or after it is persisted or fed to an external system or sub-system. To ensure that the source or destination at the right side of the sequence diagram is a known quantity (or can be correctly emulated), I have taken to teaching developers to implement these sequence diagrams from right to left.

As a vehicle to discuss sequence testing, let’s use the following sequence diagram.



While coaching XP teams, I now use sequence testing to help teams learn and discuss test first design:

1) Any pair starting on a story or a task creates a simple, hand written sequence diagram. Acting as a road map to the pair, the non-driver uses the diagram to help steer the team (as well as using it as a reference for how much the team deviates from the defined path while implementing one or more classes shown on the diagram).
2) Starting from the right side of the diagram, the pair creates the first test case for the class farthest to the right.
3) The team then continues moving to the left, always performing the following mantra after each interface for any class has one or more working tests: (U)pdate from CVS, (B)uild all, (E)xecute all unit tests, and (C)ommit.
4) This process is continued until there are one or more test methods for each public method exposed for each class method in the sequence diagram.

Quick and Dirty

The diagram(s) are hand written and flexible; they are meant to be a vehicle to discuss which classes are to be involved and the responsibility of the classes in this sequence of events. I find that they help developers visualize the design by contract idea that is so much a part of test first design. If at any point the diagram becomes obsolete or messy, rip it up and quickly create a new version. If the diagram has become so complex that if takes more than ten minutes to recreate, check the air for bad design smells (or smells that indicate the team has strayed to far from that which is the simplest).

Once I have a pair following these steps, I ask them to examine the relationship between the test code in a test class and the code in the class immediately to the left of the test class (in the upper portion of our example sequence diagram – I do not promote adding test classes while sequence testing, but I sometimes draw these diagrams as a teaching tool). Because the test class and method combine to emulate a calling client, that client being the class to the left of the test class, a majority (if not all) of the code for the client already exists in the test class in a form that is understood by both members of the pair.

A Real World Experience

In one coaching situation with a team of 15 developers, there was the usual fear that pairing and test first would "make us go slower." This situation being the rule and not the exception with teams new to XP, management also feared that the team progress would slow and resources would "be wasted."

As the developers started seeing that while moving from right to left, code could be moved from test classes to sequence classes, they began to realize that only a small effort really was for testing only. Also, as the unit testing moved from an unnatural scripting experience, to an enjoyable development experience, the tests became a common topic of discussion. The tests being a de-facto documentation of the code, the team was now engaging in daily design discussions.

Another benefit (known to the test first community) was the ability to fearless rip into problematic and bloated classes. When a team embraces the metric of one or more test methods for each public method, design smells often reveal themselves earlier if the developers feel the need to write a large number of tests for any one method or class.

Why Save a Picture of Dorian Gray?

On my current project, we now staple the sequence diagrams to the stories. At times, when a pair does some development that the team deems significant (as defined during a standup meeting) it is captured electronically and stored in CVS as a snapshot of our design at a point in time. We have agreed that we will not go back to update any pictures, but we have decided that we do like the idea of having some history to reference, even if it may become a faded view of the current code.

3 CONCLUSION

I think that most developers that have gravitated toward test first design, have done so because it matched (or formalized) their development habits. I was fortunate enough to start developing with a small group that most often went to pictures before typing.

I believe that test first design does not mean pairs should or cannot use diagrams to communicate problems and solutions. Granted, there are development efforts that simply do not need a diagram. Indeed a simple list of tasks can provide the same road map that a sequence diagram does. I have found that pictures most often illicit more dialog than lists, so I have moved in that direction while coaching as well as developing.

I am intrigued as to what level of discussion around diagramming as it relates to test first design and agile methodologies in general. I look forward to hearing what level of diagrams other XP practitioners are using. I trust that the common sense nature of the agile movement will not (and has not) rejected communicating with pictures. I hope to see a bit more discussion around this topic in the community.

Patterns for Java Program Testing

Marco Torchiano
Department of Computer and Information Science (IDI), Norwegian University of Science and Technology (NTNU),
N-7491 Trondheim, Norway.
Tel. +47 735 94489, Fax +47 735 94466, Email: Marco.Torchiano@idi.ntnu.no

1 Introduction

The NIH (Not Invented Here) syndrome often appears in software development; it makes people keep reinventing the wheel all the time. Fortunately the recent trend is to reuse existing solutions either as software component or as known designs. Patterns fall into this latter category. Patterns are reusable solution to known problems in a well-defined context [2].
There are three main types of patterns [3]: idioms, design patterns, and architectural patterns. Idioms leverage language specific features and are fairly low level. Design patterns deal with more complex structures, such as groups of classes and associations. Architectural patterns address system-level issues.
Historically, patterns have been used as a tool to build systems. Recently their use has been extended to comprehension and testing. For instance they can be used to describe typical bugs and possible solutions [4].
There are several basic Java testing techniques, which are in common use among the programmers. This paper proposes to use patterns to describe such techniques and to arrange the patterns in a pattern language. We are concerned with class-level and package-level tests. We focus on idioms and basic design patterns.

2 A pattern language

Extreme programming (XP) [1] advocates writing tests first. When focusing on Java unit testing, we identify two possible levels of detail: class and package.
A class-level test is a test design to check the feature of a single class; i.e. to checking the semantics of a class. Unfortunately, often the behavior of a class depends on several other classes, so the next step is a test of a group of classes lying in the same package.
We describe some of the most common techniques used to write test code at the class and package level. in the form of patterns.

The most common pattern is the *main method idiom*. It addresses the problem of where to write the code that initiates and drives the test. Each class can have a `public static void main(String [] args)` method, being invoked when the argument of the Java virtual machine is the class that contains such a method. Therefore we can provide each class with such a method that enables us to test it.

When we need to test the combined functionality of several classes, we have to decide where to put the test code. Following the separation of concepts principle, a new class should be created to host this code. Two possible patterns are based on this idea: the *internal class design pattern* and the *external class design pattern*.
In the former the class is inside the package that contains the classes to be tested. In the latter it is external. The pros and cons of these two solutions are based on the observation that an internal class has access to the package details while an external one does not.
On the one hand, the internal class pattern provides more insight into the classes to be tested, but is can make use of features that are not available to a “normal” client of the package. On the other hand, the external class pattern plays the role of a typical client of the packages thus providing a more realistic scenario. Both patterns can make use of the main method idiom to have a starting point.

When it comes to checking the results of a test there are different techniques.
The most naïve is the *toString method idiom*. The method `public String toString()` is defined in the base class `Object`. It can be redefined to provide a customized representation of an object. The customized `toString` method can be used both at the end of a test and during it, in order to observe the status and contents of objects involved in the test.

A somewhat more sophisticated way of checking the outcome of a test, is comparing the result objects to other objects that represent the expected result. In this case the *equals method idiom* can be applied. The method `public boolean equals(Object)` is defined in the base class Object; it can be refined to customize the comparison between objects.

The patterns described so far can be combined and used together. There are several possible combinations that can be represented in a pattern language. Figure 1 describes a pattern language based on the use relationship between patterns.

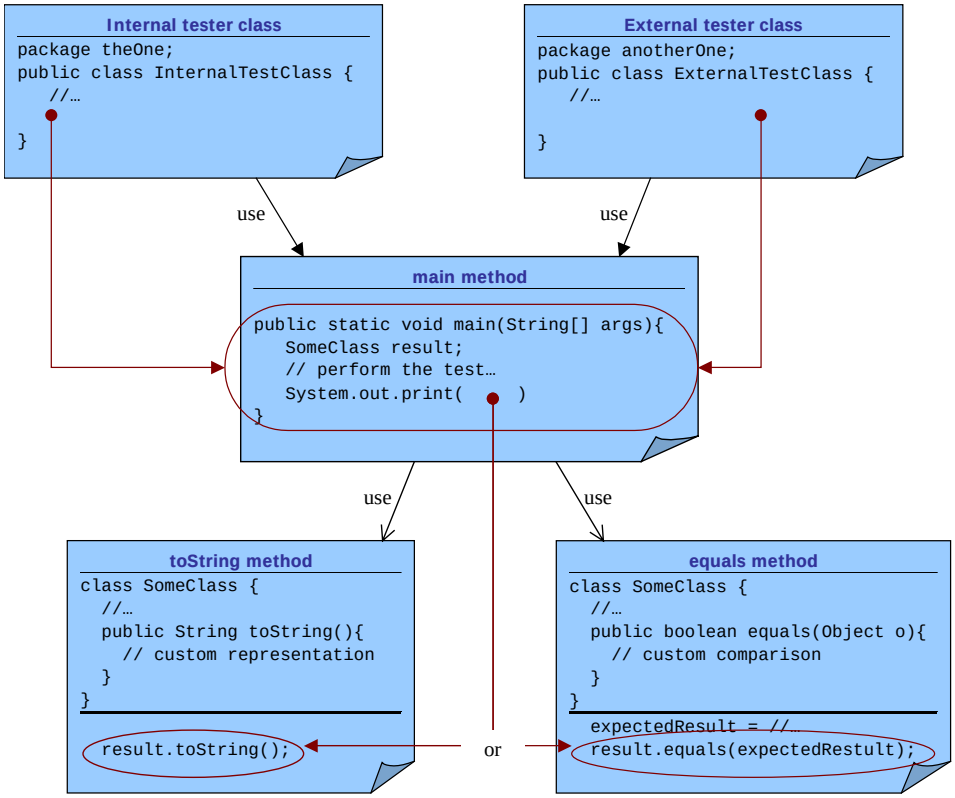


Figure 1: Java testing pattern-language.

3 Conclusions

The patterns here described represent just an excerpt from the many that can be used to run tests. Their usage context can range from home made testing to intensive extreme programming. The testing patterns can be used to improve the use of testing suite frameworks such as JUnit [5]. We plan to collect test patterns from real world projects, and arrange them in a more complete pattern language.

4 References

[1] K. Beck. “Embracing change with extreme programming”. IEEE Computer, 32(10):70-77, October 1999.

[2] E. Gamma, et al. “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, Reading, MA, 1995.

[3] J. O. Coplien "Software Design Patterns: Common Questions and Answers", in: Rising L., (Ed.), The Patterns Handbook: Techniques, Strategies, and Applications, Cambridge University Press, New York, January 1998.

[4] Eric E. Allen “Bug patterns: An introduction” available at: http://www-106.ibm.com/developerworks/java/library/j-diag1.html, February 2001.

[5] JUnit home page, at http://www.junit.org/index.htm, last visited on April 17, 2002.

Detailed Description of Patterns

Here we present the java test patterns in detail. We adopt the description of pattern propose by Coplien in [3]. Patterns are described in terms of:

- name,
- context,
- problem,
- forces,
- solution,
- examples,
- forces resolution, and
- design rationale.

Name	main method
Context	A Java class.
Problem	Where to write the code that drives and initiates the test for a class.
Forces or tradeoffs	Test must be easy to run. Code must be able to access all required features.
Solution	Put the code in the public `static void main(String[] args)` method of the class.

Examples	`public static void main(String[] args){`
	`  SomeClass result;`
	`  // perform the test…`
	`  System.out.print("result is..");`
	`}`

Force resolution	The test code can be easily invoked. The method has access to all the features of the class under test.
Design rationale	Each class can have a `public static void main(String[] args)` method, it is invoked when the argument of the java virtual machine is the class that contains such a method.

Name	toString method
Context	A Java class that represent the result of a computation.
Problem	How to check if the final and intermediate results of a test are the expected ones.
Forces or tradeoffs	Result of the test is represented by the internal state of an object Such a state must be checked.
Solution	Use the toString method to provide a customized representation of a class, such representation can be printed (and checked manually) or compared against an expected result.

Examples	`class SomeClass {`
	`  //…`
	`  public String toString(){`
	`  // custom representation`
	`  }`
	`}`

Force resolution	The internal state is represented as a String
Design rationale	The method `public String toString()` is defined in the base class Object. It can be redefined to provide a customized representation of an object.

Name	equals method
Context	A Java class that represent the result of a computation.
Problem	How to check if the final and intermediate results of a test are the expected ones.
Forces or tradeoffs	Result of the test is represented by the internal state of an object Such a state must be checked.
Solution	Define the equals method to be able to compare the actual result against the expected result.

Examples	`class SomeClass {`
	`  //…`
	`  public boolean equals(Object o){`
	`  // custom comparison`
	`  }`
	`}`

Force resolution	The result remains in the internal state but it is compare to the expected one.
Design rationale	The method `public boolean equals(Object)` is defined in the base class Object; it can be refined to customize the comparison between objects.

Name	internal tester class
Context	A Java package.
Problem	Where to place the code to test the classes of the package.
Forces or tradeoffs	Separate test code from operation code. Code must be able to access the required features of the classes. Play the role of a package client.
Solution	Add a new class in the same package where the classes to be tested are. This class contains all the code to perform the tests.

Examples	`package theOne;`
	`public class InternalTestClass {`
	`  //…`
	
	`}`

Force resolution	All the test code is inside a class separate from the rest of the code. Being in the same package the class can access most of the classes' members. It's not a real client since it has visibility on far more elements than an usual client.
Design rationale	The separation of concerns is achieved by confining the test code in a specific class. It is inside the package, therefore it has access to all the details of the classes that are to be tested. Because of this it cannot represent if a realistic way a typical client of the package.

Name	external tester class
Context	A Java package.
Problem	Where to place the code to test the classes of the package.
Forces or tradeoffs	Separate test code from operation code. Code must be able to access the required features of the classes. Play the role of a package client.
Solution	Add a new class in the same package where the classes to be tested are. This class contains all the code to perform the tests.

Examples	`package anotherOne;`
	`public class ExternalTestClass {`
	`  //…`
	
	`}`

Force resolution	All the test code is inside a class separate from the rest of the code. It has a limited access to the classes since it's outside the package. It has the same visibility on package elements as an usual client.
Design rationale	The separation of concerns is achieved by confining the test code in a specific class. It is outside the package, therefore it has not access to all the details of the classes that are to be tested. For this reason it represents in a realistic way a typical client of the package.

Testing ideas and tips from the battlefield

WTiXP 2002 Position Paper

Piergiuliano Bossi, Giannandrea Castaldi, Alberto Quario
Quinary S.p.A.
http://www.quinary.com
+39 – 02 – 30901535
p.bossi@quinary.com

Introduction

This paper provides some testing ideas and tips based on the experience that we have gained during our last projects. We will discuss how to avoid some problems that arise with acceptance testing, patterns and techniques related to mock objects usage and we will report about testing GUIs.

Acceptance Testing

Avoid BTUF

We have noticed that it is possible to make a certain kind of mistake related to acceptance tests we have called BTUF (Big Test Up Front). Developers bump into BTUF when, beginning to work on a story-card, they try to accomplish the acceptance test as the first thing. By doing so, they avoid tackling the several user-story complexities step by step, but they head for the green bar in a whole *piggy mess*. This means that development is no longer guided by many small unit tests but by a unique big acceptance test.
The origin of BTUF does not depend on the lack of specific XP practices, but rather on the absence of incrementality in the whole process. Indeed, if we observe the way programmers fall into BTUF, we notice a test first approach: the story-card implementation starts from the acceptance test followed by coding and refactoring. Moreover, at the beginning of the story-card no up-front design is carried out. But that's not the whole story.
The above process falls into hacking because it produces the following issues:

- Long refactoring: since the aim was just to pass the acceptance test, you have messed up the code a lot and now you are left with many bad smells to get rid of. The code may be difficult to manipulate and it is hard to discover recurring logics.
- Time consuming and rare integrations: since you don't carry out small increments, you need to refactor the code in many places – therefore during integration you have to solve many conflicts; besides, developers integrate less often because they are stuck with red bars.
- No continuous efforts: the XP heartbeat (small test, small code, small refactoring) is broken by the need to write and immediately pass the acceptance test.
- Fear: working by big steps leaves the system in a non-consistent status for a long time and brings developers to fear changing the code.
- More debugging: since there aren't enough unit tests to cover functionalities required by the card, when the system doesn't work you may need to debug for many hours.

Evolutionary approach to Acceptance Testing

Instead of writing the acceptance test, getting the red bar, putting it away and forgetting about it, we prefer to approach its implementation in an evolutionary way. What we do is to follow the test-first technique focusing each time on business value delivering. To do

so we need to proceed test by test with little increments, writing each time the smallest test that delivers value to the customer.
At each step the developers are about to write a test, they must ask themselves which is the minimum functionality to be added in that particular moment. As each step is small, after some time you will have a convergent succession of many unit tests that cover the requested functionality. So your system will be ready for satisfying the acceptance test, and this will manifest itself as a natural consequence of the tests set.
However, the acceptance test is not always the last step towards the user-story fulfillment: it may happen that one or more unit tests are worked out after implementing the acceptance test. Sometimes the acceptance test itself suggests tests the developers had not identified before.
In this way, there is no real contraposition between unit and acceptance test: developers write many small tests we can define *functional*, because necessary to discover a path that brings to user-story accomplishment.

Testing patterns

Programmable mock object

Whenever we feel the need to test a class in isolation, that is working with mock objects, we have found ourselves implementing a similar pattern each time. Suppose you have to test a class A that collaborates with a class B:

- We extract the interface from class B;
- We create a mock implementation of class B, say MockB, having all the public methods doing nothing or returning nulls;
- Each time we have to write a test for class A we extend the behaviour of MockB adding 3 different kind of methods:
    - `simulate<action>`: at the beginning of the test it permits to program the expected object behaviour (i.e. `simulateClickOnCloseButton()` on a GUI mock object);
    - `last<data>`, `is<propertySet>`: at the end of the test it permits to verify what A has done which has an expected impact on B;
    - `<interface method>`: basic and required class B behaviour.

Doing this way we have seen several advantages:

- Mock objects grow up incrementally based on test demands;
- Tests are more readable.

This is an example of a GUI mock object in which we are testing that the controller reads a string from the GUI and it executes a search on the mock search engine:

```
public testSearchOnClickButton()
{
      MockGui mockGui = new MockGui();
      MockSearchEngine mockEngine = new MockSearchEngine();
      Controller controller = new Controller(mockGui, mockEngine);

      mockGui.simulateSearchFor("john white");

      assertEquals("last search on engine", "john white",
```

```
            mockEngine.lastSubmittedSearchString());
}
```

Test first design

Adding new features test-first

Every time we are adding new features to a system we start unit testing from a specific object: we write one test, the code that makes it work and so on. Doing this way, we may modify an existing object or we may need to add new methods on the other objects that collaborate with it. As we often isolate the tested object with mock objects, we only add such methods on the mock object. Therefore, our message is: "When you are testing an object and you need to modify its collaborators, modify the mock objects first".
When we get the green bar we have two possibilities:

1. Pass to add the methods that were only on the mock object on the real object;
2. Remain on the object we are working on and continue to add methods on the mock objects that play the role of the real ones.

Considering the context we choose a way rather than the other:

1. We choose the first approach when we see a special value in working through vertical slices of our system, that is when we want to see immediately the consequences of adding new whole features;
2. We choose the latter when we see that the object we are working on must change a lot and then we prefer to stay focused on it. Doing this way, we can easily modify the mock objects and we can propagate the modifications to the real implementations later.

Experience reports: testing GUI

Testing GUIs is difficult, as you must face a paradigm crossing. In our last project we tried some slippery roads and finally settled on a three level structure inspired by the MVC paradigm:

Controller → Presentation → GUI

Our intent was to separate the business logic from the actual GUI representation (i.e. a Java Swing frame).

The Controller has the business logic and send/receive information to the Presentation (i.e. inserting some data in a grid or handling a button press in a window).
The Presentation provides to the Controller an abstraction of the real GUI through some services that map Controller actions onto graphic widgets; it provides also some hooks to allow the Controller to register on selected events.
What differentiates our design from MVC is the fact that we haven't applied the *Observer/Observable* relation: data Model is handled directly by Controller, which manages the updates too. We have never needed to implement several notified Views reacting to a modification in the same Model portion.
We have also introduced the Presentation abstraction because it allows testing Controller in isolation with a mock Presentation that is not interfaced with real GUI widgets. This approach has several advantages: tests are faster, we don't need many complex GUI-testing tools, orthogonal issues emerge more clearly, etc.

Using mock Presentation objects to test the dynamic behaviour of Controllers, GUI testing is reduced to verifying positions and properties. We have developed some utility objects to ease the testing burden.
`ComponentsPositionComparator` offers services for positional testing:

- Alignment
  - `public boolean areBottomAligned(JComponent aComponent, JComponent anotherComponent)`
  - `public boolean areLeftAligned(JComponent aComponent, JComponent anotherComponent)`
  - `...`
- Positions
  - `public boolean isNextToTheBottomBorderOf(JComponent aComponent, Container aContainer)`
  - `public boolean isNextToTheRightBorderOf(JComponent aComponent, Container aContainer)`
  - `...`
  - `public boolean isToTheEastOf(JComponent aVerifyingComponent, JComponent aTargetComponent)`
  - `public boolean isToTheSouthOf(JComponent aVerifyingComponent, JComponent aTargetComponent)`
  - `...`

We have also developed a more high level object named `AsserterPositionComparator`, that uses `ComponentsPositionComparator` to minimize the effort required to code assertions in tests. `AsserterPositionComparator` automatically composes the message and logs useful information in case of assert failure. Some of its methods are:

- `public void assertAreBottomAligned(JComponent aComponent, JComponent anotherComponent)`
- `public void assertIsNextToTheRightBorderOf(JComponent aComponent, Container aContainer)`
- `...`

Although these utility objects, iteration after iteration we have reached a point where we doubt that test-first design could be fully applied to GUI development with profit. At the beginning we have developed all our GUIs test-first and this has caused several problems due to volatility of customer requests. Every time that we have submitted to the customer a new GUI version based on his requests he has gained more inspirations and consequently he has had more suggestions for us. These modifications were easy to code, but updating the corresponding tests was expensive. As time goes by we have recognized that we were writing regression tests with a test-first approach: this is definitively not the intent of the test-first technique and it should be avoided. There is such a little value in developing these tests before the actual code.
For this reason the last GUIs have been implemented following a different process: firstly we have developed the code coming to a point where the customer was satisfied by the result, and then we have written the corresponding tests. Our intent for the future is to automate the development of these tests updating them after each visible GUI modification.

DiPS: Filling the Gap between System Software and Testing

Sam Michiels, Dirk Walravens, Nico Janssens, Pierre Verbaeten
DistriNet, Dept. of Computer Science, K.U.Leuven
Celestijnenlaan 200A, B-3001 Leuven, Belgium,
+32 16 327640
{Sam.Michiels, Dirk.Walravens}@cs.kuleuven.ac.be

ABSTRACT
Testing system software (such as protocol stacks or file systems) often is a tedious and error-prone process. The reason for this is that such software is very complex and often not designed to be tested. This paper presents DiPS, a component framework, which forces to develop *testable* software, and DiPSUnit, a JUnit extension, to test DiPS units in a uniform way. Although non-trivial test support is provided, using DiPSUnit keeps testing simple and intuitive thanks to the DiPS approach.

Keywords
Testing, framework, component software engineering

1 INTRODUCTION

Testing system software, such as a protocol stack or a file system, is a complex, tedious and error-prone task. The basic problem is that, for performance reasons, system software is often designed as a monolithic block of multi-threaded software. This prevents such software from being tested properly because of two reasons: first, it is very difficult to isolate the basic building blocks as stand-alone units that are independent from each other. Second, concurrency code, which is introduced in such multi-threaded system software, often crosscuts the code [4].

This paper presents DiPS (Distrinet Protocol Stack) [6], a component framework we have built to support protocol stack development. DiPS forces to deploy four design principles, which are important prerequisites to develop adaptable and *testable* software. As proof of concept, we have developed the DiPSUnit test framework, which is an extension of JUnit [3], specifically to test DiPS units.

The rest of this paper is organised as follows. Section 2 presents four essential chararcteristics of *testable* software. Sections 3 and 4 present DiPS and DiPSUnit, two frameworks we have developed to proof our ideas about software development and testing. Conclusions and some open points of discussion are formulated in section 5.

2 DESIGN FOR TESTING

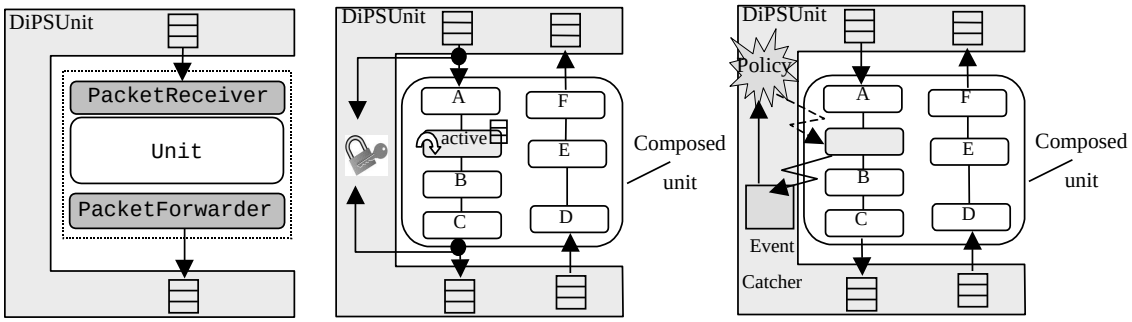
XP could be the victim of its own success: testing complex (system) software in XP could lead to *test hell*, where test code becomes so complex and unmanageable that it needs testing… One way to deal with this is to design *testable* software, i.e. software that is designed such that it can be tested easily. We distinguish four essential characteristics of so-called *testable* software:

- ***Modularity***: it is important that the design reflects fine-grained (singular) units as separate entities, to allow unit testing in isolation [5]. A collection of units can be grouped together into a *composed unit*, which is treated the same way as a singular unit.
- ***Independent units***: to allow transparent substitution of units and to reduce the risk of unexpected side effects (when units are replaced) during acceptance testing, it is essential that units are independent from each other.
- ***Separation of concurrency from functionality***: traditional system software is difficult to develop, understand, maintain, adapt or test. One of the major reasons is that concurrency code crosscuts the functional code [4]. Separating concurrency from the rest of the code facilitates development and testing because programmers can concentrate on one aspect at a time. A protocol stack developer should concentrate on creating a header parser or a packet fragmenter without being distracted by non-functional aspects such as concurrency (parallellism). Because of this separation, unit tests can be done *first* in a single-threaded context, and multi-threading can be added *later*, without changing any unit code.
- ***Uniform unit interface***: reduce the unit's interface and share the same interface type as much as possible. This facilitates reuse and raises the software's level of abstraction. This feature, combined with modularity (i.e. composed unit is again a unit), allows a uniform testing approach for both singular units and composed units.

3 THE DIPS FRAMEWORK

DiPS is a Java component framework based on units that are connected as a pipe-and-filter architecture. The framework supports the development of system software such as protocol stacks or file systems. Communication between DiPS units is intercepted by the framework. This forces units to communicate anonymously (***independent units***), since they have no explicit notion of other units in the system.

A DiPS unit is an object-oriented entity with a very specific (fine-grained) responsibility (***modularity***), such as a packet



Consistent test approach in DiPSUnit: a singular DiPS unit, a composed unit with internal concurrency and a composed unit which sends (and receives) control events

fragmenter or a header parser. DiPS units can be grouped together into composed units (such as a protocol layer in a protocol stack). A distinction has been made between *purely functional units* and *concurrency units*. This ***separation*** allows the concurrency model to change, independent from the functionality in the system.

All DiPS units process (only) packets, which are delivered via a ***uniform unit interface***. These packets can enter and leave a DiPS unit through one or more entry and exit points. A singular DiPS unit (such as a fragmenter or an encryption unit) with one entry and one exit (`PacketReceiver` and `PacketForwarder`) is shown in detail in the left figure. Next to the data (packet) flow there is a control flow that allows anonymous inter-unit control communication via *DiPS events*.

4 PROOF OF CONCEPT: DIPS UNIT TESTING

Thanks to DiPS, the DiPSUnit framework can provide a uniform way to test singular units as well as composed units (see figure). This keeps testing very intuitive and simple. However, the provided support for testing units in isolation in the presence of concurrent behavior and external control events is not trivial.

The concurrency (*active*) unit in the middle figure has an internal thread and a buffer to store incoming packets. This decouples the packet flow in two parallel flows. When such a unit is present within a composed unit, a test must be suspended until all packets have arrived or until a timeout occurs (to avoid being suspended forever in case of an error). DiPSUnit offers a monitor that blocks until all packets are processed (even in the context of internal packet removal/creation).

DiPS units can exchange control information by using DiPS events (right figure). To test a unit in isolation, all control flows must be intercepted. Although the stub mechanism is simple, transparently introducing stubs is not always trivial. DiPSUnit offers support to uniformly deal with external events. A test developer can describe how to respond to a given event by creating a Policy (which acts as a stub). The substitution of control flow functionality is transparent for the code under test. This reduces the risk of introducing errors when stubs are replaced by the actual functionality during acceptance testing.

For a detailed description of DiPSUnit we refer to [7] [8].

5 CONCLUSIONS

The combination of JUnit, DiPS and DiPSUnit seems very promising. JUnit offers the basic infrastructure to develop test cases and test suites. DiPS facilitates unit testing because it forces to create modularized architectures and because it allows units to be replaced without changing any code. Thanks to this support, DiPSUnit can consistently test DiPS units, from fine-grained to composed unit level. However, developing test cases is still intuitive and simple.

Relevant points of discussion are:

- Software engineering techniques, such as design patterns [2] and refactoring [1], do help in creating 'good' software, and the xUnit test framework helps in testing software. However, we claim that infrastructure support is required (such as a component framework) to *force* design techniques to be applied.
- What other software design principles facilitate or hinder testing?

ACKNOWLEDGEMENTS

This research has been carried out in order of Alcatel Bell with financial support of IWT (project SCAN #010319).

REFERENCES

1. M. Fowler. *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
2. E. Gamma, e.a., *Design patterns: elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
3. E. Gamma, K. Beck, *Test infected: Programmers love writing tests*, http://www.junit.org/, 1998.
4. G. Kiczales, e.a., *Aspect-Oriented Programming*, In proceedings of ECOOP'97, 1997.
5. T. Mackinnon, e.a., *Endo-Testing: Unit Testing with Mock Objects*, XP2000, June 2000.

6. F. Matthijs, *Component Framework Technology for Protocol Stacks*, Ph.D. thesis, K.U.Leuven, 1999. (Available at http://www.cs.kuleuven.ac.be/~samm/netwg/dips/)

7. S. Michiels, D. Walravens, e.a., *DiPSUnit: an Extension of the Junit Test Framework for DiPS*, Tech. Report CW-333, K.U.Leuven, Dept. Comp. Science, 2002.

8. S. Michiels, D. Walravens, e.a., *DiPSUnit: A JUnit Extension for the DiPS Framework*, To appear as experience report in XP2002. K.U.Leuven, Dept. Comp. Science, 2002.

Web Systems Acceptance Tests and Code Generation

Eduardo Aranha[1] and Paulo Borba[2]
Informatics Center
Federal University of Pernambuco
Recife, Brazil

Introduction

In Extreme Programming (XP) [2], acceptance tests are used to prove that the application works as the customer wishes. The available test languages offer low level of abstraction and legibility, because they are based in languages like Visual Basic and XML. GUI capture and playback tools facilitate the creation of test cases, though they have many limitations to program and maintain the test cases [1].

Acceptance tests interact with the GUI (Graphical User Interface) of the system, simulating the actions of users and verifying the information content presented. In Web systems, for example, the GUI is composed of Web pages and its components, like frames, links and images. In that way, the information about the GUI structure and behavior of a system can be found and extracted from its acceptance test cases, making possible the generation of part of the GUI code.

This paper presents a language and an environment to program Web Systems acceptance test cases. Code generators are presented to improve productivity and to motivate the XP practice of creation of these tests before the implementation of the proper system.

The WSat Language

The language we defined, WSat (Web System Acceptance Test), aims at a high level of abstraction and reuse, explicitly expressing aspects related to the GUI structure of the tested systems like, for example, Web pages, forms, links and texts. This is done by defining types that represent web components. In the Figure 1, we can see the initial and response page of a simple search system of Web documents.

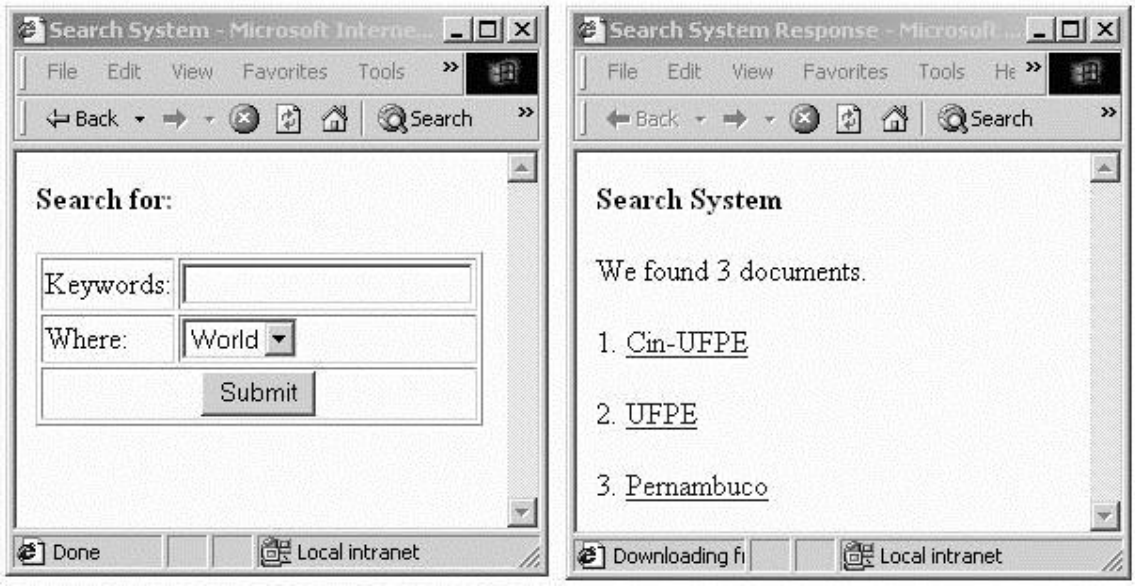


Fig. 1 – Initial and response pages of a search system.

[1] Supported in part by IPAD. Electronic mail: ehsa@cin.ufpe.br.
[2] Supported in part by CNPq, grant 521994/96-9. Electronic mail: phmb@cin.ufpe.br.

To test this system, we initially define the type `InitialPage` to represents Web pages with title "`Search System`" and an HTML form as defined by the type `SearchForm`:

```
static WebPage InitialPage {
    title = "Search System";
    SearchForm searchForm;
    ...
}
WebForm SearchForm {
    name = "searchForm";
    method = "POST";
    EditBox {
        name = "keywords";
        value = "";
    } keywords;
    ...
}
```

WSat have predefined types like `WebPage`, `WebLink` and `WebForm`. The `WebPage` type, for example, represents all possible Web pages. The defined type `InitialPage` represents all possible Web pages that satisfy its defined properties. To test the response page of the system, we define the type `ResponsePage`, as shown bellow.

```
WebPage ResponsePage {
    title = "Search System Response";
}
```

As we can see, we do not use the WSat keyword `static` in the definition of the type `ResponsePage`. This keyword indicates Web pages that are not generated dynamically by technologies like Servlets or JSP. This and others information not shown here are used only for code generation purpose. To verify the dynamic content of Web page and the system behavior, we create test cases as shown bellow.

```
testCase testSearchSystem {
    String url = "http://www.searchsystem.com";
    InitialPage page = [InitialPage] getWebPage(url);
    SearchForm form = page.searchForm;
    form.keywords.value = "ufpe";
    ResponsePage resp = [ResponsePage] form.submit();
    WebLink link = resp.findWebLinkByURL("http://www.ufpe.br");
}
```

The test case `testSearchSystem` requests the page at URL `"http://www.searchsystem.com"`, verifying if it conforms to the initial system page (`[InitialPage]` operator). Then, the test simulates the form submission with the "`ufpe`" keyword by calling the `submit` service defined in the `WebForm` type. To verify if the system give the correct answer, we look for the link "`http://www.ufpe.br`" in the response page.

As we can see, properties defined in WSat types are used to test the components of Web systems. In order to simulate the users actions, we can use the services of the WSat predefined types. Some of these services are used to test dynamic content of Web pages. We can use, for example, services like `findWebImageByName`, `findTextByRegExp` and `findWebLinkByURL` to retrieve components that represent images, texts and links with the given properties.

In order to validate WSat, we created an execution environment for it by compiling WSat programs to Java code.

Code Generators

In order to reduce development efforts with tests, we implemented two code generators. The first one is a test code generator, which generates WSat code from HTML prototypes used to validate the requirements. WSat types are generated to represent the components found like Web pages, frames, forms and links. As we can see, a lot of code to test GUI structure is generated. However, the code to test the system behavior could not be generated yet by this test code generator.

WSat types contain information about the GUI structure of the tested system. From this information, we can generate part of the GUI code using a system code generator. For example, considering GUIs implemented with Servlets, we can generate one Servlet for each Web page tested by a WSat type in the test code. The generated Servlets could be associated to response templates based in the HTML prototypes. Unit test classes for the Servlets and other types of code are generated, too.

The system code to be generated is dependent of the development environment used. For this reason, the system code generator was build following the Visitor design pattern [3]. Each visitor manipulates the syntactic tree of WSat programs and it has a specific functionality, like to generate Servlets or to generate JSP files. In this way, we can specialize the code generator to new development environment building new visitors.

Development Methodology

Aiming an efficient use of WSat and the code generators, some activities need to be added to XP methodology. In Figure 2, we can see the proposed changes in the XP flow.

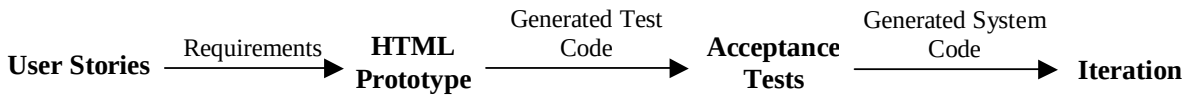


Fig. 2 – Changes in the XP flow chart.

HTML prototypes are created from the requirements found in the user stories. Then, the test code generator is used to generate part of the acceptance test code. The test programmer completes the WSat code needed in the actual iteration. The generated WSat types are complemented and new types could be created to test more complex information. The test cases are written at this time, too.

From the WSat types created in the actual iteration, we generate the part of the GUI code to be developed using the system code generator. The generated code is afterwards used by the programmer to start the system development for that iteration. With few adjustments, the code generated for the system could be executed just like the HTML prototype. The programmer is responsible now for implement the system functionalities basically writing the code under the GUI layer.

Conclusions

Through experiments, we evidence that the type definitions written in WSat code has a high level of abstraction and readability, facilitating the test programming. The use of types to represent Web components becomes the test activity more interesting and partially similar to modeling activities, eliminating part of the traditional tedium existent in writing test cases.

Programs written in WSat could check the components and the behavior of Web systems GUI, given the supporting needed to do acceptance tests. To support other types of tests, like performance and stress tests, programs WSat could have embedded Java code. However, this type of code compromises the abstraction level of code.

With the developed code generators, it is possible to generate automatically part of the test and system code, improving development productivity and motivating the creation of acceptance tests before the Web system implementation.

In one experiment done, more than 30% of test code was constituted by the declaration of WSat types (GUI structure description). The test code generator could generate a good part of that code, reducing the initial effort to program the tests. And with relation to the system code, more than 4% of it was automatically generated. The time saved by the generators, in this case, was sufficient to program simple test cases. However, it is probably not possible when we have a substantial number of test cases.

We can explore in future works the association between Web pages, like links and form actions. These associations could permit the generation of other types of code, different from the actually generated. For example, may be could be possible to generate part of the acceptance test cases.

Referências

[1] M. Finsterwalder. *Automating Acceptance Tests for GUI Applications in an Extreme Programming Environment*. In XP2001, Sardinia, Italy.

[2] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[3] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

Are Extreme Programmers writing too many Tests?

Position Paper for XP2002 Workshop on Testing in XP

Frank Westphal
Am Brunnenhof 33
22767 Hamburg
Germany
+49-177-2343664
westphal@acm.org

Testing guru Brian Marick is of the opinion that XP'ers write too many tests (and that's not something he says often). Actually, Extreme Programming has successfully put into practice what the testing people have teached for decades already. But then, there is some appeal in Brian Marick's statement. The costs and benefits of tests have to be balanced very carefully on every project. As much as XP will change the role of the traditional tester, the responsibility of comprehensively testing our own code has also changed the required skill-set for most software developers. And most of us have only begun to learn how to test in an effective and economic way.

This paper summarizes a couple of related questions and ideas about testing in Extreme Programming that I'd like to see discussed and explored during the workshop.

The next maintenance hurdle

XP teams generate as much or even more test code than functional code. In my experience, it's not uncommon to end up with a 1:1 ratio in total, and a ratio of 2:1 to 4:1 for individual parts of the software (which either implies that there are untested corners in the system, or that there is indirect testing going on to some degree).

The danger here is that the sheer amount of test code hinders exactly what it set out to support: the mobility of the code. While many non-XP teams have not been able to maintain their design documentation (if any), the same risks still stick with the design documentation in XP: the unit tests. We have already seen the first signs of XP teams complain about slowing down because of the fragile nature of their test suite. How can we help those teams?

On the surface, the solution to this dilemma might be trivial. The same quality standards apply for test and functional code. This suggests we just eliminate any code duplication cropping up in the tests and write only tests that reveal their intention. But there is more. There is a fine balance to writing tests that give enough confidence in the fitness of the program and at the same time keeping these tests as soft as possible to ease future refactoring moves. What are the recurring patterns here, what are the anti-patterns?

Using and abusing mock objects

Mock objects have become tragically hip since their presentation at XP2000. While they are indeed a big step towards isolation testing, they also have become the hammer in search of a nail on some projects that I know. People on these projects decided for some reason to test each and every class

independently of their collaborators. While this approach is fine for many hard testing problems, it quickly leads to a large number of interfaces and mock classes to be maintained.

While the opposite approach of testing classes in small units of closely collaborating classes can easily lead to the problems of micro-integration testing, in practice, we most often have to strike the right balance between the two extremes. What are the boundary conditions of when to use and when to avoid mock objects?

Related to the maintenance issue, generated mock objects in particular tend to specify the behaviour of the unit under test in too much detail. My experience is that the smaller the unit under test, the more details will typically be hard-wired into the tests which makes it almost impossible to refactor the code without breaking a test. After some personal experience of overusing mock objects myself, I suggest to test classes in small clusters as long as possible. What are other experiences?

Setting the quality bar

In XP, it's a business decision and therefore the customer's call to set the quality bar. From there, it's the developers' call to balance the costs and benefits of tests through constant reflection about their practices. However, in most conventional project settings, there is almost no way and also no person to set the quality bar. Therefore, more often than not, different team members end up with a different understanding of their testing priorities. There is no vision for the desired software quality in such teams which can be a big problem.

To my surprise, I learned the other day that setting the bar on an XP project goes almost unnoticed. First of all, the desired quality is encoded in the criteria of the acceptance tests provided by the customer role. And even if bugs slip through, it's the customer's call which bug will be fixed at what point in time through the scheduling mechanism of iteration planning. The problem still persists for teams who have adopted only XP's testing strategy, though.

I believe that every project needs some quality management of one kind or the other. One of the management tools used foremostly on XP projects are big visible charts. What kinds of charts have you found useful to communicate quality related issues to the (i) customer team, (ii) development team, and (iii) management team?

Untested corners

Almost any XP team I know has come to some point where they faced an edge of their software which was particularly hard to test. While some of these teams have tackled the testing challenge with more or less perseverance, others have left this corner untested ... and regretted this in almost all cases. Some of these teams have gone back and retrofitted their program so that they could eventually automate the testing process in this area at least to some degree.

In my opinion, the automation of acceptance tests falls in this category in many places. Every now and then, XP practitioners have been careless about the practice of automated acceptance testing on their first XP project only to push it much more so on their next one. I certainly did.

Most of the teams that overcame the challenge of untested corners experienced some profound design insight. Typically, triumphing over these testing challenges makes for a good story at the campfire. If you have been on one of those teams, what has been this insight?

Testing, when is it enough?

-Erik Bos

Introduction

Kent Beck stated in the acknowledgements of his book [Beck02]: “suggesting you type in the expected output tape from a real input tape, then code until the actual results matched the expected result”. In a Programmers Handbook of 1976 [Volmac+76] programming starts with the following phases: “Problem definition, planning, main diagram, detailed diagram, construction of a small test set, desk checking, code”. The test set consists of the input cards and expected output cards. *Desk Checking* involves going through the flowcharts with the specified test input and calculating by hand the results. The result are then checked against the expected output. You can imagine desk checking is quite a job when you want to check more than a small test set. Although tests can be automated in the modern era the primary “test before you code” practise from the infancy of computer science still is a very basic and *required* step in developing software. However, modern software is often very complex and writing, coding and executing tests for all imaginable cases would only lead to asymptotic development. In this paper I propose checking the code coverage of your predetermined test set and adding tests until you touched all important parts of your code.

Coverage Experiences

My experience is with writing code and unit tests especially for infra structural (embedded) software. Since the infrastructure software is used by all other software modules in the system, it is essential the software is very stable and extensively tested. For one of the modules I wrote test code for, I decided to check the coverage of my tests with the *Generic Coverage Tool* [Marick95]. I was surprised by the low percentage of code covered. Some parts of the code, especially the parts that needed *Multicondition Coverage*[1], took some effort to cover with a test. GCT also shows *Relational Coverage*[2] also called *Boundary Condition Testing* [Kernighan+99]. This tends to find of-by-one errors and turned out to be a very effective way of finding bugs. I ended up with as much test code as production code. Other modules

[1] if (A && B) requires test cases of A and B true, A true and B false, A false and B irrelevant

[2] if (A<5) requires a test case where A = 5 to verify whether the ‘<‘ shouldn’t be a ‘<=’

even needed far more test code as production code to cover essential parts of the code. The parts I didn't manage to cover were reviewed with care.

Another good way to increase coverage is to generate random (valid) input and *Stress Testing* [Kernighan+99] a module. Large volumes of test input tends to test modules on buffer management and robustness against overflows. Also people tend to create normal test input, whereas computer generated test input covers the whole range of possible inputs.

Threading problems also tend to show up under *Stress Testing*. Especially when using multiple processors the chance of catching a threading problem increases significantly because the usage of shared resources is increased.

All these tests were programmed by hand, which is a lot of work. Another approach is generating additional test code. Using *State Based Testing* [Turner+93] this was performed for some modules in our system. It involved listing all states the modules could be in and generating tables of state transitions. The downside of this way of testing is that the modules contained so many states that the number of tests tend to explode. It also doesn't give you the insight of how to use the software because there are so many tests, which makes reading the test code hard.

Conclusions

Just writing tests and guessing you have enough tests written to test your module or system is not enough. What you need are tools telling you what you have tested and which part of your code is not touched by your tests. You should keep adding tests until all essential parts of your code is touched by the tests[3].

From a more philosophical viewpoint one may argue that about 80% of the code is there to handle "non-normal" program flow, i.e. errors and exceptions are therefore by nature hard to test. One may opt to dedicate less effort to this (large) part of the code and only select the critical parts of it. "Critical" is here defined as: "As long as the customer doesn't go ballistic when this or that part of the "robustness" code doesn't function".

I opt for writing all tests by hand. Although it is a lot of work and generates

[3] In XP projects code which can't be easily covered, would be candidate for refactoring since it is probably too complex.

extra effort to keep in sync with changing requirements, it weights up against the advantage of showing readers of the tests how your software is to be used and how it is structured.

Generating test code might be a solution for getting a good coverage of your code, but doesn't communicate very well because of the number of tests. Generating test input and *Stress Testing* does add more coverage with limited effort.

References

[Beck02] K. Beck. *Test-Driven Development by Example*, to be published.

[Kernighan+99] B. Kernighan, R. Pike, *The Practice of Programming*, Reading, Ma., Addison-Wesley, 1999.

[Marick95] B. Marick. *The Craft of Software Testing*, Englewood Cliffs, New Jersey, Prentice Hall, 1995.

[Turner+93] C. Turner, D. Robson, *State Based Testing and Inheritance*, Durhan, England, 1993.

[Volmac+76] Various Authors, *Handboek Automatisering, Programmering*, Utrecht, The Netherlands, Automation Centre Volmac, 1976.

Acknowledgements

Many thanks to Frank Pijpers and Dave Karetnyk for their comments and suggestions for this paper.

About the Author

Erik Bos can be reached at Erik@ErikBos.net.

Using Restrictive Approaches for Continuous Testing: Pre-Integration Checking

Martin Lippert
Apcon Workplace Solutions &
University of Hamburg
Vogt-Kölln-Str. 30
Hamburg, Germany

lippert@jwam.de

Stefan Roock
Apcon Workplace Solutions

Friedrich-Ebert-Damm 143
Hamburg, Germany

roock@jwam.de

MOTIVATION

When doing Continuous Integration the correctness of the code repository is crucial. Since correctness is very hard to prove, XP uses unit tests to approximate it.

If the code in the repository is broken, all developers in the team are harmed in a short period of time.

To achieve a full working common code base every developer should run the complete test suite of the system on her machine before the integration. Only a green bar signals the developer to integrate the changes into the common code base.

Generally this is done at the Integration machine. While using CVS or some other kind of source control system the situation is slightly different. The developer should download the complete code of the project again after the integration to see whether all tests are still green on the common code base.

INTEGRATION CHECKING

To ease this handling of CVS systems as well as ensure the test are green on the common repository some tools are available. They perform the tests on the common code base automatically on every integration. One of these tools is for example CruiseControl.

Nearly all of the tools do post-integration checking. This way problems with the code in the repository are detected. But they are detected *after* integration which means the repository is already broken. Again all the developers in the team may be harmed by the problem.

PRE-INTEGRATION CHECKING

We have followed another – more restrictive - path with a small tool for the integration process: the *CVS-Checker*. The CVS-Checker is a small plug-in for CVS executing an arbitrary ANT script *before* checking code into the CVS code base.

The ANT script simply merges the CVS code with the code to be checked in and compiles and tests it. Only if both operations were successful, the code is really checked into the common code base. Otherwise the integration got rejected. This way it is guaranteed that the code in the CVS system always is compileable and all test are green on the common code base.

EXPERIENCES

From the first view the pre-integration checking facility seems to slow down the integration process because the integration now needs more than a few seconds to finish. The additional delay is about 5 minutes for 2000 classes. Therefore for most projects the delay shouldn't be a problem. If it becomes a problem it could be a hint to split up the project into sub projects with a own CVS each.

Using the Pre-Checking facility developers tend to do smaller refactorings and integrate more often since this minimizes the risk of creating merge conflicts and being rejected by the CVS-Checker.

We think the described approach is especially useful for developers which still learn XP. Often they tend to be sloppy with test execution and the green (or red) bar.

But experienced XP developers also like the tool since it takes a bit of responsibility from their shoulders. Aside of that the pre-integration checker ensures a always running common code base which makes the work within the team as smooth as possible.

OPEN RELATED QUESTIONS FOR DISCUSSION

- What happens if tests last longer than a few minutes? Is it useful to define a subset of the unit tests as integration tests? Can be simply use the acceptance tests as integration tests?
- What about tests which need a proper configured infrastructure like DB or application server?
- Do other restrictive approaches exist which may be helpful for XP training?

- Does the restrictive approach restrict the flexibility of the XP team too much? In which contexts?

Retrofitting unit tests

Steve Freeman
M3P
12 Montagu Square
London
W1H 2LD, UK
+44 (0) 797 179 4105
steve@m3p.co.uk

Paul Simmons
Independent
6 Copse Close, Pattens Lane
Rochester, Kent
ME1 2RS, U.K
+44 (0) 7967 966203
pas@pobox.com

"You can't get there from here." Punch line to old joke.

"If you do not start adding unit tests today then one year from now you will still not have a good unit test suite." Don Wells[1]

ABSTRACT
In this paper we describe techniques that we have found helpful for adding unit tests to existing code that has been written without tests. The paper presents some common coding practices that make unit tests hard to retrofit, and why. For each practice we suggest minimal refactorings to open up the code for testing.

Keywords
Refactoring, Unit Testing, Legacy Code, Retrofitting

1 INTRODUCTION

Unit tests can be hard to retrofit to legacy code, but not as hard as many developers believe; for our purposes, "legacy" is working code that must be maintained but that has been written without unit tests. We believe that it is worth attempting to improve the internal quality of any system that matters and that unit testing is a key technique for doing so.

Relentless unit testing is a core practice in Extreme Programming (XP) [1]. It gives the developers the confidence to make changes as new requirements arise or new refactorings are discovered. Furthermore, when written before the code, unit tests are a powerful design tool that act as executable specifications; they concentrate the programmer's mind on what is really needed and help to drive the code towards good coding practice [2].

Many projects, however, convert to XP after starting with another methodology, which usually means that there is an existing code base that does not have a thorough unit test suite. The dilemma for the team is that they need a testing safety net to support the agile development practices they want to adopt but cannot write unit tests for the entire code base for two reasons. First, retrofitting unit tests is expensive, full coverage can easily take as much effort to write as did the original system without adding any visible functionality. Second, there is an obvious deadlock in that legacy code often needs some refactoring to make it testable, but refactoring should not be undertaken without tests in place to prove that it's safe.

Both problems must be addressed by a combination of skill and compromise. First, unit tests can be added incrementally, perhaps before changing a component for the first time during subsequent development. Combined with some judicious functional testing, the team can give themselves enough confidence to make progress, although at less than full speed, whilst improving the quality of the code. Second, our experience is that carefully fixing a few "code smells" without unit tests can give the developer enough leverage to bootstrap the writing of a full test suite. As the test suite builds up, the developers should look for opportunities to improve it as suggested by [3].

In this paper we concentrate on those careful fixes. We describe some common code smells that we have found inhibit the retrofitting of unit tests, and suggest tactical refactorings to make such code more accessible. Most of the smells we have identified are concerned with the difficulty of isolating the code we wish to test from the rest of the system, a key requirement for effective unit testing. Our experience is that changing code to make it testable usually improves its quality, with a clearer and more flexible structure. When we retrofit unit tests, we can also try to retrofit the design benefits that come with test-first programming.

Our experience is mainly based on Java, but we believe that most of these patterns apply to other object-oriented languages. We assume that the reader is familiar with test-first development, the JUnit framework [4], and refactoring as described by Fowler [5]; we annotate patterns and refactorings from Fowler using *[F]*.

[1] http://c2.com/cgi/wiki?UnitTestingLegacyCode

2 CODE SMELLS

This section describes some common code smells that make unit tests difficult to add to legacy software.

Singleton

The Singleton is perhaps the most widely used and misunderstood pattern in Gamma *et al* [6], and is often found in legacy code. A common use of Singleton is to encapsulate external resources such as databases or files. Since it provides a single access point, calls to a singleton are often scattered throughout the code.

The issues for unit testing are: first, sometimes the singleton object cannot be changed because, for example, it is set up in a static initializer (see below). This makes it impossible to isolate the tested code from its environment by substituting a mock implementation [7] of the singleton. Second, even where the singletons can be replaced, the tests for objects that refer to many singletons will be tedious and error-prone to set up. Finally, many uses of a singleton will repeat behaviour that must be tested separately for each case, increasing the testing effort.

One solution is to add a setter method to the singleton class to overwrite its static instance. This weakens the encapsulation of the singleton itself but may be suitable for cross-application features such as a logging interface. The test suite can use the setter to assign a mock implementation and the application can continue to use the singleton as before. Rainsberger [8] suggests aggregating singletons in a *Toolbox* so that their lifetimes can be managed by the application. An alternative approach that does not alter the singleton class is to *Pass singletons through.*

Complex construction

Sometimes most of the implementation of a class is concerned with setting up its initial state and is not used again after instantiation. For example, a class to represent a financial yield curve requires complex calculations to work out its initial values, but only simple lookups when in use. Similarly, a class that represents a user may refer to an external directory service only during initialization.

The issues for unit testing are: first, it is cumbersome to create instances of the class when testing both the class itself and classes that interact with it; for example, it may be too hard to create every state that needs testing via the public constructors. Second, construction that relies on external resources is an unnecessary dependency when managing unit test suites. Third, the test suite for class instances will be less readable because it will be swamped with tests for construction rather than tests for use. These are all symptoms of a poor separation of concerns.

A first step would be to add a simple constructor to the class and to write separate test suites for construction and use. A better approach is to refactor using *Separate construction from use*.

Data class

Data class, which consists mainly of fields and their getters and setters, is described in Fowler. Data classes are often found with utility classes to support common operations on them.

The issue for unit testing is that data classes often imply that some related behaviour has been scattered around the clients of the class, so related test code has to be repeated or gathered into helper code. Furthermore, code that passes data objects around tends to have *Long methods* (see below) that are hard to test.

Even where data classes are required, perhaps for use in a reflective framework, it is often possible to move responsibility to the data class by combination of extraction, encapsulation and moving, as described in Fowler.

Static initialization

Many developers use static initialization, code that is run when a class is loaded, to set its initial state; common examples are initializing singletons, starting loggers, and loading property values from files. Whilst this technique is useful for reducing the intellectual load on the programmer and for ensuring the internal consistency of a component before it is used, there are maintenance costs if the static code is complex or refers to external resources.

The issues for unit testing are: first, it can be difficult to run repeated tests over such code. To do so requires repeated reloading of the class, it may be hard to set up conditions to test failures, and errors may be hard to trap for test results. Second, such classes are hard to instantiate outside their framework when they are required for testing other classes, especially when the source code is not available. For example, one of the authors got stuck trying to create a parameter object from an application server because a static initialization in a super type was failing silently.

The solution is to *Remove complex static initializers.*

Bleeding across layers

It is quite common to see business domain code use framework classes, such as Servlets, so that package dependencies “bleed” across the layers of an application. Examples include passing a Servlet request as a parameter to a domain class, or throwing a Servlet exception from within a domain class. This risk may be higher on Extreme Programming projects, where the programmers aspire to “Do The Simplest Thing That Could Possibly Work.”

The issue for unit testing is that bleeding across layers introduces unnecessary dependencies between components and, hence, between tests. First, anyone reading or writing a test for the business class must understand both layers and the tests are less likely to read well. Second, if classes from the framework layer change, this may require business layer tests to be changed. Finally, test setup may be

difficult if, for example, some framework classes do not have constructors that are accessible outside the framework.

The solution is to refactor at the places where the layers touch and *Weaken dependencies between layers.*

Classes as parameters
In Java, it is worth specifying the parameter and return types of a method (its signature) in terms of interfaces rather than classes, if those classes are at all complex.

The issue for unit testing is that, for parameters that are defined as classes, a mock or stub implementation can only be substituted by subclassing, which has two limitations. First, it cannot also inherit from common mock or stub implementation classes, nor can it take advantage of Java proxies, as with EasyMock [9]. Second, if the parameter class, or one of its ancestors, changes or adds a method, the stub class will no longer override all the real methods and the test case might pick up the wrong implementation. Such bugs in the test environment can be difficult to find when the test unexpectedly fails (or, worse, passes). Similar issues arise with return types; when the class itself is stubbed out for testing other classes in the code base, it may be easier to return a simple stub than an instance of the real type.

The solution is to *Replace class with interface* in the signature. If this solution is too difficult to apply at first, perhaps because the parameter class is used in many places, then first create the stub implementation as a subclass of the parameter class and later refactor both the stub and original classes to extract an interface.

Imprecise exceptions
Java supports *checked exceptions*, where the compiler will validate that all the exceptions that might be thrown from within a method are either handled or declared as part of the signature. Some developers avoid checked exceptions by catching and dropping exceptions they don't know how to handle (that is, by ignoring the signal), or by declaring the method to throw the generic type Exception. An equivalent to the latter is to always throw unchecked exceptions.

The issue for unit testing is that exception handling must also be tested. First, it may be hard to detect a result that will confirm that an exception has been thrown if the target code drops it. For example, if the beginning of a method drops an exception, its unit tests ought to be run twice, once with the exception thrown and once without. Second, where exception checking is ignored, it can take some time to work out and unit test all the possible exception paths through the code.

The solution is to be precise when managing checked exceptions. Dropped exceptions should be encapsulated by *Extract Method [F],* which will often suggest a further *Extract Class [F]* to reify the interaction with the component that throws the exception. Checked exception lists should be narrowed to just those exceptions that a method can throw, this can be propagated incrementally from where the code touches external libraries. Our experience is that a little rigour applied to indistinct Java exception management can greatly simplify the code and, hence, the unit tests to drive it.

Long method
Long method is described in Fowler. The additional issue for retrofitting unit tests is that such methods are also painful to test. Typically this involves writing a long series of tests, each of which progresses a little further through the method before forcing the next exit condition. Setting up enough state in a test to get through the entire method is, at best, complicated.

If the method is too long to test as it stands, one solution is to test and refactor incrementally. Long methods often contain several logical sections, for example: check the inputs, perform operations, and assemble the result. Test a section at a time and extract helper methods to isolate it. If possible, extract a section and its tests as a class, perhaps as a policy object. Subsequently, the new object can be replaced with a Mock Object and the tests for the method simplified.

In the best case, a long method collapses either to a class in its own right, or to a collaboration between a set of smaller objects, that can be tested separately. The tests for the refactored method need only exercise the routing between those objects.

3 REFACTORINGS

Pass singletons through
Objects that are neither ubiquitous, such as loggers, nor constant values should be passed through as method parameters, rather than retrieved as singletons; a common example is a database connection. This can be done incrementally by first adding the parameter to low-level methods (in this case DBConnection) and passing in the instance from the singleton, then later propagating the new parameter up the call stack. There is a risk that parameter lists will become too long as more singletons are removed, but in practice we have found that ex-singletons, such as external connections, are usually local to a sub-system or package. Furthermore, passing singletons through as parameters often leads to *Introducing Parameter Objects [F]* which, in turn, suggest useful refactorings.

The advantage for unit testing is that a parameter, particularly if it is an interface, is easier than a singleton to replace with a mock implementation, thus isolating the test from the rest of the application.

Separate construction from use
Where most of the implementation of a class is taken up with constructing an instance, such as calculating the yield

curve on a financial instrument, consider separating the construction aspects into a factory object—our mental image for this is the way that booster sections are jettisoned during the launch of a space rocket.

This technique is most likely to apply when the construction phase uses different resources or libraries from the use of the object. The benefit for unit testing is that the two classes should have more focused responsibilities and so be easier both to test and to stub out.

Remove complex static initializers
A first step is to move static initialization code into static methods so it can be referred to by name and parameters and results passed through. Techniques such as lazy initialization allow such methods to be called explicitly, for testing, or automatically when in production.

It may be, however, that code of any complexity should not be run implicitly, but should be made visible and called directly from the application startup sequence. This makes error handling easier to manage and ensures that failures occur at the right time. One of the authors used this technique when porting a component between two frameworks that used different error reporting. The move revealed a failure in initializing the logging library that had previously been hidden by an incorrect startup sequence.

Weaken dependencies between layers
To reduce class dependencies between layers of an application, there are three cases to consider: First, where explicit creation occurs across the boundary, such as creating a new Customer object from a servlet, consider *Replace Constructor with Factory Method [F]*. Thus the servlet might now use a CustomerFactory to create a Customer, rather than instantiating one directly. When unit testing we can substitute a mock CustomerFactory that instantiates a mock Customer.

Second, where several values are passed across a boundary, consider *Introduce Parameter Object [F]*. For example, when passing start and end dates from a user's http request to an Account object, we might bundle these into a DateRange type. This clarifies the relationship between the layers and we are likely to be able to move behaviour to the new parameter object, which can then be tested in isolation.

Third, where a framework layer needs to interrogate its client layer, it should define a callback interface that the client layer can implement. For example, where an Account object needs to extract session values from an http request, define an AccountSession interface that makes explicit what an Account needs to know about its context, then implement an HttpAccountSession class for use with servlets. We can now unit test separately the extraction of the values from the http session and the use of those values in the Account. For the Account class, we can create a MockAccountSession to isolate its tests from the servlet framework.

Replace class with interface
In Java, where the input parameters or return value of a method are typed as classes that are at all complex, consider changing those types to interfaces and renaming the classes. Types based on interfaces are easier to substitute with stub or mock implementations, so it becomes easier to test a class in isolation from the rest of the system. The overhead of maintaining the extra type is mitigated by modern development environments and by the flexibility it adds to the code. One implication of this technique is that the coding standard should not use type names to distinguish interfaces or classes, such as with a leading or trailing 'I', as this hinders refactoring between the two.

With some care, the same technique can be applied in C++ by using abstract classes as interfaces and multiple inheritance to bind them to implementation classes.

4 RELATED WORK AND OTHER TECHNIQUES

There is a growing body of experience with test-first development: Fowler [5] catalogues the core code smells and refactorings, and there are links to papers and discussions from the JUnit site [4] and on the C2 wiki [10]. This paper focuses on code smells and refactorings related to retrofitting unit tests.

There have been some interesting discussions about the use of Aspect Oriented Programming [11] for unit testing. The idea is to intercept the calls the target code makes to other objects in the application. One idea is to use this technique to implement Mock Objects, tracking calls and returning preloaded results [12]. An alternative is to log important values when running functional-level tests and check that these don't change during refactoring. In our view, these are valuable intermediate techniques to help with opening up opaque code, but we are wary that they change the actual code under test.

5 CONCLUSIONS

In this paper, we have identified some coding practices that make the retrofitting of unit tests difficult. We have identified some related refactorings that we have found allow us to "chip away" at the code enough to start adding unit tests. These tests then give us the confidence to refactor, add new functionality, or fix bugs using test-first programming.

Those of us who practice test-first programming do so because we believe that it is more effective and drives us to writing better code. Many of us, however, also have to work with existing code that we cannot break, but need to change. The authors have found that retrofitting unit tests helps to support programmers when making changes and to guide the code to a better design through refactoring.

How much time to spend on retrofitting unit tests, or whether to do so at all, is outside the scope of this paper; it can be an expensive exercise. For those who chose to do so,

we hope that this paper embodies some useful experience. Before starting to refactor for testing, we also recommend that the developers write some functional tests that touch the components concerned to catch any gross errors that they might introduce.

Finally, the real point of this paper is that, given the will and enough slack in the immediate schedule, it *is* possible to add unit tests to almost any existing code base—and for a team that wants to be agile, it is essential.

ACKNOWLEDGEMENTS

Thanks to Michael Feathers, Tim Mackinnon, Duncan McGregor, and Rachel Davies for their comments on early versions, and to the members of the Extreme Tuesday Club for being part of the community.

REFERENCES

1. Beck, K, *Extreme programming explained: embrace change*. Addison-Wesley, 1999.
2. http://c2.com/cgi/wiki?UnitInUnitTestIsntTheUnitYouAreThinkingOf
3. van Deursen, A., Moonen L, can den Bergh, A, Kok G, *Refactoring Test Code*, XP2001, Sardinia, 2001.
4. The JUnit web site. http://www.junit.org
5. Fowler M., *Refactoring: improving the design of existing code*, Addison-Wesley, 1999.
6. Gamma E, Helm, R, Johnson, R, Vlissides, J. *Design Patterns*, Addison-Wesley, 1995.
7. Mackinnon T., Freeman S., Craig P., *Endotesting: unit testing with Mock Objects*, in *Extreme Programming Examined*, Addison-Wesley, 2000.
8. Rainsberger, J. *Use your singletons wisely*, http://www-106.ibm.com/developerworks/components/library/co-single.html
9. EasyMock http://www.easymock.org/
10. http://c2.com/cgi/wiki?UnitTestingLegacyCode
11. http://www.aspectj.org
12. http://groups.yahoo.com/group/extremeprogramming/message/37004

Implementing and Using Resumable TestFailures in Smalltalk

Joseph Pelrine
MetaProg GmbH

Position paper for
Workshop on Testing in XP (WTiXP 2002)
XP 2002, Alghero, Sardinia

The high performance aspect of extreme Programming derives in part from the rapid feedback cycles in unit testing. Collection testing and validation, however, can be very time-intensive, and can slow down the development process to the point where the advantages of test-driven programming are lost. Through the implementation of "resumable" test failures, though, this deficit can be compensated for. The ResumableTestFailure (to be introduced in SUnit 3.1) offers a flexible implementation of this in Smalltalk.

The new SUnit release 3.1 adds more functionality at little cost to both Smalltalk's and extreme Programming's premier testing framework. In addition to the assert:description: family of methods (well-known from JUnit), which allow you to attach arbitrary description strings to assertions, the major change is the introduction of a resumable TestFailure.

Why would you need a resumable TestFailure? Take a look at this example from a typical test case method:

```
aCollection do: [ :each | self assert: each isFoo]
```

In this case, as soon as the first element of the collection isn't Foo, the test stops and returns a failure. Although this information is necessary for test-driven development, it normally isn't sufficient. In most cases, we would like to continue, and see both how many elements and which elements aren't Foo. It would also be nice to log this information. You can do this in this way:

```
aCollection do: [ :each |
      self
            assert: each isFoo
            description: each printString, 'is not Foo'
            resumable: true]
```

This will print out a message on the Transcript for each element that fails. It doesn't cumulate failures, i.e., if the assertion fail 10 times in your test method, you'll still only see one failure.

Implementation

As a result of SUnit being extremely lightweight, it required only minimal effort to implement the functionality required to support ResumableTestFailures.

1. The class ResumableTestFailure was created as a subclass of TestFailure, which itself is defined in the SUnitPreload package. (This package contains all dialect-specific Classes and Methods for SUnit, and makes it possible for the core SUnit package to be dialect-independent).
2. The method Exception>>#isResumable was overwritten to return true.
3. The method Exception>>#sunitExitWith: , which normally returns from the exception, was overwritten to resume execution.

While running the test cases, it was noticed that the SUnit framework had a conceptual inconsistency which was overlooked in the original implementation. The method TestResult>>#failures, which returns the collection of failures for a test run, was implemented to be an OrderedCollection. This led to each triggering of a ResumableTestFailure adding yet another failure to the collection. The implementation was changed to be a Set, based on the fact that a test case method is a failure regardless of how many assertions in the method are false. Also, implementing the failure collection as a Set reflects the fact that test cases should be non-deterministic, i.e., the order in which the test cases are executed is irrelevant.

The change in TestResult>>#failures led to a slight change in TestResult>>#defects, which was dependent on the failures being contained in an OrderedCollection. This change was minor, and will not be discussed further.

The implementation also required a method for triggering both regular and resumable TestFailures. The basic method, TestCase>>#assert:description:resumable: is illustrated below:

```
assert: aBoolean description: aString resumable:
resumableBoolean

      | exception |
      aBoolean ifFalse: [
            self logFailure: aString.
            exception := resumableBoolean
                  ifTrue: [ResumableTestFailure]
                  ifFalse: [TestResult failure].
            exception sunitSignalWith: aString]
```

Once again, the implementation of SUnit has proven to be very efficient and flexible when it comes to adding or extending behavior without changing the base packages. Of course, being in Smalltalk helps too – YMMV.

Joseph Pelrine wrote the reference implementation of SUnit 3.0. He is (together with Sames Shuster and Jeff Odell) maintainer of the SUnit distribution on Sourceforge.

He can be reached at:

Joseph Pelrine
MetaProg GmbH
Bachlettenstrasse 41
CH-4054 Basel
Switzerland
Email: jpelrine@metaprog.com