Third International Conference on Software Testing, Verification, and Validation Workshops

LiFT: Driving Development using a Business-Readable DSL for Web Testing

Robert Chatley
Develogical Ltd
95 Muswell Hill Broadway
London, N10 3RS
Email: robert@develogical.com

John Ayres
MetaBroadcast Ltd
33 Fitzroy Street
London, W1T 6DU
Email: john@metabroadcast.com

Tom White
Cloudera Inc
1409 Chapin Avenue
Burlingame, CA 94010
Email: tom@cloudera.com

***Abstract*—This paper describes the development and evolution of LiFT, a framework for writing automated tests in a style that makes them very readable, even for non-programmers. We call this style 'literate testing'. By creating a domain-specific language embedded within Java, we were able to write automated tests that read almost like natural language, allowing business requirements to be expressed very clearly. This allows development to be driven from tests that are created by developers and customers together, helping give all stakeholders confidence that the right things are being tested and hence a correct system being built. We discuss the experiences of a team using these tools and techniques in a large commercial project, and the lessons learned from the experience.**

I. Introduction

Testing aims to tell us when a piece of software has been built right - but how do we know that we are testing the right thing? The growing agile [4] and test-driven development [1] movements promote developing suites of tests to document a system's behaviour rather than producing large specification documents. Tests can provide a form of executable specification, but how can we validate that these automated tests accurately encode stakeholder requirements?

To ensure that requirements are accurately transferred from customers to testers and developers, we need to be able to communicate easily among developers, and between developers, business analysts and customers, to agree on requirements and reach a common understanding. When we have good communication in place, developers know that they are writing the right tests, and each stakeholder knows that the developers are writing the right tests. How can we achieve this level of communication?

Formalising a specification removes ambiguity from requirements by defining them precisely enough that no degree of interpretation is needed by any interested party. However, as Johnson highlights [14], formal specifications are widely found to be difficult to read, therefore communication within and between teams is inhibited. The same is true with automated tests, it is difficult to present them in a form where they can be read by all stakeholders.

In a survey of over four hundred software engineers, Austin and Parkin found that the most common reason for not using formal techniques in producing specifications (reported in 23 percent of cases), was that clients could not read the resulting specifications [19]. This highlights the need for tools that allow specifications or test suites to be written in a way that is precise enough as to be executable, yet readable enough to be accessible to customers and other non-developers.

The main artifact produced by development teams, particularly those following an agile [4] methodology, is code, and it is code that often forms the focal point of their discussions. Code can also often be difficult to read and understand, even for experienced programmers. It is generally accepted that writing clean, readable code is good practice, and we think that this is even more important in test code, but it is difficult to do. Even "good code" is often expressed using technical features in such a way that it is incomprehensible to non-programmers. Test code is often entrenched with details of the way that the system it is testing is implemented. We want to produce techniques for writing tests that produce code that is readable, even by non-programmers. In the sense popularised by Donald Knuth [16] we are aiming for our tests to be *literate*.

We have striven to develop a framework that allows automated tests to be written in a style that focuses on readability, to the point where they are (give or take a few dots and brackets imposed by our programming language of choice), written in plain English. This means that we can easily write automated tests that clearly express and test business requirements. Customers and other stakeholders can read over the test code to check that they agree with what the developers think the requirements are. Customers and developers can work together to produce sets of acceptance tests collaboratively. This significantly reduces the risk associated with translating ideas and requirements from the problem domain into code. By providing a medium that is comprehensible and open to discussion by both customers and developers, we reduce problems introduced by developers and test writers interpreting requirements incorrectly.

This paper presents the design and evolution of techniques for writing literate tests in Java, focusing on testing web applications. Over time we have developed a framework for literate testing, which we call LiFT, that allows test code to describe the intended function of the system under test, without being tied to the way that the system is implemented. Moreover, over time we have developed ideas, tools and techniques that allow a business-readable domain specific language [8] for

978-0-7695-4050-4/10 $26.00 © 2010 IEEE
DOI 10.1109/ICSTW.2010.12

writing automated acceptance tests to be embedded in Java. The framework is extensible, and it is easy for the test writer to develop custom extensions. Such extensions can enrich the test language, bringing it closer to the problem domain, and we have found that this is of great importance in creating tests that are meaningful to both development and business stakeholders.

At the time of the initial development of the LiFT framework, the authors were all working for the software development company Kizoom in London, although since then all three have moved to new companies. LiFT was used to develop acceptance tests for a large scale commercial application developed by Kizoom. We discuss the experiences of the team in adopting the framework, customising it to their needs, and the benefits and difficulties that they experienced.

II. Background

A. Literate Programming

Donald Knuth popularised a concept that he called Literate Programming [16]. Literate programming is an approach to programming that emphasises that programs should be written to be read by people as well as compilers. Knuth's idea was to combine source code and documentation, making programs more readable, easier to understand and therefore easier to maintain. He wrote: "Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do".

Johnson presents the idea of *literate specification* [14] to try to make complex formal specifications more comprehensible to new team members and customers. Formal and informal descriptions are combined to present different views on the specification. Mathematical formal descriptions are given in Z, while accompanying design rationale is given in English.

Rather than combining code and documentation in one document which can be processed by tools to produce running code, we have taken the view that the code itself should be readable by programmers or non-programmers. By adopting a particular style of coding, it should be possible to produce code expressions that read as natural language sentences, but are still executable. Designing a language or API that allows code to be written that has this quality is difficult. Fowler [9] notes that "any fool can write code that a computer can understand. Good programmers write code that humans can understand". One of the main challenges addressed in this work has been producing an API that is flexible and easy to use, while allowing terms to be used in a syntax that reads naturally.

B. Expressiveness in Unit Testing

Unit testing is now a common practice in the software development industry. Unit tests provide verification that code behaves as expected, but also provide documentation for code. The developer of an API can develop in parallel a set of unit tests that exercise that API, and provide examples of its intended use. One could write detailed comments along with the unit tests to provide documentation, but this increases the number of lines of code to be written and read, and introduces the possibility of inconsistency between comments and the code they are supposed to document. It is all too easy to forget to update comments when making a change to a piece of code. It is safer to rely on the tests for a description of the system's behaviour, but for this to be effective, test writers must place great importance on writing good, clean, readable code.

In Java, unit tests are commonly written using the JUnit framework [2]. JUnit provides a set of assertion methods that can be used in tests. For example `assertTrue()`, `assertFalse()` and `assertEquals()`. These are highly functional, but can lack expressiveness.

The idea of using flexible assertions in JUnit using the `assertThat()` construct was introduced by Walnes [22]. Using this form, where with plain JUnit one would write:

```
assertEquals(``Hello'', myString);
```

using `assertThat` one can write:

```
assertThat(myString, isEqualTo(``Hello''));
```

The method `isEqualTo()` returns a *constraint* which is evaluated for the subject of the assertion when the `assertThat()` method executes. Using different constraints in this way means that all assertions can be written as stylised sentences of the form *assert that subject satisfies constraint*. These read as English from left to right, which plain JUnit assertions do not. The Hamcrest library [23] (originally extracted from the jMock project [10] and recently integrated into the JUnit distribution) makes use of this concept of constraints, which it calls *matchers*, and defines an interface that means that the test writer can easily extend the vocabulary of available constraints to meet their needs.

C. A Fluent Style

In designing our tools and APIs, we focussed heavily on how test code would read. To build more extended phrases, we often adopted a programming style that Fowler calls a *fluent interface* [7], which allows multiple calls to an object to be chained together, rather than writing each as a separate statement. This is achieved by having each method return the receiver. A simple example might be the following creation of an object to represent a restaurant meal.

```
Meal order = new Breakfast()
    .withEggs().withBacon().withCoffee();
```

Chaining clauses in this way allows code to be formed in a more readable, more literate, style.

D. Acceptance Testing

Unit testing provides assurance about the correctness of individual units inside a software system, but systems are typically built by combining many hundreds of such units. In addition to testing units in isolation, often from a computational point of view, we also need to test the system as whole, from a user point of view. Here we test at a high level that the system as a whole fulfils the requirements it was designed to meet. This type of testing is often called functional or acceptance testing.

Again, in recent times there has been movement in the software engineering community to automate this type of testing as much as possible.

The philosophy of test-driven development [1] applies equally to acceptance testing as it does to unit testing. Acceptance tests are codifications of user requirements in executable tests. They often reflect scenarios as described by the customer. TDD advocates creating test cases before writing code. Developers start by creating a failing test, and then add and refactor code until the test passes. If they start with a good test, at this point the developers know they can tick off the implementation of the feature. User acceptance tests are different from lower level unit tests, as they express what the user can do with the software, not what the software does; they express intention rather than implementation detail. But how do the developers know that they have written the right acceptance tests? How do the customers know that the developers have written the right tests? To get assurance, we need to enable customers, analysts and developers to discuss their test cases in a common language, so that tests can be developed collaboratively, validated and agreed.

There are a number of testing tools available, for example HttpUnit [11], HtmlUnit [3] and Watir [20], which provide the programmer with the facility to create automated acceptance tests for web applications, but the resulting code is often verbose, tied strongly to the implementation technology of the application, and not particularly readable. To enable the tests to form the basis for discussion and communication between different stakeholders, we need something that allows the user's expectations of the function of the system to be expressed more clearly.

III. Literate Testing

> "A good programmer in these times does not just write programs. A good programmer builds a working vocabulary. In other words, a good programmer does language design, though not from scratch, but by building on the frame of a base language." Guy Steele [12].

In developing the literate style of test that we now advocate, we experimented with and evolved various different styles of test, aiming ultimately to produce something that was easy to read, easy to write, composable and extensible. This section discusses the motivation behind each style's inception and its relative success.

A. Vanilla HtmlUnit

The first tests were written by directly interacting with the the HtmlUnit [3] API. Figure 1 shows an example of this style of test. The test does the following:

- Loads www.example.com and (by the cast to HtmlPage) asserts that it is an HTML page.
- Asserts that the title of the page is "Example.com".
- Asserts that the page contains "Welcome to example.com".

Even in a simple test such as this, it should be noted that there is an overlap between two concepts: what we are testing and the details of the technology we are using to test it. Notice how the method by which the web page is retrieved is exposed, in this case via the HtmlUnit WebClient object. To a test implementer or reader the method by which the page was retrieved is of no interest [1].

Another disadvantage of this approach is highlighted in the final line of the test. The author of this line intended to assert that the page contains a certain string of text. But the test itself reads "assert it is true that calling the `asText` method of the page returns a string that contains 'Welcome to example.com'". This is a procedural (step by step) approach rather than a declarative (this is what I want to test) approach. Test cases in 'real world' enterprise situations are rarely this simple. Figure 2 shows a more complicated example. Again, let us examine the author's intention when writing the test:

- In the table named 'matrix' select the radio button in the third row.
- In the form named 'customer_form' enter 'john' into the field named 'username'.
- Check that after clicking submit the next page has the title 'Next page'.

A clear indicator of the unwieldiness of this style is the disparity between the size of the author's intentions (three medium length sentences) and the length of the realised test (around nine lines of code). The test also utilises a rather cryptic XPath expression `//table[@name='matrix']...` to extract the relevant 'radio button' node from the HTML document. Again we see the steps required to perform the test are intertwined with the intention of the test. The test contains various programming constructs such as assignment, casting and encapsulated methods, along with the previously mentioned XPath expression. It is unreadable for anyone without programming knowledge.

B. Extending HtmlUnit

Our next species of functional test made use of a number of static Java methods that were coded to automate common tasks. Two such methods created in the early phases of development were `assertPageHasTitle()` and `assertPageHasText()` which are shown in Figure 3. These allow the assertions from Figure 1 to be reduced to `assertPageHasTitle("Example.com")` and

[1]Rare exclusions to this rule may include performance or compatibility testing across multiple browsers

```
WebClient webClient = new WebClient();
HtmlPage page = ((HtmlPage) webClient.getPage(new URL("http://www.example.com")));
assertEquals("Example.com", page.getTitleText());
assertTrue(page.asText().contains("Welcome to example.com"));
```

Fig. 1. A simple HtmlUnit test

```
WebClient webClient = new WebClient();
HtmlPage page = ((HtmlPage) webClient.getPage(new URL("http://www.example.com'')));

HtmlRadioButton radioButton =
  (HtmlRadioButton) page.getElement("//table[@name='matrix']/tr[2]/td/input[@type='radio']");

radioButton.click();

HtmlForm form = page.getFormByName("customer_form");
HtmlTextInput username = (HtmlTextInput) form.getInputByName("username");
username.setValue("john");

page = form.submit();

assertEquals("Next page", page.getTitleText())
```

Fig. 2. A more complicated HtmlUnit test

```
static void assertPageHasTitle(HtmlPage page, String expectedTitle) {
  assertEquals(expected, page.getTitleText());
}

static void assertPageHasText(HtmlPage page, String expectedText) {
  assertTrue(page.asText().contains(expectedText))
}
```

Fig. 3. Utility methods for checking a page

`assertPageHasText("Welcome to example.com")`. This approach not only shortens the test but makes it much easier to read. Note also how the specifics of how we ensured there was a certain string of text on the page are not visible. In effect we are simply applying a "composed method" refactoring [15] to raise the level of abstraction at which our test method is written. This moves us closer to two of our goals, namely ease of reading and writing.

However, our goal of composition is not served well by this approach. As pages become more complicated and tests more thorough, it becomes clear that it is infeasible to create specific methods for the vast array of possible assertions. The utility methods in Figure 3 can only make one logical assertion. For a concrete example let us imagine Example.com has a rival called 'Specimen.com' and wish to assert that their rival company's name does not appear on any of their pages. We cannot use Java's negation operator on the `assertPageHasText` method since it is a complete assertion rather than a building block for a compound assertion. Thus we require another method: `assertPageDoesNotHaveText`. If tests as complex as Figure 2 are required then the number of utility methods required quickly becomes unmaintainable.

C. A More General Approach

We began work on a more general literate testing framework to try and overcome the problems that we had experienced with previous approaches. This approach utilises Walnes' constraint based assertion style [22], and builds on top of it an extensible framework that allows constructs for matching particular artifacts to be combined with actions that describe interactions with the system. We have devised a number of mechanisms that wrap the basic functionality of HtmlUnit to interact with a web application, but give a much richer set of expressive constructs that make for much more readable tests. Figure 4 shows a test written using LiFT that exercises the Google search engine. The test demonstrates the use of a number of different features of LiFT. The example shows immediately the increased readability achieved by using the LiFT API, and the way that the different features combine. The following sections describe the different sorts of features used in the test.

1) Actions: Actions are methods that are used to specify an interaction with the system, for example clicking on a hyperlink to navigate to a different page, entering a value into a

text field, or clicking a button to submit a form. These methods update the state of the system under test. Examples of actions are:

```
goToTheIndexPage();
clickOn(button().labelled("go"));
enter("example", into(textField());
```

N.B. The method `into` simply returns its argument. It is used only as linguistic device and provides no extra functionality, but makes the line read much more naturally.

2) Implicit Variables: All of our literate test cases inherit from a common base class which carries some state throughout the duration of the test. Most importantly it has a field `page` which always represents the current page. When a navigation action is performed, such as `clickOn(link("Home"))` this field is updated to represent the change of state. This enables us to make assertions about the different pages in the navigational flow without having to make explicit assignments in the test code.

3) Finders: Finders are at the centre our framework, they provide a mechanism for locating elements of interest on a page for use with an assertion or action. A finder can be applied to a section of a page to extract elements of a particular type, for example applying a `TableFinder` to a page will return a list containing all the HTML table elements on that page. Rather than creating new finder objects inline in the test, we provide a factory class `Finders` whose methods can be statically imported. We can then write the more literate `table()` instead of `new TableFinder()`. This is common idiom that we employ in constructing tests.

Finders adhere to a well defined interface, which makes it easy for test writers to add to the collection of available finders. We have found that this is one of areas in which test writers most often extend the framework. Finders provide the nouns in the vocabulary available to write tests. To tailor the vocabulary to their domain needs, test writers often add new nouns.

4) Refinements: A finder will match all occurrences of a particular artifact in the current page, for example all the `links()` or all the `images()` . Often we want to check for the presence, or absence, of a particular link or image. To do this we use the concept of a refinement. Refinements are applied to finders by chaining clauses in a fluent style. For example:

```
link().withText("Home").withUrl("/home")
image().withUrlThat(endsWith("logo.jpg"))
```

In the first line we see how multiple clauses can be chained until the refinement is specific enough. In the second, rather than testing if the image's URL is equal to a specific string, we want to be less strict. Here we use a refinement that takes a further constraint as a parameter. We want to match a URL that `endsWith()` logo.jpg. `endsWith()` returns a constraint against which image URLs are evaluated. Here we see how different building blocks can be composed to increase the expressiveness and power of the framework. Both lines read naturally if the punctuation is ignored.

5) Constraints: The `assertThat()` construct takes as parameters a constraint and an object to apply it to. We have added a number of new constraints in developing the literate test framework, the most commonly used being `has()` and `doesNotHave()`. These constraints take a finder as a parameter, and on evaluation apply the finder to the page. They then evaluate to true or false dependent on the number of matches that they make. For `has()` to return true, there must be at least one match. For `doesNotHave()` there must be no matches. We also have variants that allow an additional constraint to be passed controlling the number of matches that must be made. Assertions follow the form:

```
assertThat(page, has(title("Welcome")));
assertThat(page, doesNotHave(text("Error")));
assertThat(page, has(2, images()));
assertThat(page, has(atLeast(2), images()));
```

IV. Tool Support

In developing LiFT we wanted to harness as much as possible existing tools commonly being used by developers. This was one of the reasons for embedding the language in Java rather than create a custom domain specific language. A custom (external) DSL would require custom tools to be built to process it. It would then be hard for users to extend the DSL to more fully describe their domain, which we have identified as an important feature of literate testing. The supporting tools would have to be extended and customised for each extension to the testing language. Developers also tend to work with a minimal set of tools, and do not want to have to learn a new tool for each task, or switch between tools for testing and implementing.

For these reasons we used JUnit as our test harness, as most Java developers doing some form of automated testing will already be using it. We also benefit from the static typing of Java, and the code completion facilities available in modern IDEs like Eclipse or IntelliJ IDEA. At each point in writing a line of a test, the IDE can offer suggestions as to what can come next. Figure 5 shows this in Eclipse. As language extensions are implemented just by creating new Java classes, these are automatically picked up and supported by Java IDEs with no further work on the part of the developer.

V. Case Study

The framework described was used in constructing a set of automated acceptance tests for a large web-based application built by Kizoom to supply public transport information in the UK. This website habitually services over one million journey enquiries per day. The development team working on the application consisted of nine developers, a project manager and a business analyst/customer representative. In total around

```
public class GoogleTest extends NavigatingLiftTestCase {

  public void testGoogleImageSearch() throws Exception {
    goTo("http://www.google.com/");
    assertThat(page, has(title("Google")));
    clickOn(link("Images"));
    enter("kizoom", into(textField()));
    clickOn(button("Search Images"));
    assertThat(page, has(text("Kizoom summer party")));
    assertThat(page, has(image().withUrlThat(endsWith("summer04.jpg"))));
  }
}
```

Fig. 4. A Literate Functional Test

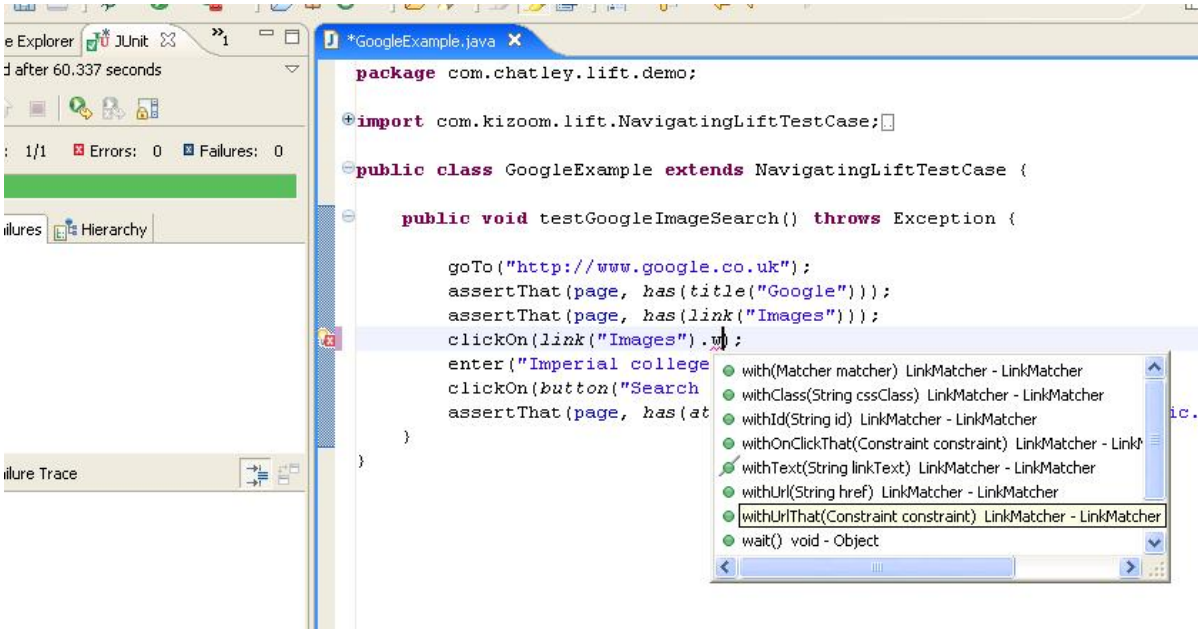

Fig. 5. Code completion writing a LiFT test in Eclipse.

200 acceptance test cases were developed in the literate style to cover the business requirements for the website as described by the customer.

Initially, getting the developers to adopt the new style of test writing, and to work with the new framework was difficult. While the more readable style appealed immediately to some, others found the new style of programming difficult and unnatural. They found it difficult to see the benefit of writing tests in the new style, and found adjusting to a new way of thinking difficult. Writing literate tests meant they needed to type fewer characters, but had to spend longer thinking about how to express what they wanted to test. Initially, for several team members, it would have been faster for them to write tests in the traditional, procedural style. However, once they had gained some experience with the literate testing framework, and had some examples to work from, their understanding and productivity increased and they began to accept the new technique.

It was notable that in general the programmers who found it more difficult to adapt to the new style were those who had only ever programmed, at least professionally, in Java. The declarative style, and notably the absence of explicit assignment statements in the literate style code, was unnatural for them at first and it took them some time to become happy with writing tests using the framework. Programmers who had experience of a wider range of programming languages and paradigms, particularly functional programming, seemed more immediately happy with the new style.

The static typing of Java, combined with the support of Eclipse, meant that developers could get automatic code completion while constructing their tests. This meant, for example, that it was easy for them to see the available refinements at each point in a matching expression. However, as we used static imports for the matcher methods, rather than creating a deep class hierarchy which might limit interoperation with other testing frameworks, it was difficult to get auto-

completion or suggestions of these, as at the time of writing, Eclipse's support for working with statically imported methods is somewhat weak.

The flexibility that we had aimed for in developing the framework, producing composable building blocks, meant that there were often different ways that it would be possible to formulate a test. This sometimes led to confusion as to the 'right' way to express something. Some developers at times felt that they might benefit from a more constrained API with fewer possible options. However, this might have limited the ability to express things in a way that read naturally.

There was also some discussion amongst the team regarding the degree to which we were subverting the 'normal' Java style, and whether new developers coming to the team might find it more difficult to get up to speed. Also, were we in danger of having inconsistent styles in the codebase depending on whether the code was for a test, or for the implementation of the system - where the literate style did not spread so readily, as there was a wide variety of different APIs to interact with, and it was not possible to base all code on one framework as was possible with tests.

Literate testing was introduced part way through the project. This meant that at one point in development, a proportion of the tests in the suite were written in the new style, but the remainder were written using HtmlUnit directly. It was agreed that it would be better to have a consistent style across all of the tests, but converting the HtmlUnit tests was hard work. This was mostly because initially, not all of the finders, refinements and constraints needed to express the tests were available. The framework needed to be grown to increase the expressiveness, tailoring the available vocabulary to the domain. As the framework became richer, the need to add to it diminished with each test written, increasing the speed with which tests could be written, and allowing developers to be users of the API rather than developers of it.

The decision to move to the literate style was aided by developers' experiences of debugging failing tests, especially those written by other developers. With a literate test it proved very quick to determine the intention of the test, and to reproduce the steps manually using a browser to observe the application's behaviour. Although this was still possible with HtmlUnit tests, it took more detective work, and a more detailed understanding of the test code to reveal the same information. Developers reported that they typically spent only one quarter of the time to debug a failure from a LiFT test compared with plain HtmlUnit tests.

Throughout development, we found that developers were able to discuss queries over requirements with the business analyst and project manager at the level of their tests. They were able to review tests together and alter them to test accurately the customer's requirements. When the customer required a report of the tests that had been performed during development, this was assembled by taking the code of the acceptance tests, and simply editing it to remove excess punctuation and add spaces into identifiers composed of several words. This was much easier than writing out test scenarios by hand, trying to describe the actions performed by the automated test, which again would have been a point where errors could have been introduced. The process of translating the literate tests to English suitable for pasting in to a document was fairly mechanical, and if it needed to be done again, it was suggested that a pretty printing tool could be written to automate this task. This technique was also used to produce scripts to be followed by manual testers.

VI. The Next Generation

The LiFT framework that we developed was built on top of HtmlUnit to interact with the web. At the same time that we were improving tools for expressing tests at a higher level of abstraction, advances were being made in the tools at the lower level. The increase in complexity of web applications, and the browsers that support them, particularly with the increase in the amount of JavaScript employed in a new generation of rich web applications, meant that the browser was an intrinsic part of the test infrastructure. HtmlUnit emulates a browser, but this means that its interpretation of a web page may differ from that of popular browsers such as Firefox, Internet Explorer or Safari, especially for applications that make use of the asynchronous loading of data (typically referred to as AJAX).

Simon Stewart developed the WebDriver tool [21] to solve these problems, by allowing tests to run against a real instance of the browser, but keeping a common abstraction layer above it, so that the same test could be run across several browsers to check compatibility. WebDriver's API is simple, but quite low-level. We wanted to integrate LiFT to build on top of this, so we could take advantage of WebDriver's browser automation, together with expressive readable test code. We took the opportunity to rework LiFT, taking into account some of the lessons that we had learned from experience building tests with the first version.

One of the main things that we found when using the original LiFT on a large project was the need to continually add to the library of constructs available, to test different aspects of a given webpage (for example different types of HTML elements) and to give these names that mapped into the users' application domain. During the development of the WebDriver version, we focussed on making this extension as easy as possible. Previously it had been necessary to change the code of LiFT itself in order to integrate new features smoothly. In the new version we made a separation between the core layer, which should remain largely constant, and a layer of user defined language that could be easily plugged in. This was a key learning. In order to provide an effective TDD tool, we needed to enforce enough structure and direction in the API that it was clear how it was supposed to be used, and "what to do next" when using it. However, we could never anticipate all of the users' needs in terms of mapping to their domain and the business language of their application. Therefore, we must provide suitable extension points for small modules to be plugged in creating seamless extensions to the DSL. A colleague Romilly Cocking noted how easy this was in the new version [5], where he created a heading finder to

locate heading elements within a webpage with just a few simple lines of code.

Finding the balance between providing function and allowing extension was difficult. It was only by studying how developers used the tools on a large project that we came to realise where the extension points needed to be. An iterative approach must be taken to the development of tools as well as the development of applications.

VII. Related Work

There are other tools and techniques that aim to allow customers, testers and and programmers to collaborate on acceptance tests. Ward Cunningham developed FIT [6], a Framework for Integrated Tests. FIT uses the concept of tables to organise customers' expectations for a system and to report the results of running tests against the system. Tests can be written by creating tables using tools like Microsoft Excel, or in a web page using wiki markup. Testers create tables giving examples of combinations of input and output data. These can then be tied to the application using *fixtures*, code to connect the table to the system under test. The tests can then be run and a report generated in the form of an HTML page, with parts of the table coloured red or green depending on whether the expectation was met. FIT is good for creating tests for applications that perform calculations on data. It is easy to write a table giving examples of inputs and expected outputs for a component that performs a mathematical function. It is more difficult to specify less structured test activities like browsing around the pages of a web application. We believe that the flexibility of expression provided by literate testing gives it an advantage over FIT in these circumstances. Another difficulty of FIT is the need to develop a fixture. The tables themselves are not independently executable. This is an extra layer which could contain bugs, and leaves some of the interpretation of the tables' meaning in the hands of the developer. With literate testing, the tests are written in code, and so are directly executable within the JUnit framework.

Watir [20] (pronounced 'water'), is a framework for writing acceptance tests for web applications in Ruby. Its tests are fairly readable when compared with something like HtmlUnit, but do not focus on readability as a key concern. Watir tests revolve around performing operations on an object that represents a browser as demonstrated in the following example adapted from those in the Watir distribution:

```
test_site = 'http://www.google.com'
#open the IE browser
$ie = IE.new
$ie.goto(test_site)
$ie.text_field(:name, "q").set("literate")
```

Brian Marick has had success writing "sentence style tests" [17], also in Ruby. These use a technique creating objects to model different pages to abstract from the implementation of the application, and allow tests to be written in a way that makes them readable to customers. The tests take the following form:

```
page.main_text.
  should_have_no_list_named(:visits).
  should_have_no_list_named(:audits).
  and_should_have_a_help_popup
```

Marick's tests are perhaps more readable because a lot of the functions are written explicitly to test particular cases, so the logic or quantification is combined with the noun to be matched, in the same statement. Also, they benefit from the fact that they are written in Ruby rather than Java, so the syntax is somewhat less cumbersome. Our framework tries to construct more general building blocks, which means we tend to have rather more nesting of clauses, but our tests can be written without having to create new assertion methods for each test. Of his experience working with customers using this style of test, Marick writes, "I last week demonstrated my sentence style tests ... to an audience consisting of a bank CIO, his reports-in-charge-of-software, and some of their reports. It went really well - they appeared to understand the tests, and very few of them were actual software people." [18].

Cucumber [13] provides a very readable way of writing plain English specifications, but allows these to be executed by mapping particular handlers to particular keywords in the text. This mapping, and some associated Ruby code, provides a fixture attaching the test to the system. All the other words in the test specification are ignored by the test runner and are for documentation only. This technique does lead to a particularly readable specification, but it is effectively an external DSL, as specifications are just plain text. This means that there is no tool support for writing Cucumber tests, only for running them, and therefore writing them really requires a developer with knowledge of how the test tool and fixture is implemented. It must be known which words are special keywords and which are not, which statements will trigger the system under test and which will not.

VIII. Conclusions and Future Work

We have presented the development and evolution of a framework for driving development through executable tests written in a 'literate' style which makes them readable by programmers and non-programmers alike. We have shown how a Java API was developed to support programming in such a literate style, and the effect that this has on the way that tests are written.

Writing tests in a literate style means that it is easy to see what the intention of a test is, i.e. what it is supposed to be testing. This makes it easy to match tests again customer requirements as it is possible for the customer to validate the tests that have been written to check that they are testing the behaviour that they expect, before the implementation is started. This can be done by simply reading over the test cases and discussing them with the developer. The literate style also makes debugging easier.

We have shown the value experienced by a commercial team using the literate testing framework to develop a suite of acceptance tests for a large scale web application. Converting from a more procedural style to the more declarative literate style was difficult for some developers, but overall the benefits gained from the increased readability and ease of expressing requirements outweighed these difficulties. We found that developers, business analysts and customers were all able to discuss the tests and understand what they meant, that this aided communication amongst team members, and meant that subsequently more of the features were implemented correctly.

We developed the framework in Java as its initial target audience was Java developers, and it was intended to interoperate with other Java testing tools such as JUnit. It has been suggested that languages such as Ruby or SmallTalk might be particularly amenable to writing code in a literate style, as they require less in the way of punctuation in the code. Notably in Ruby the parentheses around method parameters are optional. Future work may investigate the possibility of translating the framework to another language.

Another area of future research would be to try and enable other domains to be tested. Up to now we have focussed on the domain of web applications, but it may be possible to re-engineer the tool set to allow different drivers to be plugged in, and the syntax extended to fit different domains. Then we could maybe support testing desktop GUI applications, XML web services, and other data formats such as flat files or spreadsheets. Initial work suggests that the same form of assertions and constraints work across domains, while the set of finders and refinements required varies. As these are the parts of the framework that are designed to be extensible, it seems that it should be possible to move to different domains with some work.

We released the original LiFT framework as an open source project hosted at https://lift.dev.java.net. However, work on the original version pretty much stopped after the integration with WebDriver was started. That work is also open source and can be found at http://code.google.com/p/webdriver/wiki/LiftStyleApi

IX. Acknowledgements

The authors would like to thank the development team at Kizoom for the time spent developing, using and commenting on the literate testing framework. We would also like to thank Nat Pryce, Steve Freeman, Joe Walnes, Brian Marick, Neil Dunn and staff at Imperial College London for their ideas and feedback during the development of the framework.

References

[1] K. Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, November 2002.

[2] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7):37–50, 1998.

[3] M. Bowler. HtmlUnit. Technical report, http://htmlunit.sourceforge.net/, 2002-2006.

[4] A. Cockburn and J. Highsmith. Agile software development: The people factor. *Computer*, 34(11):131–133, 2001.

[5] R. Cocking. Testing RIAs with WebDriver. Technical report, http://www.cocking.co.uk/blog/2008/03/testing-rias-with-webdriver.html, 2008.

[6] W. Cunningham. Fit: Framework for Integrated Tests. http://fit.c2.com/, 2005.

[7] M. Fowler. Fluent Interface. http://www.martinfowler.com/bliki/FluentInterface.html, 2005.

[8] M. Fowler. Business Readable DSL. http://www.martinfowler.com/bliki/BusinessReadableDSL.html, 2008.

[9] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, June 1999.

[10] S. Freeman, T. Mackinnon, N. Pryce, and J. Walnes. jMock: supporting responsibility-based design with mock objects. In *OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 4–5, New York, NY, USA, 2004. ACM Press.

[11] R. Gold. HttpUnit. Technical report, http://httpunit.sourceforge.net/, 2003-2006.

[12] J. Guy L. Steele. Growing a language. In *OOPSLA '98 Addendum: Addendum to the 1998 proceedings of the conference on Object-oriented programming, systems, languages, and applications (Addendum)*, New York, NY, USA, 1998. ACM Press.

[13] A. Hellesoy. Cucumber - Making BDD fun. Technical report, http://cukes.info/, 2008-2010.

[14] C. W. Johnson. Literate specifications. *Software Engineering Journal*, 11(4):225–237, July 1996.

[15] J. Kerievsky. *Refactoring to Patterns*. Addison Wesley, 2004.

[16] D. E. Knuth. Literate programming. Technical report STAN-CS-83-981, 1983.

[17] B. Marick. Sentence Style for Tests. http://www.testing.com/cgi-bin/blog/2006/06/09#mock-style, 2006.

[18] B. Marick. Sentence style for tests. private communication, June 2006.

[19] G. I. Parkin and S. Austin. Overview: Survey of formal methods in industry. In *FORTE '93: Proceedings of the IFIP TC6/WG6.1 Sixth International Conference on Formal Description Techniques, VI*, pages 189–203. North-Holland, 1994.

[20] P. Rogers, B. Pettichord, and J. Kohl. Watir: Web Application Testing in Ruby. Technical report, http://wtr.rubyforge.org/, 2005.

[21] S. Stewart. WebDriver. http://code.google.com/p/webdriver/, 2007-2010.

[22] J. Walnes. Flexible JUnit Assertions with assertThat(). http://joe.truemesh.com/blog/000511.html, 2005.

[23] J. Walnes. Hamcrest. Technical report, http://code.google.com/p/hamcrest/, 2006-2010.