Future Internet **2011**, *3*, 159-174; doi:10.3390/fi3020159

OPEN ACCESS

future internet

ISSN 1999-5903

www.mdpi.com/journal/futureinternet

Article

Evolving Web-Based Test Automation into Agile Business Specifications

Rick Mugridge [1,⋆], **Mark Utting** [1] **and David Streader** [2]

[1] Rimu Research, 271 Ararimu Valley Road, RD 2 Waimauku, Auckland 0882, New Zealand; E-Mail: marku@cs.waikato.ac.nz

[2] Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton 3240, New Zealand; E-Mail: davidistreader@gmail.com

⋆ Author to whom correspondence should be addressed; E-Mail: rick@rimuresearch.com; Tel.: +64-9-411-5498.

Received: 31 March 2011; in revised form: 26 May 2011 / Accepted: 27 May 2011 / Published: 3 June 2011

Abstract: Usually, test automation scripts for a web application directly mirror the actions that the tester carries out in the browser, but they tend to be verbose and repetitive, making them expensive to maintain and ineffective in an agile setting. Our research has focussed on providing tool-support for business-level, example-based specifications that are mapped to the browser level for automatic verification. We provide refactoring support for the evolution of existing browser-level tests into business-level specifications. As resulting business rule tables may be incomplete, redundant or contradictory, our tool provides feedback on coverage.

Keywords: web test automation; agile; business specifications; abstraction; coverage

---

1. Introduction

Not surprisingly, a common approach to test automation of a web application is based on the way that manual testing is carried out. Test scripts consist of the sequence of actions and checks that a tester would carry out in the browser to verify the system. Such scripts may be hand-written or they may be recorded with record-and-playback tools [1].

Unfortunately, there are several serious problems with test scripts that are directly expressed in terms of the browser, as well as a significant lost opportunity. Such scripts tend to be low-level, verbose, and repetitive. They are not effective in an agile setting, as they tend to be written after development, once the page details are finished. They also tend to be slow to execute.

Given the inherent repetition, due to common subsequences of steps, such tests prove to be expensive to create and maintain. Sometimes, small changes to the system can have an impact on many tests. High test maintenance costs has lead some teams to abandon test automation as a bad idea. Other teams try to avoid these problems by rewriting tests at a higher level. But, with large test suites, making such changes can be expensive and error-prone.

An alternative approach is to start with the goal of creating business-level specifications based on meaningful examples. These specifications are mapped into actions in the browser (or other implementation level) for automatic verification.

In an agile setting, such specifications evolve iteratively, along with the code. In this way, the specifications serve two goals: understanding and verification. They are used to incrementally specify business rules, constraints, and processes, which help with a wider understanding and communication of the business level. They aid in discussing changes in an agile/lean development process before implementation. They verify that changes have been made and that existing functionality has not been broken. They make it much easier to understand business rules, and to see whether there are important missing cases. As they can be verified as consistent with the system, they can be trusted as documentation. As they are not repetitive, and not directly dependent on implementation details, they are much easier to maintain [2].

Our research has focussed on *ZiBreve*, an open source system written in Scala, to provide tool-support for creating business-level, example-based specifications. This supports the creation, evolution and organisation of such specifications so that they can be effective across the whole software development process.

In this paper, we mainly focus on how *ZiBreve* solves the problem of legacy test automation, with two main approaches. First, we provide support for the evolution of existing implementation-level tests into business-level specifications based on examples. This is akin to Extract Method refactoring with program code [3]. The tool rapidly searches for repetition across a test suite and proposes abstractions, with parameters, based on the compression that can be obtained. The user selects and alters these, to be applied automatically to the suite. These serve to define a multi-level mapping between the business level of business rules *etc*. and the implementation level of the browser. Step by step, a knowledgeable person can extract the business level from a large suite. At the same time the tests are compressed into concise and focused specifications based on meaningful examples.

Once business rule specifications have been extracted, *ZiBreve* can provide feedback on coverage in business rule tables, pointing out any redundancy and contradictions and suggesting possible additions for cases that may not have been considered.

The next section details the problems of traditional approaches to test automation. We then look at how business specifications, with mappings, can provide a superior solution, with an important impact on collaboration and on communicating business goals within the wider software team. The third section shows how a knowledgeable person or pair can, step by step, extract the business level from low-level

test scripts. The following section briefly covers coverage feedback. Finally, we discuss related work and conclude with future work.

2. Traditional Test Automation

Test automation is usually written (or recorded) in terms of the implementation level of a system under test, such as the browser for web-based applications. This approach, at first, seems obvious, because it is necessary to verify that the tests and the system are consistent, and it seems obvious to test in terms of the user interface because that is the users' view of the system.

Hence each test is a trace through the system, which makes changes and verifies results through the browser, or through another user interface, a database, a web services, and *etc*. The test steps are in terms of Xpaths, GUI locators, database tables and columns, XML, *etc*.

2.1. Example of a Test Script

Figure 1 shows a segment of a test for a web application, using *SpiderFixture*. This is a part of *FitLibrary* [4], an open source testing tool that can be used with *FitNesse* [5].

Figure 1. The middle segment of a longer test.

FitNesse is a wiki-based system for managing tests. A test is represented as a wiki page containing tables and other text, which can be edited using wiki markup. Suites of tests are based on a hierarchy of wiki pages. Wiki markup is used to define tables, include mechanisms, and configuration information about test execution. *FitNesse* manages test or suite execution by running a separate test framework, passing test information to it, and presenting reports back to the user. Three test frameworks supported are *Fit*, *FitLibrary* and *Slim*, all of which are based on tables [5–7].

FitLibrary is an interpreter for table-based domain-specific languages (DSLs [8]), specialised for test automation like its predecessor *Fit* [6,7]. Tables conveniently provide a simple syntactic structure. Test feedback is added directly to the tables, so it is easier to see what is going on than if a separate report

were provided. The contents of tables are dynamically mapped by *FitLibrary* into method calls into the underlying code, such as *SpiderFixture*. The action keywords (cells shown in grey in Figure 1) are mapped into the name of a method, and the data cells (shown in white) are the parameters.

SpiderFixture includes methods for operating on web pages, available to test writers as table actions. For example the action in the first table in Figure 1 corresponds to the *SpiderFixture* method *click()* with one parameter (an element locator), while the fourth table corresponds to the method *withAddText()*, with two arguments (an element locator for a text field and the text to add, a credit card number). Elements on the page, such as buttons, selects, and text fields, are located in various ways, including through IDs and Xpaths (such as "//input[@id='creditCardCardNo']").

SpiderFixture is implemented with *Selenium 2* [9], which supports test automation through a range of browsers and *HtmlUnit* [10], a "headless" test system. It is convenient to see the changes happening in the browser as a test executes.

2.2. Problems with Traditional Test Automation

Writing (or recording) tests at this low-level works fine when there are a small number of tests and a relatively stable interface into the system under test, but it breaks down as the number of tests grow. It becomes harder to organise the tests, and it becomes harder to tell whether the interesting test cases have been covered. It can be difficult to read low-level tests with excessive "hieroglyphics"; such tests are analogous to assembler code.

This approach especially breaks down when the system is evolving, either due to an agile process or due to "maintenance". There are likely to be changes to UI layout (or database tables, XML structure, *etc*.) that break the tests. Copy, paste and alter is a common technique to use in writing new tests, but this amplifies the duplication and leads to maintenance problems. Xpaths and other ways of locating elements on the page are particularly vulnerable. With more tests, and more redundancy, more places have to be changed to keep up with system changes. For example, if additional information needs to be gathered about a credit card, it is necessary to find all the tests where this occurs. In essence, the implementation-specific nature of the tests makes them low-level, verbose, and repetitive.

More importantly, the tests remain at a very technical level. They are as poor as code for understanding the business rules, constraints and processes that are the intention of the system. It is necessary to infer the larger-scale from the finer details, and this can only be done by someone who is knowledgeable in both the business level and the implementation level. So other documentation is needed for the business-level view, such as documents of requirements, for communicating with business stakeholders. Unfortunately, such documents can quickly get out of date and serve only as a historical record of the original goals, decisions and intentions of the system.

The final problem with this approach occurs in an agile/lean setting. Tests for new functionality can only be completed once the implementation is done, because the details of such tests depend on detailed design and implementation decisions. At a process level, this means that the focus remains on picking up problems after the fact, rather than helping with understanding what is required at a business level earlier in the process. From a lean perspective, preparing tests afterwards is wasteful [11,12].

3. Business Specifications Based on Examples

The solution to the above problems with traditional test automation is to instead write specifications in terms of examples at the business level. Rather than being a technical activity, specifying by example is a way to involve the larger team in clarifying, understanding and documenting aspects that are important at the business level [2,13]. This corresponds to the idea of *essential use cases* [14], although specifications are example-based and have a different emphasis.

This carries the idea of Test Driven Development to the whole-system level [15]. eXtreme Programming introduced this as the practice of "acceptance tests" [16], although this has often been neglected by agile teams.

The "ubiquitous" language used needs to be meaningful to business stakeholders [17]. It needs to evolve, as the goals and details of the "requirements" grow and as they are better understood. The specifications focus on business rules, constraints, processes and objects, using meaningful, concise, and well-organised examples. In this way, they serve to gain and maintain a common understanding of what is needed among the wider team.

Separately, a mapping is defined between the business level specifications and the implementation level, so that the specifications also serve as automated tests. As we will see soon, those mappings may be organised into layers.

3.1. Specifying by Example

Figure 2 shows, within *ZiBreve*, an example-based specification at the business level, where the language is concerned with business rules, rather than how they might be implemented. This includes an (abridged) specification of the validity, or otherwise, of credit card details (before the card is validated by external systems).

Figure 2. A business-level specification.

Valid cards

Valid Credit Cards		
Credit Card No	*Card Type*	*Expires*
5169961933224494	Mastercard	04/12
4485891284549100	Visa	01/12
378238305872855	American Express	12/11
30328219479275	Diners Club	11/11

Incorrect card types

Invalid Credit Cards		
Credit Card No	*Card Type*	*Expires*
5420218419129712	American Express	12/12
4532236754255544	Visar	12/11

Expired

Invalid Credit Cards		
Credit Card No	*Card Type*	*Expires*
4024007154824048	Visa	02/10

Invalid card numbers

Invalid Credit Cards		
Credit Card No	*Card Type*	*Expires*
5169961983244414	Mastercard	02/13
xxxxxxxxxxxxxxx	Visa	02/12

Each table consists of two header rows, in grey, which serve to identify what is being specified and the various fields of interest. Subsequent rows of the table, in white, are independent examples of valid or invalid credit card information.

A business stakeholder who is familiar with credit card validation can determine whether the examples are correct and complete, and can recommend changes (e.g., "We need to include the CVV2 or CVC2 number from the back of the card"). For this business rule, they do not need to know when and how the credit card information is gathered and verified in the web interface, how error messages are presented to the user, or what happens next. Some of these are business process issues that are best considered in other specifications.

3.2. Mapping between Layers

Independently, a mapping is defined between the business-level language and implementation-level, so that the consistency of the specifications and of the current system can be determined through test execution.

In general, there may be several such layers, and with mappings to several distinct implementation levels. This is illustrated in Table 1, where the business level is at the top. This is mapped through several layers to a *fixturing* layer and then into three implementation levels: Web browser, web services, and indirect access into the application code.

The mappings between the intermediate layers in Table 1 are managed with *defined actions* in *FitLibrary*. These are table-based procedural abstractions, with parameters, as discussed in the next section. As we will see, these play an important role in providing diagnostic information when specifications fail during test execution.

Fixture code provides a mapping into application or driver code. For example, *SpiderFixture* is an *adaptor* for the API of Selenium 2 that handles the language of the table-based actions, as well as hiding away details, and managing state and the asynchronous nature of testing against AJAX.

Table 1. Multiple levels mapping to multiple implementations.

<table>
<tr><th colspan="3">Business Level Specifications</th><td>high level</td></tr>
<tr><td>scenario level</td><td>service transaction level</td><td>rule table level</td><td></td></tr>
<tr><td>page level</td><td rowspan="2">service data level</td><td rowspan="3">application fixtures</td><td></td></tr>
<tr><td>field level</td><td></td></tr>
<tr><td>spider fixture</td><td>web services client fixture</td><td></td></tr>
<tr><td>Selenium 2
(Firefox, Chrome, etc.)</td><td>HttpClient</td><td>Application API</td><td>low level</td></tr>
</table>

3.3. Using ZiBreve

ZiBreve supports the development of such example-based specifications. It uses the same underlying wiki pages as with *FitNesse* so that *ZiBreve* can be used in conjunction with *FitNesse*. However, *ZiBreve*

provides improved editing, undo/redo, structuring and other capabilities that are common in development environments for programming languages.

In addition, *ZiBreve* supports the user in dealing with legacy test automation, so that the problems of such tests can be avoided, making the benefits of example-based specifications available for future evolution of the system.

4. Extracting the Business Level

Business level specifications are extracted from implementation-level tests through multiple cycles of abstraction. It is often easier to do this bottom-up, by starting with simple subsequences that can be clearly isolated and named. As the tests are compressed, it is easier to see the patterns and to extract them up the layers towards a business level. As we will see, this often shows shortcomings in the specifications that may need to be addressed.

4.1. Locating Abstraction Sites

In order to begin to extract business-level specifications, possible extraction sites within the implementation-levels tests need to be found. The user can simply scan through the tests, looking for parts that stand out as suitable targets for abstraction.

Or the user can request that *ZiBreve* search across all the tests in a suite, or a sub-suite, looking for repetition. This results in a list of abstractions with parameters, ordered by the compression that they achieve. The compression is based on the text in the potential call sites and the abstraction, the number of call sites, and the number of parameters. As only some of the resulting abstractions will be meaningful, this may simply serve to guide the user.

4.2. Focussing on Abstractions

Either way, the user can then drive the process more directly by selecting a sequence of tables and requesting that *ZiBreve* search for abstractions that involve that sequence. This gives immediate feedback on how frequently the sequence is used, as well as possible parameterisations.

Figure 3 shows a sequence of tables that has been selected by the user as a focus for abstraction. Figure 4 shows the best abstraction that *ZiBreve* finds for these selected tables across the tests in the overall suite. The top part includes the potential abstraction. The first table is the generated header, with four parameters. Subsequent tables in the top part are the body, based on the selected sequence, using the parameters. For example, the formal parameter `p0` is referenced in the body as `@{p0}`.

The bottom part of Figure 4 shows the potential calls into that abstraction. Each row shows the name of the test, following by the four arguments that correspond to the four parameters in the abstraction. This allows the user to see how useful that abstraction will be.

Figure 3. User selects some tables as a focus for abstracting.

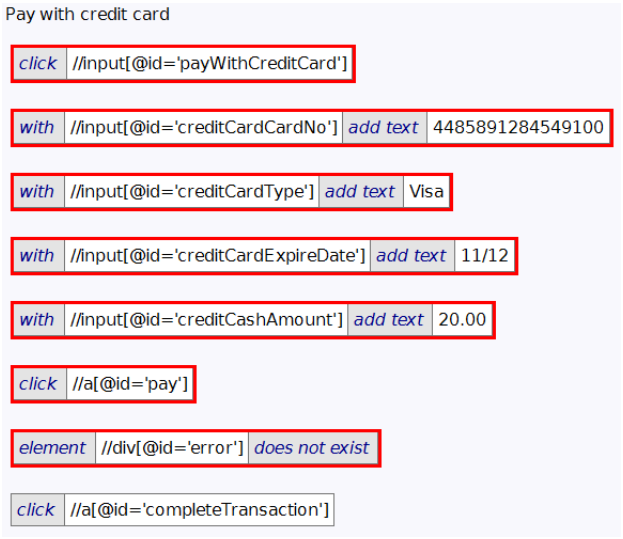
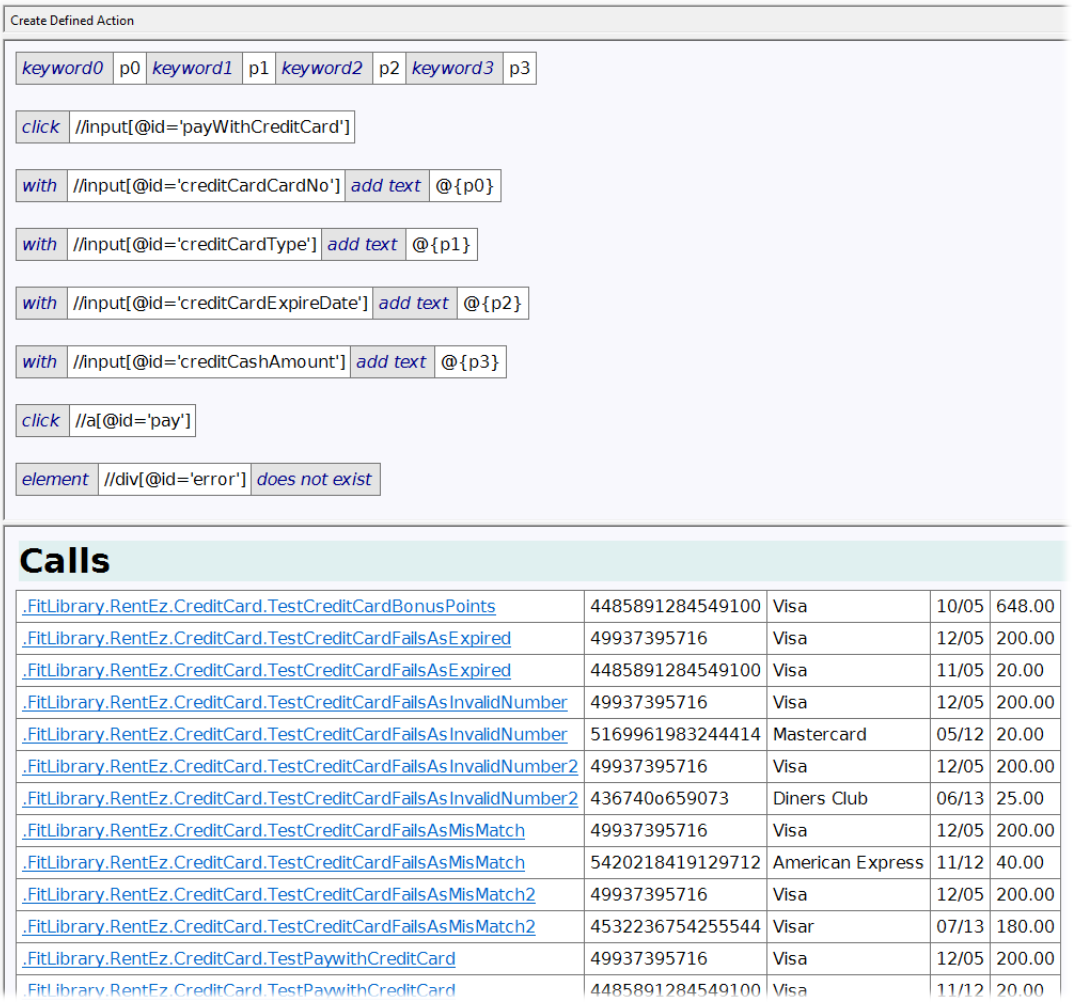


Figure 4. A possible abstraction.



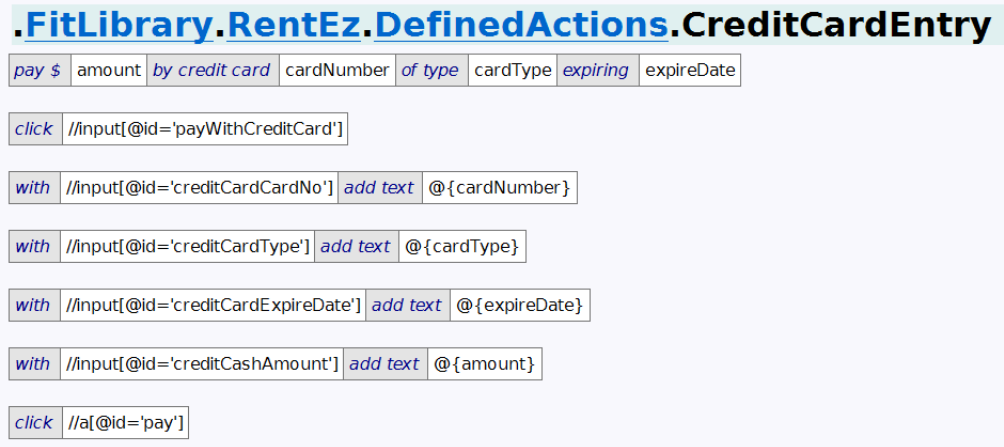
4.3. Creating a Defined Action

If the user likes the abstraction, they can create a *defined action* from it by clicking on the *Create Defined Action* button shown at the top of Figure 4.

Figure 5 shows the *defined action* that results, after the user has renamed the keywords and parameters in the header, and decided to delete the last table. As with an abstraction, the first table, the header table, names the defined action keywords and its parameters. Subsequent tables make up the body. In this case, the user has reordered the parameters to make the header more readable.

When a *defined action* is called, the actual parameters are bound to the formal parameters and the body is executed. If the body succeeds, the call as a whole is coloured green to show success. However, if a table in the body fails, the result (with parameter bindings) is included in the report at the point of call. This provides useful diagnostic information during test execution, when it is useful to see what happens down through the layers to the implementation level.

Figure 5. The resulting defined action.



4.4. Applying a Defined Action

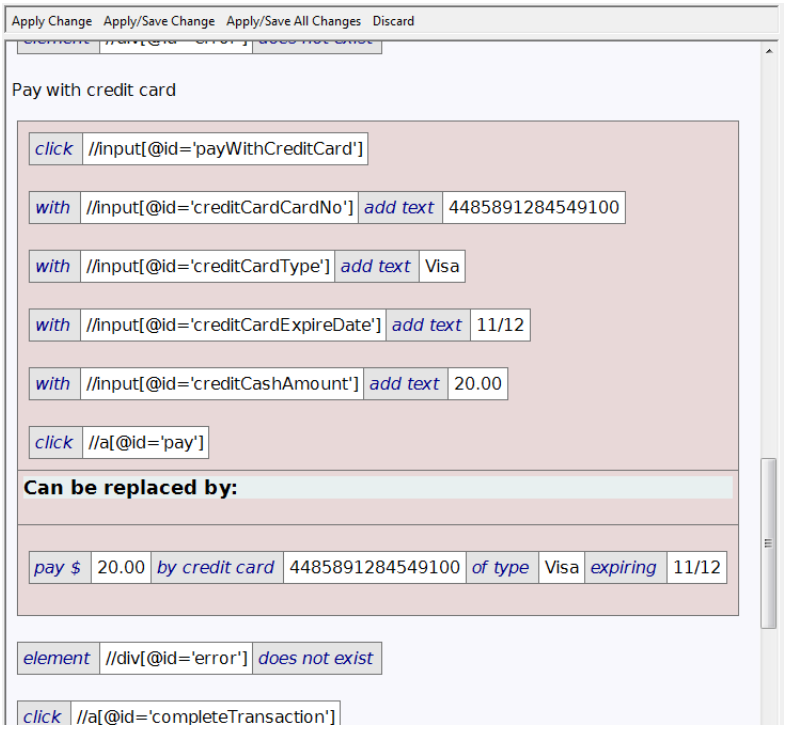
In the final step of the cycle, the user can then apply the new *defined action*. On request, *ZiBreve* searches for all possible call sites for the given *defined action* (or across all *defined actions*).

Figure 6 shows the details of one potential call site. Within the larger test, the tables to be replaced, and the resultant action, are shown in the coloured table. The user can click on the *Apply Change* button (at the top of Figure 6) to have it applied, or click on the *Apply/Save All Changes* button to have all remaining potential calls applied.

Finding potential calls is carried out as a separate step from abstracting for several reasons:

- The user may alter the *defined action* after it has been generated from the abstraction, as we saw in Figure 5.
- The user may choose to write a *defined action* independently.
- An existing *defined action* may apply in new tests.

Figure 6. A potential call.



4.5. Multiple Cycles

As refactorings are carried out in this way, and *defined actions* are introduced for the various layers, the tests shrink. This makes it easier to see the intent behind each test and to see new abstractions that reduce repetition further.

Much of the detail in a test is irrelevant to its main point, both in terms of the data and in terms of the process. For example, where the actual customer's name and address has no impact, a *defined action* can be introduced to define a standard customer. This is used wherever a customer is needed, but where any will do. Personas can also be created to represent different types of users or business objects.

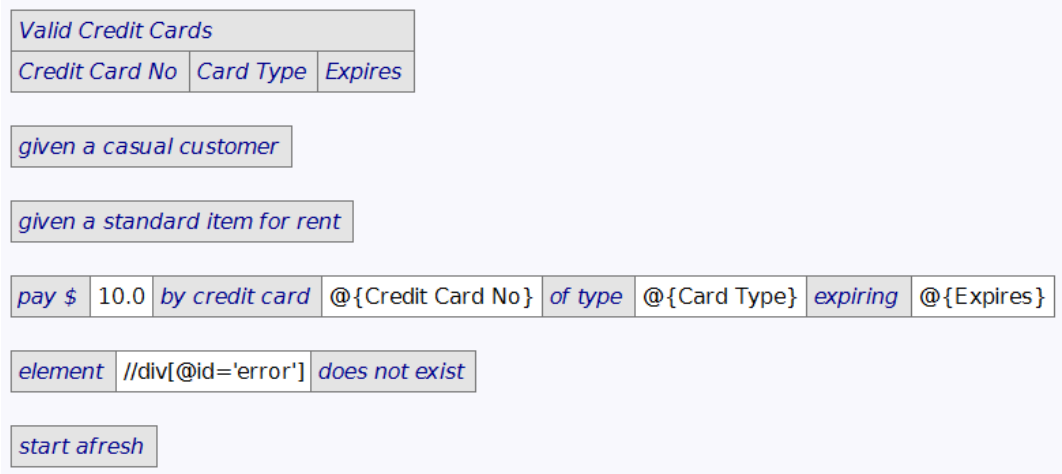
In other tests, parts of the process will be irrelevant, and those parts can also be abstracted away with *defined actions*. For example, when verifying a credit card, all that's needed is that a customer is at the point of paying. This is illustrated by the first two tables in the test shown in Figure 7.

Where multiple tests have been used to verify a business rule, these can eventually be collapsed into a single test. For example, Figure 7 includes a repeating sequence of steps to verify credit card validation that has resulted from this process. After abstracting the *multi-defined action* in Figure 8 and applying it, and doing this again for invalid credit cards, the test that results is the one shown earlier in Figure 2.

A *multi-defined action* is a variation on a *defined action* that allows for repeating groups, such as in a business rule. As can be seen in the *multi-defined action* shown in Figure 8, the header table corresponds to the first two rows of the first table in Figure 2. The second row of the header table in Figure 8 specifies the parameters, which are used in the body using the same notation as *defined actions*. For each data row in a call to a *multi-defined action*, its body is executed after the parameters have been bound from that row.

Figure 7. A repeating test.

Figure 8. A multi-defined action.



ZiBreve is designed to provide very fast feedback with abstractions, so that a quick cycle of improvements can be made. For example, when searching for any repetition across a suite containing 2600 tests, it provides useful abstraction information within 10 s.

4.6. Repair

As the cycles of refactorings continue, certain problems are likely to appear:

- Some of the original tests are testing several different aspects, which need to be untangled and split into separate specifications. This corresponds to the *code smell* of a class or method with too many responsibilities [3].
- Important cases have not been covered, now that it is possible to see all of them in one table. Adding a row in a table is much easier than copying and altering a long test sequence. In Section 5 we describe how the tool can suggest missing cases.
- As the specifications are reviewed by the business stakeholders, they realise that important distinctions have been missed.

Two types of untangling may be required. Business rules may be mixed with business process flow. For example, a test may verify that a credit card is invalid, and that a suitable error message is provided, and that subsequent work flow is correct. Separating these concerns leads to clearer and more modular specifications. A business rule table may be the join of several dependent rules; pulling these apart can aid focus, clarity and simplify the specifications.

4.7. Speed and Implementation-Independence

Fast feedback from test automation during continuous integration is most important in an agile process. Unfortunately, testing through the web interface is rather slow. The impact of this can be reduced through parallel execution of tests against multiple copies of a web application, or through careful scheduling of parallel tests against a single instance. Another approach is to reduce the use of the web interface itself.

Once implementation-level tests are refactored to business-level, example-based specifications, it is easier to handle such speed problems. Alternative mappings can be provided so that some aspects of test execution run through other interfaces. For example, rather than entering setup data through the browser, it is quicker to enter this data directly in the database. Rather than testing business rules through the browser, some of these can be tested directly through the application API or web services. In general, the same business specifications can be run through different mappings, with earlier tests using faster mappings to provide quick feedback and later ones verifying that they also work through the browser.

This approach also means that changes to the underlying technology can be straightforward to handle. For example, with the addition of mappings for a REST-based interface, existing specifications can also be run to test through that mechanism.

In the extreme case, a system may be completely re-implemented. With business-level specifications to support this process, important functionality need not be lost nor broken. New mappings can be developed to the new platform as it is developed.

4.8. Implementation and Initial Results

The abstraction process is based on work on finding clones in code [18]. Subsequences of tables from the tests in the suite are hashed based on the "shape" of each table (table and row lengths plus the contents of the first cell of the first and second row, where it exists). This collects together sets of related tables that are then processed further to find abstractions. The sets are split into subsets, again using hashing, based on the cell contents and the introduction of parameters. The resulting abstractions are ordered by the compression they achieve, with the best ones shown to the user.

In initial trials, useful abstractions from a suite of 200 tests were found in 20 s. With a larger suite of 2600 tests, the algorithm did not stop in a reasonable time. We realised that responsiveness was more important than completeness, allowing the user to quickly focus on points of high repetition, remove them through abstraction, and then repeat the process.

We have now revised the algorithm so that abstractions with potentially-higher call rates are processed first. As abstractions are created, the best call rate is used to cut off work on creating new abstractions with a lower call rate as early as possible. We use a simulated annealing approach, so that an exponentially greater proportion of the best call rate is used over time, reaching 100% after 10 s of processing already-loaded tests. This allows for some variability, but limits the time that is used. This has led to excellent responsiveness and abstraction results, with abstractions being provided in 7 s for the larger suite of 2600 tests. However, the time permitted may need to be configured in *ZiBreve* for test suites that are an order of magnitude larger than the ones we have tried.

5. Feedback on Business Rule Table Coverage

Once abstraction has been applied and business rules tables created, such as the ones shown in Figure 2, it is much easier to review them. They are now in a compact and focused form, without inessential clutter. The effort of adding new cases has changed from writing a whole new test, with minor variations from others, to the effort of adding a single row. In addition, it is now possible to apply analysis techniques.

Within the tool, the test designer chooses a business rule table and requests that it be analyzed. The user is informed of redundant input cases in the table that are a special case of another row, and cases that contradict each other. *ZiBreve* also analyzes the input data from the table for incompleteness and offers suggestions in a separate Analysis table.

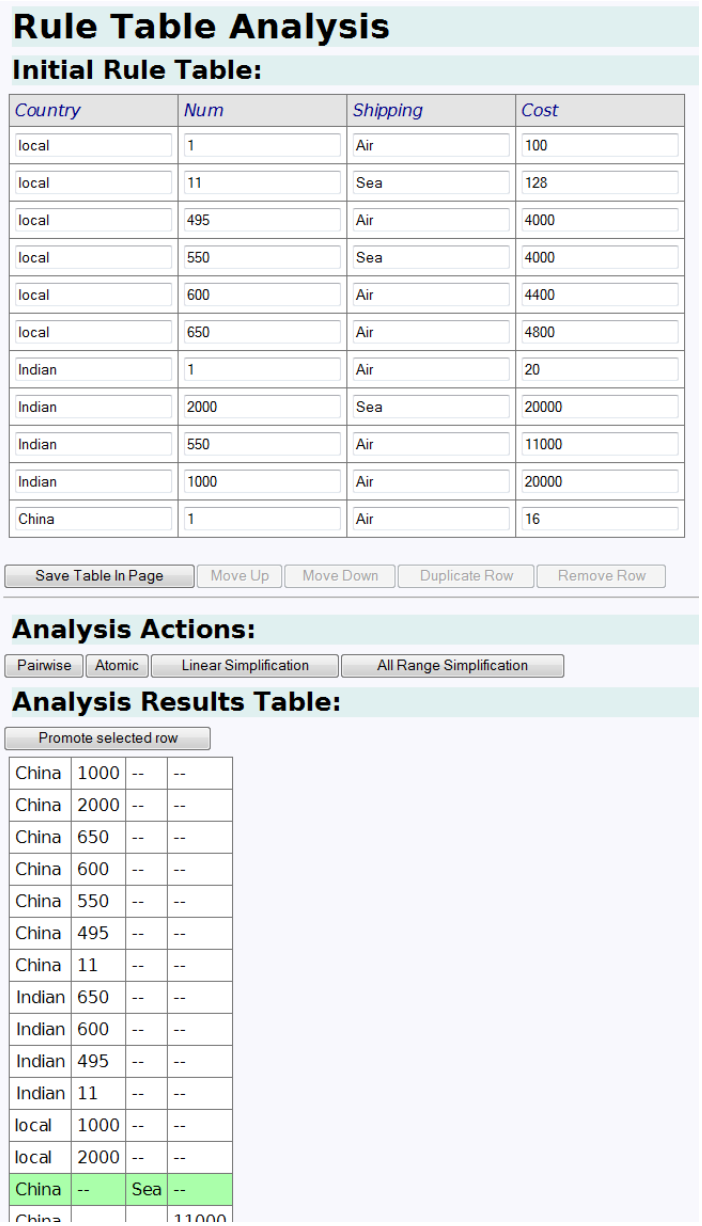
For example, the table at the top of Figure 9 is a business rule table after it has been selected. The bottom Analysis table (shown in part) results from applying pairwise analysis. This algorithm builds the domain of values for each column, takes pairs of domains and builds the Cartesian product. It adds a row to the bottom table for any pairs that are not in the top table, with all other columns set to "–".

The designer is able to promote a selected row from the bottom table to the top and change it to make it valid. The rows in the top table can then be reordered and changed, before repeating an analysis until the designer is satisfied.

We use several algorithms for analyzing such tables for redundancies, inconsistencies, missing cases, and other forms of incompleteness [19]. We have explored several algorithms for generating suggestions. Some of them (e.g., pairwise analysis of enumerated values, and MC/DC analysis of input/output

causality relationships) suggest additional test cases that may be missing from the input table [20]. Other, more sophisticated algorithms generate suggestions that try to abstract one or more of the existing tests in the input table, in order to generalise some of the input table (e.g., replacing one input column by the special "wildcard" character "–", or by a range of allowable values, "n..m"). This can raise the abstraction level of the tests, and make them slightly more specification-like.

Figure 9. Business rule table coverage.



6. Conclusions and Future Work

There are many benefits in shifting focus from test automation at the implementation level to business-level, example-based specifications. The specifications are much easier to maintain and evolve. They can be used in an agile approach to development, supporting collaboration, communication and understanding among the wider team early in an agile iteration ("what the system should do" as well as "how"). They serve as trustworthy documentation of the business view of the system. They help ensure that work is completed, and that existing functionality has not been broken. They enable faster testing for faster feedback, and they can serve to guide brand new developments of old systems.

Our work aims to support the process of refactoring implementation-level tests to business-level specifications. Given that repetition occurs across many implementation-level tests in a suite, our tool quickly provides feedback on possible points of abstraction. The user can focus the abstraction process by selecting parts of tests, and *ZiBreve* will suggest possible abstractions and apply them. We have illustrated this cyclic process with browser-based tests for a web application.

Once business rule tables have been extracted, our tool supports the analysis of the test cases, providing feedback on redundancies, inconsistencies, and possible incompleteness.

Once the tests have been refactored to specifications, a focussed and effective domain-specific language should have been created. This can then be used directly in future evolution of the system, with *ZiBreve* supporting the corresponding evolution of the domain-specific language where needed.

We have also been considering systems that lack automated tests entirely. We have just added a variant of record-and-playback to *ZiBreve*, which incorporates abstraction as a part of the process. This allows tests to be recorded, but at a higher-level, resulting in business-level, example-based specifications, rather than implementation-level test scripts. We believe that this will be an effective way of dealing with "legacy" systems (those without test automation [21]).

Given that example-based specifications serve both as detailed requirements specifications and as automated tests, it is interesting to consider other uses. We are well underway on supporting the generation of user-level documentation from example-based specifications. The mapping from business level to implementation makes it possible to describe what is going on at multiple levels, and to incorporate screen shots from the implementation level.

Some tests, such as for web services, incorporate large amounts of semi-structured data such as XML, JSON, or YAML. Such data is not currently handled in our abstraction process, which focusses on abstracting actions. We plan to use a variation of our current algorithms for searching for potential abstractions across such data.

Acknowledgements

The authors acknowledge the financial support of the New Zealand Ministry of Science and Innovation's *Software Process and Product Improvement Project*.

References

1. Meszaros, G. *xUnit Test Patterns: Refactoring Test Code*; Addison-Wesley: Upper Saddle River, NJ, USA, 2007.

2. Adzic, G. *Specification By Example: How Successful Teams Deliver the Right Software*; Manning Publications Co.: Greenwich, CT, USA, 2011.
3. Fowler, M. *Refactoring: Improving the Design of Existing Code*; Addison-Wesley: Upper Saddle River, NJ, USA, 1999.
4. FitLibrary. Available online: http://sourceforge.net/projects/fitlibrary/ (accessed on 31 May 2011).
5. FitNesse. Available online: http://www.fitnesse.org/ (accessed on 31 May 2011).
6. Fit: Framework for Integrated Test. Available online: http://fit.c2.com/ (accessed on 31 May 2011).
7. Mugridge, R.; Cunningham, W. *Fit for Developing Software: Framework for Integrated Tests*; Prentice Hall: Englewood Cliffs, NJ, USA, 2005.
8. Fowler, M. *Domain Specific Languages*; Addison-Wesley: Upper Saddle River, NJ, USA, 2010.
9. Selenium. Available online: http://code.google.com/p/selenium/ (accessed on 31 May 2011).
10. HtmlUnit. Available online: http://htmlunit.sourceforge.net/ (accessed on 31 May 2011).
11. Poppendieck, M.; Poppendieck, T. *Lean Software Development: An Agile Toolkit for Software Development Managers*; Addison-Wesley: Upper Saddle River, NJ, USA, 2003.
12. Poppendieck, M.; Poppendieck, T. *Leading Lean Software Development: Results Are Not the Point*; Addison-Wesley: Upper Saddle River, NJ, USA, 2010.
13. Adzic, G. *Bridging the Communication Gap: Specification by Example and Agile Acceptance Testing*; Neuri Limited: London, UK, 2009.
14. Constantine, L.L.; Lockwood, L.A.D. *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design*; Addison-Wesley: Upper Saddle River, NJ, USA, 1999.
15. Beck, K. *Test Driven Development: By Example*; Addison-Wesley: Upper Saddle River, NJ, USA, 2002.
16. Beck, K. *eXtreme Programming Explained*, 2nd ed.; Addison-Wesley: Upper Saddle River, NJ, USA, 2004.
17. Evans, E. *Domain-Driven Design: Tackling Complexity in the Heart of Software*; Addison-Wesley: Upper Saddle River, NJ, USA, 2003.
18. Li, H.Q.; Thompson, S. Similar code detection and elimination for erlang programs. In *Practical Aspects of Declarative languages 2010*; Carro, M., Pena, R., Eds.; Springer: Berlin, Germany, 2010; pp. 104–118.
19. Zhu, H.; Hall, P.A.V.; May, J.H.R. Software unit test coverage and adequacy. *ACM Comput. Surv.* **1997**, *29*, 366–427,
20. Chilenski, J.; Miller, S. Applicability of modified condition/decision coverage to software testing. *Softw. Eng. J.* **1994**, *9*, 193–200.
21. Feathers, M. *Working Effectively with Legacy Code*; Prentice Hall: Englewood Cliffs, NJ, USA, 2004.