Legend: An Agile DSL Toolset for Web Acceptance Testing

Tariq M. King, Gabriel Nunez, Dionny Santiago, Adam Cando, and Cody Mack
Ultimate Software Group, Inc.
2000 Ultimate Way
Weston, FL 33326, USA

{tariq_king, gabriel_nunez, dionny_santiago, adam_cando, cody_mack}
@ultimatesoftware.com

ABSTRACT

Agile development emphasizes collaborations among customers, business analysts, domain experts, developers, and testers. However, the large scale and rapid pace of many agile projects presents challenges during testing activities. Large sets of test artifacts must be comprehensible and available to various stakeholders, traceable to requirements, and easily maintainable as the software evolves. In this paper we describe Legend, a toolset that leverages domain-specific language to streamline functional testing in agile projects. Some key features of the toolset include test template generation from user stories, model-based automation, test inventory synchronization, and centralized test tagging.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing Tools, Debugging Aids*

General Terms

Languages, Verification, Documentation

Keywords

Software Testing, Test Automation, Domain-Specific Languages, Behavior-Driven Development, Agile Development

1. INTRODUCTION

During agile software development, requirements are transformed into working software through an iterative and incremental process. Proponents of the agile movement place emphasis on team interactions, working software, customer collaboration, and responsiveness to change over other aspects of software development such as processes, documentation, and rigid planning. The idea is that although there is value in both sets of items, the former are valued more than the latter.

Several practices have been devised to support using agile methods at various stages of the software lifecycle [2, 10]. For example, during software specification it is common to formulate requirements as user stories written by business analysts. Agile design and implementation practices include story-driven modeling and behavioral-driven development (BDD) [2, 7, 8]. Agile testing involves having all members of a self-organized team collaborate to ensure the delivery of quality software that provides the desired business value [10].

Adequate functional testing requires collaboration among customers, business analysts, domain experts, developers, and testers. Business analysts elicit customer needs and capture them as requirements with defined acceptance criteria. Testers use the acceptance criteria to create test cases that validate whether the software implementation satisfies the needs of the customer. To facilitate regression testing, test cases are typically encoded as scripts that can be interpreted and executed by an automated testing tool.

In this paper, we describe a novel BDD toolset to help improve functional testing in large-scale agile software projects. The toolset, codenamed Legend, is built around a domain-specific testing language designed to enable non-technical stakeholders to read and write automated tests [9]. To complement the toolset description, we discuss our motivating challenges, background research, and related work.

2. CHALLENGES

Ultimate Software uses agile testing methodologies to validate UltiPro – a cloud-based human capital management solution [11]. In this section we describe the major challenges experienced during functional testing of UltiPro, which have motivated our research and tool development.

2.1 Test Comprehensibility

Prior to signing off on a given release, business analysts must confirm that the software is working as expected under various conditions. Running through the automated tests that were developed is one way to assist this validation activity. However, this can present a challenge if the automated tests are not written in a language that is easy to understand. Many automated testing tools use a programming language to define tests. As a result, the essence of each test may be obscured by technical details, making it difficult for non-technical analysts and domain experts to leverage them during the software development process.

2.2 Test Traceability

Traceability between tests and the requirements being tested is an important aspect of software engineering. Knowing

ISSTA '14, July 21-25, 2014, San Jose, CA, USA

which tests are associated with each requirement helps answer questions such as: (1) Is every requirement covered by an adequate set of tests? If not, what are the gaps in coverage? (2) If a requirement changes, how many tests will be affected by that change? and (3) If a test that previously ran and passed is now failing, what features or requirements do these tests actually exercise?

For agile software projects which emphasize responsiveness to change, establishing and maintaining traceability relationships on a large scale can be challenging. Without proper tooling, traceability links across user stories, documented tests, and automated test scripts can easily become outdated. As the number of requirements and tests in the project grow, the aforementioned questions become increasingly difficult and eventually impossible to answer.

2.3 Test Documentation

To mitigate the comprehensibility challenge and promote test reviews early in the development process, testers create a set of documented test cases that describe the high-level scenarios to be tested. For test management in large-scale projects, documented tests are typically stored in a centralized test inventory or repository.

As the application evolves, testers must keep both the inventory of documented tests and the source repository of automated tests up-to-date. Maintaining both sets of artifacts in two separate places represents significant overhead in large projects. The end result is that the documentation, rather than the automation, becomes obsolete. This is because failing test automation is more visible to the business than poorly maintained test documentation.

2.4 Test Selection

In projects where there are hundreds or even thousands of test cases, the ability to filter and select a strict subset of tests for execution is vital. To support test selection, many tools provide mechanisms for tagging test cases with user-defined categories. However, the tagging feature of these tools tends to be general-purpose, allowing users to enter any value for a given tag. This can be problematic and lead to tests being excluded from a selection if: (1) different tags are used for tests that should be in a single category, i.e., synonyms; and (2) typographical errors are made when tagging test cases because even a slight variation in spelling will be treated as a new category.

2.5 Test Fragility

The way in which an automated test is written can impact its ability to provide consistent results. One of the risks with using a testing framework that lives in the context of a programming language, is that test authors may start to use programming constructs that make their tests less reliable. For example, while it may be acceptable to use conditionals, loops, and thread sleeps in program code, using these constructs in test code introduces non-determinism. This produces tests that exhibit different behavior on different runs even though the system under test has not changed.

3. DOMAIN-SPECIFIC WEB TESTING

To tackle the aforementioned testing challenges, we devised a novel, model-driven, test specification approach. Our approach, presented in Santiago et al. [9], provides a method for architecting a test specification and automation language

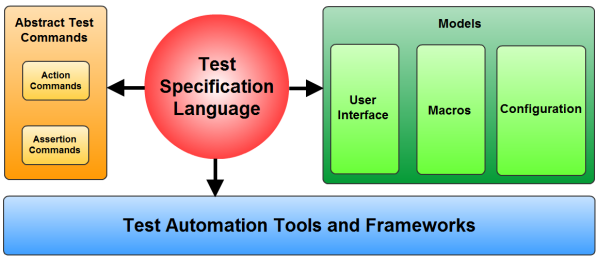

Figure 1: An automated test specification DSL.

that integrates with domain models. We applied our testing approach to the domain of human capital management [11].

Domain-Specific Languages (DSLs) are special-purpose computing languages created to describe tasks in a particular field. DSLs provide stakeholders with a common vocabulary for describing application elements and behaviors. Using a common language helps to improve communication in software projects, and reduces the probability of errors that may arise due to misunderstandings between stakeholders.

Figure 1 shows the major components of a DSL-based testing language under our approach. These are: (1) Abstract Test Commands – a set of action and assertion commands, which apply input to and validate the output of the system under test respectively; (2) Application and Configuration Models – object-oriented abstractions of the web UI, along with reusable test operations called macros, named using domain terminology; and (3) Test Automation Tools and Frameworks – programs and libraries for interacting with the UI and persistent storage of the system under test.

The result of the approach is a well-designed DSL that allows non-technical stakeholders to read and write automated tests, better engaging them in software testing activities. Furthermore, the DSL serves as a focal point for developing practical tools to support functional testing.

4. LEGEND

Our DSL-based testing toolset, codenamed Legend, was developed in C# as a Visual Studio extension. User interface elements were implemented using Windows Presentation Foundation [6], while workspace and compilation services were realized via the Visual Studio SDK and Roslyn .NET compiler platform [3, 4]. At the lower level, a selenium-based framework called Echo is used for test execution [9].

4.1 Overview

Figure 2 provides a high-level overview of the Legend workspace. The workspace consists of three major components, which correspond to the labels in Figure 2: (a) DSL Editor – a test authoring environment that facilitates text editing, and includes features such as syntax coloring, error highlighting, intelligent suggestions, block outlining, tool tips, and breakpoints (b) Data Manager – a grid control for adding, removing, and updating different types of test data; and (c) Test Explorer – a navigable tree control for browsing, searching, organizing, and executing tests.

4.2 Features

The main features of Legend are: story linking and navigation, test template generation, model-based automation, test inventory synchronization, and centralized test tagging.

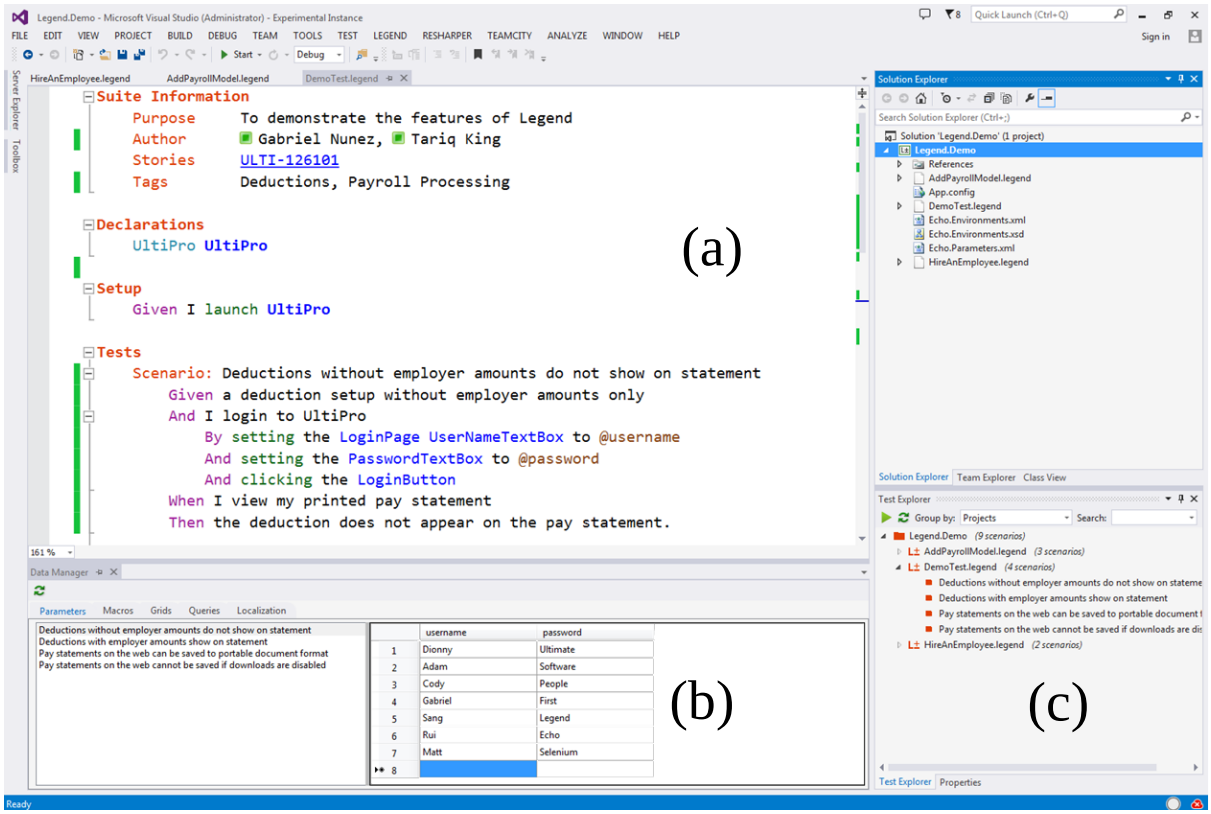

Figure 2: Legend workspace components: (a) DSL Editor, (b) Data Manager, (c) Test Explorer.

4.2.1 *Story Linking and Navigation*

The DSL editor provides the user with the ability to link one or more business requirements to a test suite. This is achieved by entering a list of user story identifiers into the `Stories` section of the `Suite Information` block (top of Figure 2a). Upon inputting a story identifier, Legend queries the requirements inventory to ensure it is valid, and transforms the identifier text into a clickable hyperlink. Clicking the hyperlink opens a tab within the project workspace and automatically navigates to the user story. To facilitate requirements traceability, Legend requires that each test suite be linked to at least one user story, otherwise a syntax error is shown on the editor.

4.2.2 *Test Template Generation*

As part of the authoring experience, users can choose to `Create a New Legend Test from User Story`. This feature prompts the user to enter the unique identifier for the story under test. Legend then generates a template to aid the user when writing automated tests against the business requirements contained in the story. The result of test suite template generation is a syntactically correct Legend test file, pre-populated with the following information: (1) a set of high-level test scenarios copied from the acceptance criteria defined in the user story; (2) the author name extracted from the active directory service of the organization, and (3) a clickable hyperlink to the user story under test.

Test suites created in this manner are comprehensible by all stakeholders because the high-level test steps are expressed using business language. Furthermore, the test suite is automatically made traceable to the author and user story.

4.2.3 *Model-Based Automation*

High-level test steps can be refined with automation commands in a stepwise fashion. On the line immediately following a test step, the author can use the `By` keyword to indicate that he or she wishes to add a block of automated test commands. Each automated test command acts upon one or more elements that model the web application under test. For example, on the third line of the `Scenario` defined in the `Tests` block (Figure 2a), the token `LoginPage` refers to a web page abstraction for the login screen of `UltiPro`. Abstractions for web UI controls such as the `UsernameTextbox`, `PasswordTextbox` and `LoginButton` are further encapsulated within the `LoginPage`. Examples of automated test commands for interacting with these elements include `set` and `click`.

Any data values specified in tests may be replaced with formal parameters defined by prefixing a parameter name with the @ symbol, e.g., `@username`. The actual values used in parameterized tests are maintained in the `Data Manager` window (Figure 2b). Recall that special types of domain-specific automated test commands called macros are integrated into Legend. Macros are reusable, user-defined test commands that may have data associated with them. As a result, the data manager window also serves to organize the formal parameters and values passed into test macros. Having a separate window for data management helps avoid technical clutter in test automation.

The Legend toolset contains a DSL compiler that supports the execution and debugging of automated tests. Tests are

executed via a context menu in the DSL editor, and the user may set debugging breakpoints to suspend a test during execution. Furthermore, the Legend DSL compiler ensures that test automation guidelines are always enforced. In other words, since we have full control over the language definition and formatting rules, we can fail compilation if test documentation and coding standards are not met.

4.2.4 Test Inventory Synchronization

To facilitate searching, reviewing, and organizing documented tests outside of the authoring environment, Legend seamlessly integrates with external test case inventory systems, e.g., Team Foundation Server, Zephyr [1, 5]. A context menu within the DSL editor provides the test author with an option for publishing the active test file to the inventory. Legend then parses the test suite and inserts the test scenarios into the inventory via API calls. The first time a test suite is published to the test inventory, new test case artifacts are created within the inventory. However, subsequent synchronizations of the test suite triggers an update to the existing artifacts stored in the inventory.

Legend provides a mechanism for directly viewing test inventory artifacts from within the project workspace. The test author can use this feature to confirm the synchronization was successful, or view any comments that reviewers post on the test cases in the inventory. Legend also makes it easy for authors to request feedback on their test cases. This is achieved through mechanisms for e-mailing or instant messaging team members listed as authors.

4.2.5 Centralized Test Tagging

Challenges associated with test filtering and selection are tackled by incorporating a centralized taxonomy of tags into Legend. The DSL editor presents test authors with a list of valid test categories that can be inserted into the `Tags` section of the `Suite Information` block (Figure 2a). These categories are then consumed by the `Test Explorer` component (Figure 2c), which facilitates viewing test cases by `Tag`, `Author`, or `Project`. While browsing tests in any of these logical views, users can navigate directly to individual test scenarios or selectively execute a subset of tests. During test execution, the test explorer window provides real-time visual feedback on which tests passed or failed.

5. RELATED WORK

Several behavioral-driven development (BDD) tools have been developed to tie acceptance tests to business requirements [2, 7, 8, 10]. Most of these tools are general-purpose and work by associating two sets of files – specifications and test step definitions [2]. Test automation resides in the test step definition files and are typically implemented in a technical language, limiting tool usage to technical users for writing automated tests. On the other hand, Legend combines test specification and automation activities into a single, non-technical, test authoring experience. The test automation language itself is english-like, allowing non-technical stakeholders to read and write automated tests.

Similar to test template generation in Legend, SpecFlow [7] and StoryQ [8] allow the shell of a test step definition file to be automatically generated based on a given specification. This is done by manually recreating the user story in a separate generation tool and/or code repository. We improve upon this idea by: (1) embedding the template generation process into a unified toolset, and (2) integrating the toolset with a requirements inventory to reduce manual effort and story duplication.

Legend also differs from the aforementioned tools in that it leverages a DSL-based compiler to enforce test documentation and automation standards. To the best of our knowledge, the toolset described in this paper consists of the first BDD framework to incorporate DSL checking into software testing artifacts. Lastly, test documentation-to-inventory synchronization is another aspect of our toolset that is not provided by previously existing BDD frameworks.

6. CONCLUSION

This paper described Legend, a DSL toolset aimed at streamlining functional testing in large-scale agile projects. The Legend toolset has been applied in the context of testing UltiPro [9, 11]. Support for other applications can be achieved by creating domain models and plugging them into the underlying framework. Future work calls for the development of features for database interaction.

The authors would like to thank Sang Nguyen, Rui Lin, Matt Wallick, and Robert Vanderwall for their contributions to this work. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors, and do not necessarily reflect the views of the Ultimate Software Group, Inc.

7. REFERENCES

[1] Atlassian. Zephyr for JIRA - Test Management, April 2014. `https://marketplace.atlassian.com/plugins/com.thed.zephyr.je`.

[2] A. Hellesoy and M. Wynne. *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Programmers. Pragmatic Bookshelf, 2012.

[3] Microsoft Corporation. Roslyn Compiler, April 2014. `msdn.microsoft.com/en-us/vstudio/roslyn.aspx`.

[4] Microsoft Corporation. Visual Studio, April 2014. `msdn.microsoft.com/en-us/library/bb166441.aspx`.

[5] Microsoft Corporation. Visual Studio TFS, April 2014. `msdn.microsoft.com/en-us/vstudio/ff637362.aspx`.

[6] Microsoft Corporation. Windows Presentation Foundation, April 2014. `msdn.microsoft.com/en-us/library/ms754130.aspx`.

[7] G. Nagy, J. Bandi, and C. Hassa. SpecFlow: Pragmatic BDD for .NET, November 2009. `http://www.specflow.org/specflownew/` (July 2013).

[8] R. Fonseca-Ensor. StoryQ - Portable embedded BDD framework for .NET 3.5, July 2010. `http://storyq.codeplex.com/` (April 2014).

[9] D. Santiago, A. Cando, C. Mack, G. Nunez, T. Thomas, and T. M. King. Towards Domain-Specific Testing Languages for Software-as-a-Service. In *ACM-IEEE MODELS 2013: Workshop on Model-Driven Engineering for High Performance and Cloud Computing*, volume 1118, pages 43–52. CEUR, 2013.

[10] ThoughtWorks. Twist Agile Testing, April 2014. `http://www.thoughtworks.com/products/twist-agile-testing`.

[11] Ultimate Software. UltiPro Enterprise: Human Resource Information Systems Solutions, April 2014. `www.ultimatesoftware.com/HRIS`.

APPENDIX

This section contains a detailed set of demo steps and sample artifacts for showcasing Legend.

A. DEMO STEPS

Step 1: Generate a new test suite template using acceptance criteria from a user story.

a) Add a *New Legend Test from User Story* using the identifier of the sample user story.

b) Use the story hyperlink to compare the acceptance criteria with the generated test scenarios.

c) Attempt to link the test suite to a user story that does not exist to show validation.

Step 2: Add an additional high-level test scenario to the generated test suite.

a) Insert a new test scenario template via intellisense by pressing *CTRL+Space* in the editor.

b) Fill in the scenario title and Given, When, Then behavioral descriptions.

Step 3: Refine a high-level step with test automation.

a) Add the sample test automation to showcase intellisense on application models.

b) Toggle the block outlining of the refined step to show or hide the test automation details.

c) Replace string literals in test automation with parameters and add rows to the data manager.

Step 4: Perform stepped execution of a test scenario.

a) Set a breakpoint on one of the test automation commands.

b) Invoke the test runner on the corresponding scenario block.

c) Click the *Step Over* button on the toolbar to debug through the automated test.

Step 5: Publish tests to inventory and request a review.

a) Right-click within the editor and select *Publish to Test Inventory*.

b) Confirm that the test suite was published by selecting *Go to Work Item*.

c) Right-click the editor and select *Copy Work Item to Clipboard*.

d) Add a reviewer to the authors list and send them a message containing the work item link.

Step 6: Explore and execute tests based on search criteria.

a) Tag tests with categories from the built-in feature taxonomy.

b) Use the test explorer to group tests by tag, author, and project.

c) Navigate directly to a test scenario by double-clicking it in the test explorer tree structure.

d) Perform a selective regression test run by tag.

B. SAMPLE ARTIFACTS

UltiPro / ULTI-126101

Legend Sample User Story

Title	Given	When	Then
Deductions with employer amounts show on statement	A deduction setup with employer amounts only and I login to UltiPro	I view my printed pay statement	The deduction appears on the pay statement and also shows on the pay tab summary
Pay statements on the web can be saved to portable document format	Downloads are enabled for web pay statements and I login to UltiPro	I view my web pay statement and click Download PDF	A file dialog should appear prompting me to enter the disk location where I want the pay statement to be saved
Pay statements on the web cannot be saved if downloads are disabled	Downloads are disabled for web pay statements and I login to UltiPro	I view my web pay statement and click Download PDF	No file dialog appears on the screen because the Download PDF button is disabled

Additional Test Scenario

```
Scenario: Deductions without employer amounts do not show on statement
  Given A deduction setup without employer amounts only
  And I login to UltiPro
  When I view my printed pay statement
  Then the deduction does not appear on the pay statement
```

Sample Test Automation

```
And I login to UltiPro
  By setting the LoginPage UserNameTextBox to @username
  And setting the PasswordTextBox to @password
  And clicking the LoginButton
```

Parameterized Test Data

username	password
Dionny	Ultimate
Adam	Software
Cody	People
Gabriel	First