Automated Web Testing based on Textual-Visual UI Patterns: the UTF Approach

Jingang Zhou and Kun Yin
State Key Laboratory of Software Architecture (Neusoft Corporation)
116085, Dalian, China
{zhou-jg, yink}@neusoft.com

ABSTRACT
Automated software testing is the only resort for delivering quality software, since there are usually large test suites to be executed, especially for regression testing. Though many automated testing tools and techniques have been developed, they still do not solve all problems like cost and maintenance, and they can even be brittle in some situations, thus confining their adoption. To address these issues, we develop a pattern-based automated testing framework, called UTF (User-oriented Testing Framework), for Web applications. UTF encodes textual-visual information about and relationships between widgets into a domain specific language for test scripts based on the underlying invariant structural patterns in the DOM, which allows test scripts to be easily created and maintained. In addition, UTF provides flexible extension and customization capabilities to make it adaptable for various Web-application scenarios. Our experiences show UTF can greatly reduce the cost of adopting automated testing and facilitate its institutionalization.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging –*Testing tools.*

General Terms
Design, Languages.

Keywords
Web application; Automated testing; Domain-specific language; User-interface pattern

1. INTRODUCTION
Software functional testing is an essential part of a software development process ensuring that a software system meets its functional requirements before its release or delivery to customers. In industrial practice, functional tests are intended to be executed by humans according to the functional specification or test cases of the system under testing (SUT). However, in many cases the test cases are large, which makes it costly and impractical for a manual approach, especially in situations where testing needs to be performed repeatedly, e.g., regression testing in an agile development environment [1]. This is why an automated-testing approach is desired. The idea behind automated testing is mimicking what a human can do with the SUT and seeing whether the desired results are achieved [16].

For automated testing, an essential question is how to construct test scripts that then can be executed by a machine. From the view of test-script construction, there are generally four kinds of approaches. They are *record-and-play* (R&P), *keyword/table-driven frameworks* (KDF), *automated generation and transformation* (AGT) and *fully manual programming* (FMP).

R&P may be the most simple and intuitive way, but its disadvantages are also evident. Current R&P tools require not only recording tests manually by interacting with the SUT, inserting verification points and modifying the test scripts to fix changes continuously, but also the effort required to understand and maintain the recorded scripts, which are normally hard to read [1, 2]. Furthermore, scripts generated by recording are usually platform-dependent and hence are not cross-browser and are even brittle in some complex and dynamic GUI situations [3, 4].

KDF involves creating a library of reusable functions (or keywords), translating a manual test into a sequence of keywords, and writing a driver program to interpret the keyword sequence. Usually, a KDF is limited by the expression capability of its keywords. In addition, developing such an automation framework is expensive, requiring significant time and expertise [1].

AGT focuses on transforming test cases from a natural language version to a well-formatted or structured version by extracting information like actions, targets, and so on, which then can be translated to executable scripts according to, e.g., a KDF. However, the feasibility of this approach is determined by the structure quality of the test cases in the natural language version, and also requires human interventions for extracting some non-deterministic actions in the transformation process.

The limitations of the above approaches leave room for FMP, where testers write whole test scripts manually either with a general-purpose language like Java or a domain-specific language like LiFT [5]. Unlimited expression power makes this approach popular in industry and many driver frameworks were developed (e.g., JUnit [1] and TestNG[2]). However, FMP requires a tester to have programming skills, which is not desirable for non-technicians. In addition, script writing in FMP is generally tightly coupled with the code implementation of the SUT, which usually leads to insufficient testing due to delivery pressure and contradicts agile paradigms.

To leverage the expressive power of FMP and comply with existing testing habits, and to allow automated testing to be adopted smoothly and widely in industrial environments, we propose an agile automated-testing approach for Web applications (WAs) with easier test-script construction and organization. The central idea is writing scripts in a high-level abstraction to improve productivity and lower maintenance cost, and separate widget localization from scripts with design contracts between test scripts and code implementations. Our approach is particularly suitable for WAs developed with UI libraries or frameworks (which have consistent coding standards, thus the design contracts are easier to establish in these situations), and we believe framework or platform-based WA development is the norm for current practice. The main characteristics of our approach are:

- *Change-resilient*. Decoupling test scripts from code implementation to reduce the change effects caused by the latter as much as possible.

[1]http://junit.org/

[2]http://testng.org/doc/index.html

DOI:10.1145/2659118.2659136 http://doi.acm.org/10.1145/2659118.2659136

- *User-oriented*. Making test scripts easy to read, write and maintain for all stakeholders of a system, including non-technicians and even end users.
- *Lightweight*. Adopting the approach, which includes infrastructure construction and usage learning, is easy and can be afforded by most (if not all) companies and organizations.
- *Pragmatic*. Fitting into most test scenarios and providing extension and adaptation mechanisms for those not directly supported scenarios with little effort.

Though some ideas of our approach are not new (e.g., script abstraction [6]), we have found that our approach provides a more feasible and practical way to adopt automated testing for modern WAs, especially enterprise applications. The main contributions of the paper are:

- A pattern-based approach to Web widget encapsulation, which allows users to specify UIs at a higher level of abstraction.
- An XML-based domain-specific language, which allows users to write test scripts declaratively.
- An automated Web-testing framework, which demonstrates the concepts above and allows extension and customization.
- Some lessons learned when performing empirical evaluations on real industry projects.

For the rest of the paper, we present technical details of our approach and our practices in Sections 2 and 3, respectively. Discussions and lessons learned are in Section 4. Related work is in Section 5. We conclude in Section 6.

2. USER-ORIENTED TESTING FRAMEWORK

Our approach basically belongs to FMP, which allows users to freely express, easily write and understand, as well as organize their test logic. Our approach is also based on a testing framework, of which agility is the soul in its design philosophy. A main characteristic of this agility is that it allows all stakeholders of an application to understand the scripts and maintain them. This is the main reason why we call the framework a user-oriented testing framework, acronymed UTF. Before we discuss the technical aspects of UTF, let's first demonstrate its general use from a tester's view.

2.1 A Tester's View

To make the description clear, we use a typical *retrieval* scenario (Figure 1) as an example. A user needs to enter a query string ("test" in our case), then clicks the Query button to see available templates that can be deployed in a virtualization environment. Finally, the user selects one template by clicking the Rec. button in that row, after which the recommended template will be removed from the template list. Though this scenario is specific, the structure of the Web page and related interactions are common to many WAs.

Figure 2 illustrates the script for the test scenario of Figure 1. Each test step is encoded with **<event>**, which has four attributes, *id* denotes the target (Web widget or control) involved in the interaction, *name* and *value* denote the operation performed on and the value property of the widget, respectively. Another (not shown) attribute *wait* denotes the time needed for Web page rendering after the interaction. The Appendix gives a full description of the elements and attributes used in our script language. Thus, a tester can write a piece of script simply as a pattern of <web widget, [operation], [value], [wait]>, and UTF allows some parts of the pattern to be omitted since they have default values according to the context. For example, line 6 of Figure 2 only specifies the widget without other attributes because the only valid operation for a button is click.

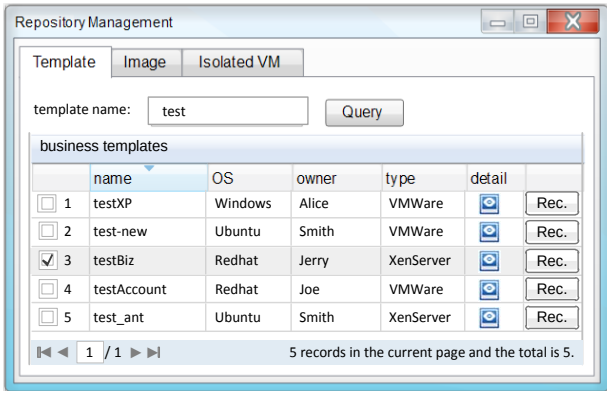

Figure 1. Exemplar Web page for business templates selection.

```
1  <test-suite>
2    <test-case name="VM Template Recommendation"
3        url="/aclome/vmTempRec.action">
4      <method id="queryVMTemplate">
5        <event id="[input]template name"
                name="setValue" value="test"/>
6        <event id="[button]Query"/>
7      </method>
8      <method id="recommendVMTemplate">
9        <event id="[gridRow]business templates, testBiz"
             name="Rec."/>
10     </method>
11     <method id="assertRecommend">
12       <event id="[input]template name" value="test"/>
13       <event id="[button]Query"/>
14       <event id="[gridRow]business templates, testBiz"
           name="assertNotExist"/>
15     </method>
16   </test-case>
17 </test-suite>
```

Figure 2. UTF script for test scenario in Fig. 1.

We use visual information as clues on a Web page to locate a widget or control. For example, in line 5 of Figure 2, we use a label ("template name") with a pattern type ([input]) to locate an input field. Such visual information can be grouped in visual_clue hierarchies in the pattern of

'[' widget_pattern']' visual_clue (, visual clue)*

For example, in line 9 of Figure 2, we use the text information of a grid and a column to find the needed row. Hierarchy is very useful for complex widgets like grid, tree, etc. and also allows widgets with the same visual information to be described in a different context (c.f. Figure 6). This kind of abstraction not only indicates which widget to interact with and how to proceed, but also allows testers to neither rely on nor assume its underlying implementation, hence making the scripts easy to evolve and change-resilient. In addition, such an expression also aligns with the mental activity of a user when interacting with an application. Take the user management scenario of Figure 3 as an example and read the script words in bold.

```
<event id="[input]Account" value="Tom"/>
<event id="[input]Password" value="…"/>
<event id="[select]Gender" value="Male"/>
<event id="[date]Birthday" value="03/10/80"/>
…
```

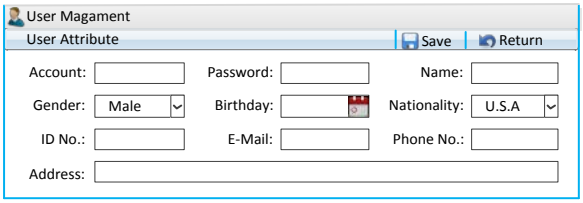

Figure 3. Exemplar Web page for script readability

Our script language provides *Equal*, *NotEqual*, *Exist*, *NotExist*, *GreaterThan*, *GreaterOrEqual*, *LessThan*, *LessThanOrEqual*, *Null*,

DOI:10.1145/2659118.2659136 http://doi.acm.org/10.1145/2659118.2659136

NotNull assertions for use. For example, in line 14 of Figure 2, we assert that the specified template (which has been recommended) does not exist in the grid (result list). Parameterization is also supported in UTF using `${val}` in which `val` represents a parameter name.

Besides the essential test steps, there are three other elements grouped in a hierarchy to play an organizational role for test scripts, since test step grouping and test case organization are also important issues [7].

- **`<test-suite>`** is the root of a test-script file, which contains one or more test cases.
- A **`<test-case>`** typically represents a business process in a test scenario, which can be further divided into several test methods.
- A **`<method>`** represents a high-level UI interaction for a business objective, which includes a sequence of low-level UI interactions, i.e., **`<event>`**s.

To organize large test cases, we provide another container file (shown in Figure 4) for organizing test-suite files (each has the format shown in Figure 2). The attribute *path* indicates where to find the test suite file. The attribute *enabled* (default value is true) indicates whether test cases in this file are to be executed or not in the testing. All the test suites are executed sequentially.

```
<suite-files>
  <test-suite path="filePath" enabled="true|false"?/>*
</suite-files>
```

Figure 4. Structure of file for all test suites.

2.2 Invariant UI Code Pattern-based Implementation

Patterns have been used extensively in software development for two decades and provide a vocabulary among software engineers for expressing architectural visions and concise representative designs, as well as detailed implementations [17]. Besides software development, other domains like human-computer interaction also heavily use patterns. Patterns can be classified into various categories. The patterns we are concerned with here are visual-textual patterns between widgets and their underlying structures (code patterns). These patterns are usually associated with coding standards enforced by a development organization or specific UI frameworks or libraries.

Consider the textual-visual pattern of "label for" shown in Figure 5, in which the preceding label "template name :" grabs focus for its following text field with id "tName". This code pattern is very common to WAs. Behind this UI pattern, the code structure between these two kinds of widgets is invariant for the same UI frameworks at the DOM level. This invariant relationship is illustrated in Figure 5 according to the Dojo[3] framework. With this kind of textual-visual information and invariant code structure among widgets, we can locate needed widgets using XPath expressions. We can list similar textual patterns among widgets, such as from a tree root to any tree nodes, from a table to any table rows, columns, and cells, and so on.

Our implementation of widget localization takes a similar approach with the helper pattern mentioned by A. Bruns et. al [8], in which the authors encapsulate widget-localization code (that use XPath expressions as widget locators) into helper methods to make test code more readable and manageable. But we provide a higher-level abstraction and a systematic way tailored to UI frameworks or libraries rather than one system or application to get a greater return on investment (ROI) from testing infrastructure. This kind of encapsulation of widget localization relies on code implementation standards in Web UIs. Thus, it works best for WAs derived with the same UI frameworks or libraries, i.e., product lines or families.

[3]http://dojotoolkit.org/

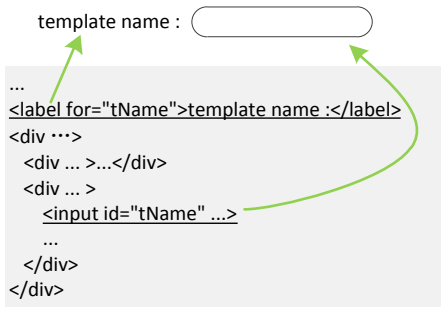

Figure 5. An example of the label for pattern.

UTF contains three core modules, a script editor for test-script writing with content assist, an interpreter that translates XML text into executable code, and an execution engine that locates widgets and delegates test execution to Selenium [10] (specifically the WebDriver component [11]). Other modules like result analysis and reporting make UTF a total test-automation solution. The widget locator manager provides flexible strategies for extension, which enables UTF to be reusable and adaptable for diverse WAs.

2.3 Framework Extension

Since there are so many UI frameworks or code standards, UTF cannot directly support them all. But UTF provides mechanisms for users to adapt UTF to their environment by using extensions.

The simplest extension point provided by UTF allows users to directly write a Java class (e.g., Demo) with test methods (e.g., foo()) for cases where the test event is just a backend operation or UI interactions are not easily expressed with UI information, e.g., simulating a reader that reads ID cards. Then users can use the following syntax in their test scripts:

```
<event id="[Java]Demo" name="foo" value="val"?/>
```

Another more general extension point provided by UTF allows users to define their own custom test events with corresponding interpreters. For example, Figure 6 depicts a page for bug review in a fault management system. The available operations are provided through buttons (area enclosed with a solid red rectangle in Figure 6). For this business specific context, a user may encapsulate his/her test scripts as `<event id="[bug]No.5" name="Confirm"/>` where `[bug]No.5` specifies the area enclosed with the red dashed rectangle and `Confirm` indicates the operation to be performed on the specified button. This kind of expression is business-oriented and friendly to users.

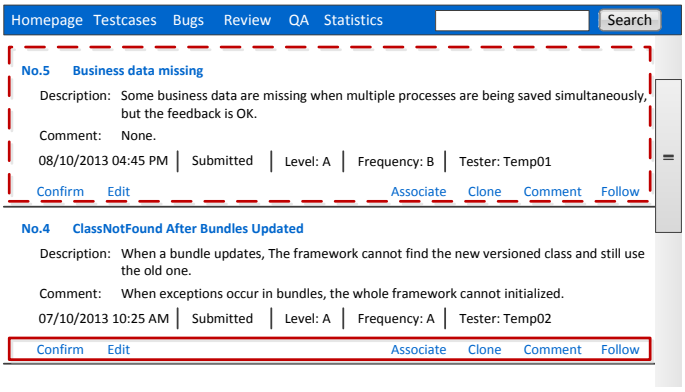

Figure 6. A screenshot of the bug review page in a fault management system.

In addition to the two simpler extensions above, UTF provides another complex yet advanced extension point that allows users to set their own event interpreters and widget locators under a target UI framework using the abstract factory design pattern [13], i.e., register these locators with a locator factory, then register the locator factory into the widget

locator manager of UTF. Thus the UTF locator manager can forward widget localization tasks to the proper locators. In this way, we provide direct support for UI frameworks like Dojo, Ext[4], JQuery[5], and SmartClient[6] in the UTF tool suite.

3. INDUSTRIAL PRACTICE

In the last two years, UTF has been applied in several development organizations and business lines of a large enterprise application provider in China and we established an automated-testing culture for that company. We discuss our experiences using UTF in practice in the following subsections.

3.1 Enabling Testing Ahead

UTF allows test script writing activities to be started immediately when Web page design completes, which is parallel with or ahead of code implementation, so long as a code contract is established under a common UI pattern specification between the two. This can be seen as another kind of test-driven development (TDD) paradigm [19], which does not care about the underlying code implementation and refactoring. Some development teams are trying to replace test cases in natural language with UTF test scripts. They tell us that writing test cases in two languages is a waste of time and the best documents (test cases) are the scripts themselves. This is more evident when using a more business-oriented format like Excel. However, not all teams agree with this point. Therefore, we still need experimentation on script format and tools to make executable scripts replace the test cases in conventional natural language.

3.2 Script Construction

Though UTF provides a script editor for writing scripts, some testing teams are developing their own script tools and integrating them in their own test-management tool suites. For instance, one development team developed an editor with a tree panel for organizing scripts on the left and a grid panel for editing test events on the right. Another team uses a customized table format based on Excel in a more readable format for their scripts, similar to the end-user development paradigm [18].

We recommended that some teams use a model-driven approach to generate scripts automatically during the code-generation phase according to the UI patterns and the code standard specified in the transformation-and-generation model, since they apply a domain-specific modeling approach to WA development [15]. But we found that testers were not as interested in such a method as we expected. The main reason is that the generated scripts are disordered, ungrouped, etc. and also require additional editing for parameters and assertions, which bring much cognitive burden on testers for reorganizing the generated scripts. So, more interactive support should be considered in tool design for test-script generation and the role of testers should also be taken into consideration in UI modeling and meta-modeling.

3.3 Advanced and Coarse Assertions

Too many basic assertions (e.g., value-equation assertions) scattered in scripts lowers readability. We provide a coarse assertion on the test-method level (scope of a **`<method>`** element) by image comparison implicitly, i.e., UTF captures a screenshot as the execution result for each test method and compares it with the one in the benchmark automatically.

Based on image comparisons, we recommend that testers organize their scripts in logical (operational) order so that they can locate and analyze a fault easier when the fault occurs in test methods having causality relations. This approach is very efficient for regression testing since most functionalities are stable on UIs and most test scenarios are data driven like the *create-read-update-delete* (CRUD) pattern. So, this kind of assertion brings a high ROI for many development teams.

3.4 Script Extension Scenarios

In practice, we have found UTF can meet more than 90 percent of the test requirements and scenarios for enterprise applications directly. Most of the unsupported ones can be categorized as follows and be realized by the extensions presented in Section 2.3:

- Environment initialization, e.g., data initialization. Many test scenarios require loading a plenty of business data first, e.g., by executing SQL script files or calling some service APIs.
- Ah-hoc or complicated UI interactions, e.g., widget drag and drop in which we only care about the result, very specific widgets like SVG or Flash, and getting results from hardware devices, e.g., an ID-card reader.
- Backend assertions. Assertions on the business logic and even database states.

3.5 Quality Assurance

With a pattern-based approach like UTF, many quality-related problems (anti-patterns) can be easily found. Such problems are listed below:

- Duplicated *id* values among different controls on the same page; *id* is blank or violates established norms or conventions.
- Useless DOM objects are not properly destroyed, which may lead to memory leaks.
- Using white spaces for layout purposes, which leads to cross-browser problems.
- Wrong groups of controls with the same visible text.

Most of these problems are not easy to find with a manual or R&P approach. Therefore, UTF not only serves automated testing, but also provides an effective means to drive code testable by detecting and eliminating those anti-patterns, and enforcing code standards in an automated environment.

3.6 Adoption Cost

There are three main cost factors for UTF adoption:

- Cost of UTF construction. The code for core UTF functionality is nearly 10KLOC and took 3 man-months for developers familiar with Selenium WebDriver, XPath, and TestNG.
- Cost of specific UI framework-oriented adaptation and extension. Our experience in UTF promotions shows it only takes 5 man-days for developers familiar with HTML, XPath, and Java to adapt UTF to a UI framework like JQuery and Ext.
- Cost of learning the script language. According to our experience, this only needs one or two hours for testers familiar with XML.

4. OBSERVATIONS AND DISCUSSION

4.1 Script Language Design Issues

Firstly, as a domain-specific language for testing, our script language only has a few keywords (elements and attributes) and has no complex control structures like loop and branch, which allows the language to be easy to learn and use, since many testers lack training in programming practices and are reluctant to learn to use an imperative programming language. Secondly, using XML as the carrier for the language is a double-edged sword [11]. The good news is XML has a good availability and wide adoption in industry, which makes it familiar to most of the staff and there are lots of tools available to handle XML. Furthermore, the declarative nature of XML makes it preferable over other imperative languages for testers to use. The bad news is the

[4]http://www.sencha.com/products/extjs/

[5]http://jquery.com/

[6]https://smartclient.com/

DOI:10.1145/2659118.2659136 http://doi.acm.org/10.1145/2659118.2659136

verbose format of XML may make it less readable. But we have not received any feedback from users on the readability of the scripts.

4.2 Limitations

Currently, UTF has the following limitations:

- UTF does not provide script expression for images, thus it is not easy to judge a state change of an image, e.g., to judge whether an image of a button displays in gray when the button is disabled. Though the image comparison approach can be used, it is still inefficient. Chang et al. [12] provides image-based scripts for GUI testing, which can handle this issue well.
- Though UTF is browser-independent, our image comparison approach is browser-dependent because different browsers have different themes or styles for presentation, which may cause much inaccuracy for images generated in a different browser environment when compared to the images in the benchmark. We are planning to design an improved algorithm with fuzzy matching for this issue.
- Consider the script of test methods queryVMTemplate and assertRecommend of Figure 2, where the latter only adds an assertion to the former. Do we need a module (test case, test method) reuse mechanism in the language to avoid script cloning? This issue still needs experimentation and we believe it is more habitual than technical.

4.3 Lessons Learned

- Development organizations should enforce code standards in their systems to make them more testable using an automated approach.
- Pattern-based automated testing is very helpful to find anti-pattern-related problems, and hence to make systems more maintainable and testable.
- We still need a manual way for some specific scenarios in the presence of an automated approach (at least ours) to make up the limitations of the latter.

5. RELATED WORK

S. Thummalapenta et. al [1, 4] develop a test automation tool called ATA, which can be used to translate test cases written in stylized natural language into a keyword-driven representation for execution. The main idea is to extract executable instruction segments from manual test cases. ATA leverages natural language understanding and artificial intelligence techniques to transform test cases into a group of *action-target-data* (ATD) tuples, where each tuple represents a test step. However, the feasibility of this approach is determined by the quality of the test cases in terms of the structure format specification. Still, ATA requires human intervention and feedback in some situations during a transformation process to determine the proper sequence of segments in the correct order for the whole test case.

R. Chatley et. al [5] provide a framework called LiFT for writing automated tests with a domain-specific language embedded within Java and the language is tailored to testing to make the tests clearer and more readable. However, this approach requires programming skills for testers, which is undesirable for testers who usually do not have such skills.

The CoTester system [7] provides another high-level test scripting language in the ClearScript language to make the test scripts readable and maintainable. Similar to ATA, all the test instructions of the scripts are in a flat structure, which is not easy to manage. So, the designers of CoTester provide a machine-learning algorithm and train it to automatically identify subroutines from test scripts to make them more manageable. However, only a few (seven kinds) of subroutines can be identified automatically in the current implementation. UTF encodes organization information directly in test scripts with organization tags (e.g., `<method>`), which make it more practical in real use.

Zhu et. al [14] provide a solution for automated GUI functional test in which the authors design a test driver to integrate the tools for test case generation, selection, execution, and reporting. The solution uses the KDF approach and relies on the IBM Rational Functional Tester (RFT) to execute the test scripts, which needs the GUIs to be designed first then captured and recognized by RFT. Another limitation of this solution is that it only provides a few keywords and cannot handle complex UI situations.

Jubula [3] is a new keyword-driven tool added to the Eclipse platform for the automated testing of GUIs written with Java (Swing, SWT/RCP, GEF) and HTML. Tests (i.e., sequences of actions) for the SUT are constructed by dragging and dropping pre-defined modules (or test cases, or keywords). However, Jubula needs sophisticated configuration and test-case organization, as well as object mapping while the SUT runs, which makes this approach heavyweight and undesirable.

Chang et. al [12] provide the Sikuli test approach, which enables testers to write visual scripts by embedding images to locate Web widgets based on computer vision techniques. Such an approach provides an intuitive way for locating widgets and improves understandability, and is especially applicable for scenarios in which the states of images need to be verified. However, this approach lacks support for platform portability and dynamically-generated widgets. In addition, visual test scripts may not be shared or reused among different browsers since they may have different visual themes.

6. CONCLUSION AND FUTURE WORK

Automated testing is currently a hot topic in both academia and industry. In this article, we provide a pattern-based automated-testing framework for WAs, especially for enterprise applications. This approach encapsulates textual-visual information between widgets and maps these relationships to their underlying invariant code structures enforced by UI frameworks or libraries. By abstraction, encapsulation, separation of concerns, as well as reuse of open-source testing utilities, our approach not only makes automated testing feasible and affordable, but also provides a feasible way to drive a system to be more testable. Currently, UTF is widely used in a large enterprise software and service provider in China.

For future work, we are planning to experiment with a reuse mechanism in the scripting language for modules and design an advanced image-comparison technique for cross-browser compatibility testing. We also consider conducting an empirical study on the potential roles that test scripts can play. In addition, tool improvement and integration are also in planning to give better experience for testers and end users.

7. ACKNOWLEDGMENTS

The authors thank Michael Wing for his polishing the paper to make it clearer and more readable.

8. REFERENCES

[1] Thummalapenta, S., Devaki , P., Sinha, S., Chandra, S., Gnanasundaram, S., Nagaraj, D., et al, 2013. Efficient and change-resilient test automation: An industrial case study. In *Proceedings of the 35th International Conference on Software Engineering*, ICSE 2013, IEEE Press, 1002-1011.

[2] Memon, A. 2002. GUI Testing: Pitfalls and Process. *Computer*, 35, 8, (Aug. 2002), 87-88.

[3] Jubula, Automated functional testing. http://www.eclipse.org/jubula/.

[4] Thummalapenta, S., Sinha, S., Singhania, N. and Chandra, S. 2012. Automating Test Automation. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE 2012, IEEE Press, 881-891.

[5] Chatley, R., Ayres, J. and White, T. 2010. LiFT: Driving Development using a Business-Readable DSL for Web Testing. In *Proceedings of 3rd International Conference on Software Testing, Verification, and Validation Workshops*. IEEE CS Press, 460-468.

[6] Lowell, C. and Stell-Smith,J. 2003. Successful Automation of GUI Driven Acceptance Testing. In *Proceedings of the 4th International Conference on Extreme Programming and Agile Processes in Software Engineering*. Springer, LNCS 2675, 331-333.

[7] Mahmud, J. and Lau,T. 2010. Lowering the barriers to website testing with CoTester. In *Proceedings of the 15th International Conference on Intelligent User Interfaces*, (IUI 2010). ACM Press, 169-178.

[8] Bruns, A. Kornstädt, A. and Wichmann, D. 2009. Web Application Tests with Selenium. *IEEE Software*, 26, 5 (Sept. 2009). 88-91.

[9] Selenium-Web Browser Automation. http://docs.seleniumhq.org/

[10] W3C WebDriver. http://www.w3.org/TR/2013/WD-webdriver-20130312/

[11] Zhou, J., Zhao, D., Xu, L.and Liu, J. 2012. Do We Need another Textual Language for Feature Modeling-A Preliminary Evaluation on the XML based Approach. *Studies in Computational Intelligence*, 430, 97-111.

[12] Chang, T., Yeh, T. and Miller, R. 2010. GUI Testing Using Computer Vision. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. CHI 2010. ACM Press, 1535-1544.

[13] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. 1995. *DESIGN PATTERNS - Elements of Reusable Object-Oriented Software*. Addison-Wesley Publishing Co.

[14] Zhu, X., Zhou, B., Li, J. and Gao, Q. 2008. A test automation solution on GUI functional test. In *Proceedings of IEEE International Conference on Industrial Informatics* (July 2008). IEEE Press, 1413-1418.

[15] Zhou, J., Zhang, D. and Zhao, D. 2013. Pragmatic Enterprise Web User Interface Development andReuse: An Industrial Case Study. *Journal of Network & Information Security*, 4, 2, 101-116.

[16] Pradhan, L. 2011. User Interface Test Automation and its Challenges in an Industrial Scenario. Master thesis, Mälardalen University.

[17] Buschmann, F., Henney, K. and Schmidt, D. 2007. Past, Present, and Future Trends in Software Patterns. *IEEE Software*, 24, 4 (July 2007). 31-37.

[18] Ko, A., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., et. al. 2011. The State of the Art in End-User Software Engineering. *ACM Computing Surveys*, 43, 3 (Apr. 2011). Article No. 21.

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

Appendix

Table 1. Simplified concrete syntax of UTF script

Name	O	V	Semantic
testsuite	—	—	A test suite (root of a script file)
elementFinder	O	—	Test-suite-level widget finder
testcase	—	—	A test case
name	M	—	Name of the test case
url	O	—	Page's url (relative to the root of the Web application)
enabled	O	T	Whether to be tested
refresh	O	T	Refreshing the Web page when test case switches
method	—	—	A high-level business action, which aggregates a sequence of basic UI actions
name	M	—	Method's name
UIComp	O	T	Sreenshotting or not
HTMLComp	O	F	Whether to generate a backup file for source code of a Web page or not
wait	O	2s	Needed time for the Web page initialization or rendering
frame	O	—	Specify the frame in which the method executes
elementFinder	O	—	Method-level widgets finder
refresh	O	F	Whether to refresh a page when the method executes
event	—	—	A basic UI action
id	M	—	A locator for a specified widget
name	O	*	Specified operation on the widget
value	O	—	Text to be passed to the widget
wait	O	1s	Needed time after the action's execution for its following actions.

M: mandatory, O : optional, V : default value, T : true, F : false;

* : depends on the widget kind

DOI:10.1145/2659118.2659136 http://doi.acm.org/10.1145/2659118.2659136