

Make Haste, not Waste: Automated System Testing

Carl Erickson^{1,3}, Ralph Palmer², David Crosby¹, Michael Marsiglia¹, Micah Alles¹

¹ Atomic Object LLC, 419 Norwood Ave SE Suite 190, Grand Rapids MI 49506
{carl, david, mike, micah}@atomicobject.com
<http://atomicobject.com/>

² Burke Porter Machinery, 730 Plymouth NE, Grand Rapids MI 49505
Ralph.Palmer@bepco.com
<http://www.bepco.com/>

³ Department of Information Technology, Uppsala University, Uppsala, Sweden
carle@docs.uu.se

Abstract. Haste (High-level Automated System Test Environment) represents an approach to system testing that is philosophically consistent with standard XP unit testing practices. Test code runs in the same address space as the application under test, allowing for ready examination of application state. The fundamental Haste abstractions of Story, Step, and StoryBook provide a framework to implement system tests. Utility classes simplify test development. In addition to acting as XP acceptance tests, Haste tests aid source maintenance and extension, and can play an important role in a release process. This paper describes the elements of Haste, our experience with using it to test a complex Java Swing application, and the perspective of the client for whom the application was developed. Haste is available under an open source license.

Keywords. System, acceptance, automation, GUI, testing, Haste.

1 System Testing

System tests validate the soundness and behavior of the application from the user's perspective [1]. In an XP project, system tests serve as acceptance tests. In this role they measure progress on a project, and provide a form of customer acceptance of the delivered application [10]. Ideally, they are written by the customers themselves. In practice, system tests must often be translated by a programmer from an elaborated customer story into testing code. A lightweight automation framework for system tests can extend the benefits of XP unit testing to a higher level, supporting test first development for system tests, and decreasing the difficulty of writing system tests.

There appears to be an emerging consensus on the use of system testing in XP projects [9]. Consistent with this view, we use system tests for four purposes:

- regression testing for code development and maintenance
- customer communication and specification
- customer project progress gauge and buyoff
- quality control element of the release process

System testing requires exercising an application via its user interface. Without automation, system tests cannot play the same role in XP development that unit and integration tests play, and the four uses identified above for system tests would not be practical. System tests are distinct from unit and integration tests in that they validate the functionality of the application as a whole, rather than a single method, object or component. Test assertions need to be made about the state of the application as a whole, which in turn may be represented by the state of many unrelated objects. Passing a system test means every step in a potentially lengthy process was executed correctly.

Several toolkits have been created for automating system tests of web applications. Examples include HttpUnit [2], Avignon [3], Canoo WebTest [12], and jWebUnit [4]. To our knowledge, no general purpose application system testing framework, analogous to JUnit for unit testing, exists. While it is in fact built on JUnit, Haste was designed for general purpose system testing and reflects the distinct needs of system testing. Haste has been used for system testing web applications and Java Swing applications.

1.1 Automated GUI Testing

Creating automated system tests for applications with graphical user interfaces can be difficult. The rising popularity of automated unit testing seems to have inspired the creation of several GUI test toolkit projects. JFCUnit [5] has a tool class called JFCTestHelper for examining the state of the graphical environment, as well as massaging the event stream to programmatically manipulate components. Tests are coordinated with JUnit. Jemmy [6] is a library for automating Java GUI applications. It has an advanced abstraction tree for finding, examining and manipulating specific graphical components.

Abbot [7] is a very well developed extension to `java.awt.Robot`. An operator class is used to manipulate particular types of components via native events generated with Robot. For system testing, scenarios are recorded, assertions added, and scripts executed to create a test. Marathon [8] lets the tester record, edit and execute GUI tests via an embedded Python interface. The resulting Python code is very easy to understand and hence readable by the customer. Abbot and Marathon have as a goal the automation of system tests, and not simply the automation of GUI component tests. The difference between them and Haste is that they use the script recording approach to organizing system tests, where as Haste offers a programmatic framework for testing.

In our experience, the best toolkit to use for programmatically manipulating Java GUI components depends on the component. Haste lets the tester mix and match GUI component manipulation objects by hiding this implementation detail behind a common interface. Haste also includes a rudimentary toolkit for GUI component manipulation.

2 The Haste Environment

We call Haste an environment because it is less than a standalone tool and more than a testing approach. Haste consists of a framework for system testing, analogous to JUnit, useful design patterns, a toolkit for GUI automation, and utility classes to support automated system tests in an XP environment. The concepts in Haste are language independent; the first implementation of Haste is in Java.

Haste was created to extend our test-first development environment and testing style into the realm of system tests. The goal was to have a continuum of test suites from unit tests through system tests. Consistency across testing levels encourages the practice of programmers and testers working closely together, or in the case of small scale teams, having programmers do test-first development.

The most important design decision we made for Haste was to execute test code in the same address space as the application under test. This allowed for an internal, programmatic form of testing, rather than an external, scripted approach. This approach saves time and avoids the difficulty of exposing an application's state via an external interface [9]. System testing with Haste is philosophically similar to unit and integration testing, and hence in keeping with standard XP testing practice.

Haste has three main elements. The testing framework element consists of three key abstractions and several utility classes. The second element is a pattern for exposing the internal state of objects for making test assertions. We call this a Narcitecture. The Haste framework and Narcitecture are used for testing all types of applications. Testing applications with graphical user interfaces is possible with the third element of Haste: Pilots and Droid. Pilot interfaces simplify and standardize access to complex GUI components. A Pilot hides the actual GUI manipulation object used. The Droid class allows programmatic execution of an application via the generation of native input events. Droid is not required for Haste tests, and it may be replaced by other utilities for programmatically manipulating graphical components.

2.1 Haste Abstractions

The key abstractions of the Haste testing framework are the Story, Step, and StoryBook. The class relationships for these abstractions are illustrated in Figure 1.

Story. A Story corresponds to a test of a user story. Each Story consists of a set of ordered steps. Steps make JUnit assertions, and the failure of any assertion causes the execution of the Story to stop and be reported as a failure. Story differs from the JUnit TestCase upon which it is built in two important ways. First, the individual test steps are executed in a well-defined and intentional order, unlike test methods in a TestCase. Second, failure of any step causes failure of the Story. A Story stops executing at the first failure of a Step. The interface for Story is shown below:

```
public abstract class Story extends junit.framework.TestCase{
    protected abstract List steps();
    protected abstract boolean storySetup() throws Throwable;
    protected abstract void storyTearDown() throws Throwable;
}
```

Step. A Step is a relatively small, independent action within a Story. Steps are implemented as classes. This allows for easy sharing of common test steps between user stories. A Step consists of Java statements and JUnit assertions that exercise and validate some aspect of the application under test. For graphical applications, Pilots allow the code for Steps to be easily read and to correlate clearly with a user story elaboration. Step classes that are not shared between stories are typically implemented as Java inner classes. The interface for Step is shown below:

```
public abstract class Step extends junit.framework.Assert {
    public abstract void runStep() throws Throwable;
}
```

StoryBook. A StoryBook contains a collection of Story and/or StoryBook objects, and is used to organize and group story tests. Story test objects are run by a utility class called the JVMStoryRunner. This class allows each Story in a StoryBook to execute with a new instance of the application under test in its own private Java virtual machine. We find that the run-time penalty for this approach is more than made up for by eliminating potentially difficult-to-debug Story interaction due to indeterminate application state when a Story executes. The interface for StoryBook is shown below:

```
public abstract class StoryBook extends
    junit.framework.TestSuite {

    public void addStory(Story s) {}
    public void addStoryBook(StoryBook b) {}
    protected abstract void stories();
}
```

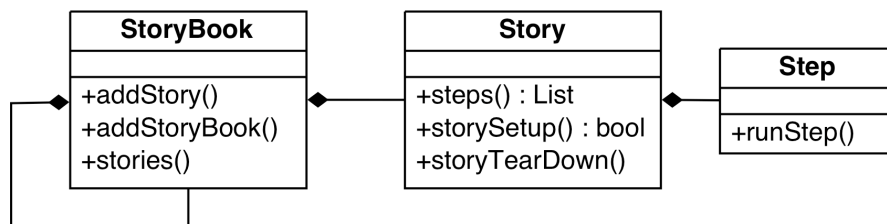


Figure 1. Class diagram of key abstractions in Haste system test framework

2.2 Narcitecture

Unit and integration testing generally involves manipulating an object via its interface and making assertions about the state of the object to detect failures. In Java, access to the internal state of an object is easily gained by placing the test class in the same package with the class under test. In C++ the friend modifier is used for the same effect. System tests are naturally higher-level tests than unit or integration tests, and therefore assertions need to be made on multiple objects. The state of the application, rather than simply the state of a single object, needs to be evaluated in a system test.

A Java class can be a member of only a single package, limiting the use of package level access to grant a test Story access to the internal state of the application. Our convention is to place stories in packages designed only for logically grouping stories. The Haste Narcitecture was created to solve the problem of getting access to the internal state of the application.

A Narcitecture is a set of classes created for a particular application that make the internal state of the application available to Story objects. A narc class is created to reveal the internal state of a particular source class. Narc classes live in a parallel package hierarchy to the application's source classes, just like unit test classes when using JUnit. The narc exploits its package level access to a source class and makes that state available to the Story via a public interface. Narc classes can be automatically created via reflection with the NarcGenerator utility.

2.3 Pilots and Droid

Pilot interfaces wrap the complex behavior of GUI components into simpler interfaces that, together with a specific GUI manipulation toolkit, allow for test step classes to be written at a more abstract level. Complex GUI components such as combo boxes, menu trees, and file dialogs require multiple actions to do something which is conceptually simple from the user story elaboration perspective. In addition, these types of components are often platform dependent, behaving quite differently between different Java look-and-feels or operating systems. An Abstract Factory pattern is used for constructing Pilot objects. The Pilot factory insures that all Pilot objects used in a test suite are consistent with the look-and-feel or platform being tested.

Each Pilot has a one-to-one relationship with a particular class of GUI component. Pilot methods are implemented by an object from a particular GUI testing toolkit. That object manipulates the underlying GUI component to achieve the action desired in the user story elaboration. For example, the following snippet of user story

... the user selects a file to load...

corresponds to test step code using a JComboBoxPilot Pilot as follows:

600000 Carl Erickson, Ralph Palmer, David Crosby, Michael Marsiglia, Micah Alles

```
PilotFactory factory = PilotFactory.getNewFactory();
JcomboBoxPilot boxPilot =
    factory.createJcomboBoxPilot(fileChooser);
boxPilot.setEventDelay(50);
boxPilot.clickItem(1);
```

The `clickItem()` method is approximately 40 lines of code using a Droid to manipulate the `fileChooser` combo box. The complexity wrapped by the Pilot makes the test step simpler.

The Haste Pilot interfaces decouple system test code from any specific GUI testing toolkit. Haste includes a utility class, Droid, which extends `java.awt.Robot`. Robot was created to automate GUI testing. Robot is used to generate native system input events such as mouse clicks and keyboard strokes. Droid also allows for control over event timing, and for synchronization of events on the Java event queue. By driving the GUI programmatically, Droid isolates the test code from the detailed appearance of the interface. Droid is similar in approach to JFCUnit [5] and Abbot [7].

An illustrative subset of Droid's interface is shown below. Droid offers a higher-level, more convenient interface for manipulating GUI components. For example, Robot provides `keyPress(int code)` and `mouseMove(int x, int y)`, while Droid builds on these to allow for typing a string or clicking a component.

```
public void typeString(String s);
public void clickComponent(Component c);
public void typeKeyShift(int k);
public static void waitOnEventQueue();
```

3 Example and Experience

Atomic Object develops custom software using XP practices. The first use of Haste was for a contract project for Burke E. Porter Machinery. We were contracted to implement the client side of a next generation, customer configurable, dynamic vehicle test (DVT) machine.

3.1 CCRT Application

Burke Porter DVTs perform the final evaluation of new vehicles at the end of an automotive assembly line, and are deployed throughout the world. The DVT has a real-time server implemented in C++ on QNX, and a platform independent, heavily multithreaded, fully internationalized Java client driving dual video displays.

With the lone exception of acceptance testing, development of the CCRT client application was done following standard XP practices. The CCRT application unit and integration test suite consists of 1,980 test methods for a source tree of 206 classes. The difficulty of automating acceptance tests for a complex application with a

graphical user interface stalled us on this XP practice. Facing the first field deployment of the CCRT application in the summer of 2002, we decided we could delay no longer and needed to solve this thorny problem. Haste was born of this need.

Nine months into the project and ready for the deployment of the version 1.0 beta of the application, the customer selected the most vital functionality from among the approximately 80 story cards in the project and we created system tests with Haste. From that time forward we adopted a concurrent development strategy for system tests. As of February 2003, the CCRT acceptance test suite consists of 24 story tests. The time to develop each Story has ranged from 30 minutes to 8 hours (pair-time).

3.2 Restricted Configuration Story

An example Story from the CCRT application involves operation of the application's configuration panel. The story card reads as follows:

*The configuration panel will only be accessible to users with sufficient privileges.
Users will authenticate themselves via a login and password.*

The elaboration of this story involved more details concerning special subpanels of configuration, a multi-level access control scheme, what should happen for non-authorized users, automatic de-authentication, etc. The Story class created for this user story was called `StoryRestrictedConfiguration`¹. The StoryBook named `ConfigurationStories` contains several related stories:

```
public class ConfigurationStories extends StoryBook {
    protected void stories() {
        addStory( new StoryRestrictedConfiguration() );
        ...
    }
}
```

The Story consists of the following steps in a total of 300 lines of Java, including comments:

```
public class StoryRestrictedConfiguration extends Story {
    Droid r2d2;
    protected boolean storySetUp() throws Throwable {
        r2d2 = new Droid();
        r2d2.setAutoDelay(50);
        return true;
    }
    protected void storyTearDown() throws Throwable {
        new StepStopApp().runStep();
    }
}
```

¹ Atomic Object uses a convention of naming all stories by the beginning word `Story`, and all test steps that are shared between stories with the beginning word `Step`. Haste does not enforce this convention.

800000 Carl Erickson, Ralph Palmer, David Crosby, Michael Marsiglia, Micah Alles

```
protected List steps() {
    List steps = new ArrayList();
    // app starts with known preferences file
    steps.add( new StepRestoreRtcProperties() );
    // shared step for starting the application
    steps.add( new StepStartApp() );
    // user enters the configuration screen
    steps.add( new SelectConfig() );
    // non-authenticated user is denied access
    steps.add( new NotLoggedIn() );
    // authenticated user is allowed access
    steps.add( new LoggedIn() );
    // only special user can access user config panel
    steps.add( new AccessUserAdmin() );
    // confirm auto logout when the user leaves
    steps.add( new AutoLogout() );

    return steps;
}
}
```

The source code for the AccessUserAdmin test Step is shown below:

```
class AccessUserAdmin extends Step {

    public void runStep() throws Throwable {
        String usersButton = "users";
        Controller controller = Main.controller;

        ConfigurationManager configManager =
            controller.getGUIManager().getConfigurationManager();
        ConfigurationManagerNarc configNarc =
            new ConfigurationManagerNarc(configManager);
        String currentButton =
            configNarc.getSelectedButton();

        assertTrue("Should not already be in 'users'",
            !usersButton.equals(currentButton));

        // enter configuration panel by clicking users button
        // login dialog should display with focus
        configNarc.clickConfigButton(usersButton);

        assertTrue("Users panel visible but not authenticated",
            !configNarc.getSelectedPanel().isVisible());
        assertTrue("Users button should be selected",
            usersButton.equals(configNarc.getSelectedButton()));
        // login with the standard login and password
        r2d2.typeString("bogus\tbogus\t ");
        assertTrue("Authenticated but users panel is not visible",
            configNarc.getSelectedPanel().isVisible());
    }
}
```

3.3 Customer perspective

Burke E. Porter Machinery specializes in embedded real-time vehicle test software designed to verify quality and performance capability of state-of-the-art advanced vehicle systems. This specialty of testing vehicle engine control units, power train components and subsystems during the OEM design and verification phases demands total flexibility and short iterative design cycles. We employ rapid control prototyping (RCP) or the practice of testing control software with a real system as our standard development approach to solve this challenge. When we looked for a partner to implement the GUI for our new real time system, initially we chose Atomic Object due to their utilization of XP and in particular, User Stories. We are in a niche business and a big concern of ours was the time required to bring an outside contractor up to speed on the requirements for our application. The User Story approach allowed us to quickly transfer our design concept to Atomic Object and avoid the effort of compiling a static specification doomed to become “shelfware”.

We quickly learned that the User Story approach was just one of the benefits to Atomic Object’s approach. Since we were able to communicate our design concept in discrete User Stories, we were able to prioritize and receive them according to individual or User Story unique deadlines. The Haste Story tests were initially seen more as a method to ensure a minimum level of quality of a component in isolation. It was incorrectly seen as a type of sorting process where “bad” components were stopped prior to release. We realized, however, that we were taking delivery not of just components but of working sub-systems. As Story tests validated each additional feature, our trust and acceptance of the software grew allowing us to confidently integrate it with our real time code and not introduce bugs.

Rapid, confident iteration had many benefits for us. As embedded programming specialists, the value add we deliver to our customers is in the “black box”. Traditionally, this is developed first and the user interface is developed second. This often creates the situation where the end user has to be a software engineer to understand the true state of the system. With working GUI sub-systems however, we were able to get customer feedback very early on allowing us to judge customer satisfaction and mitigate risk. Poorly understood customer requirements, desires, usability or even potential feature enhancements that could or would not have even been conceived of were made obvious by these early builds. This allowed us to adapt to these issues and take appropriate action without threat to schedule.

Lastly, the XP development process with Haste story tests provided us with a sales tool at the earliest possible point in the software development cycle. The ability to show working systems of any size is a valuable tool to demonstrate that our solutions are real and not vaporware.

4 Conclusion

Haste was built and first used in the middle of an 18 month XP project. For the CCRT application, Haste system tests satisfied the need for quality assurance in a release process, for customer buyoff, and for code maintenance via regression testing. During the release of the final 1.0 version of the application, the only bug reported in the field occurred in a related application for which no system tests had been written. We believe that the obvious system tests would have detected this bug. Haste is now used in our test-first development fashion, with system test development an integral part of working on new user stories.

Haste is available under the LGPL license. The home on the web for Haste is <http://atomicobject.com/haste>. The Haste project is maintained on Source Forge, and includes source code, test code, documentation, and sample applications.

Acknowledgements

Large projects and good ideas are rarely the sole work of the authors of a paper. We would like to acknowledge and thank the other team members of Atomic Object, namely Bill Bereza, Karlin Fox, Jeff Martin, Chris TenHarmsel, and Daniel Estrada for hard work and good ideas along the way. Burke Porter engineers, particularly the CCRT core team of Kevin Hykin, Tim Bochenek, and Brian Meinke also played an important role in the development of Haste. Paul Jorgensen made insightful comments on an early draft of the paper. Reviewer comments were helpful and much appreciated.

References

1. Jorgensen, P.: Software Testing, A Craftsman's Approach, 2nd Edition. CRC Press (2002)
2. Gold, R.: HTTPUnit: <http://httpunit.sourceforge.net/>
3. Kitiyakara, N.: Acceptance Testing HTML, Extreme Programming and Agile Methods – XP/Agile Universe 2002, Wells, D., Williams, L., Editors (2002)
4. jWebUnit: <http://jwebunit.sourceforge.net/>
5. JFCUnit: <http://jfcunit.sourceforge.net/>
6. Jemmy: <http://jemmy.netbeans.org/>
7. Abbot: <http://abbot.sourceforge.net/>
8. Marathon: <http://marathonman.sourceforge.net/>
9. Marick, B., Pettichord, B.: Workshop on agile acceptance tests, XP Universe 2002, Chicago IL, http://www.pettichord.com/agile_workshop.html
10. Beck, K.: Extreme Programming Explained, Addison Wesley (2000)
11. Canoo WebTest: <http://webtest.canoo.com/webtest>