2009 33rd Annual IEEE International Computer Software and Applications Conference

Photon: A Domain-specific Language for Testing Converged Applications

Anne Miller
Verizon Communications
anne.miller@verizon.com

Balaji Kumar
Verizon Communications
balaji.kumar@verizon.com

Anukul Singhal
Verizon Communications
anukul.x.singhal@verizon.com

Abstract

Automated testing of converged applications can be complex, as it is rare for a single testing tool to provide a single solution for all access points which a given application supports. As such, testing teams often create customized testing frameworks, which integrate several different testing tools, and a myriad of programming languages and scripting tools. When an application's unique set of access points changes, or a new testing tool comes to market which offers a competitive advantage over existing test tools, the cost of updating these customized frameworks can be difficult to justify. This paper provides a solution to this problem by introducing "Photonese," a domain-specific language which testers can use to compose automation scripts which are independent of the test tool used for automation. In this way, the tester creates reusable testing assets in a framework which is reusable across multiple projects.

1. The problem with test automation

1.1. What is a test automation framework?

A test automation framework is made up of a set of tools which provide support for automated software testing. The main advantages of such a framework are productivity and efficiency, as fewer man-hours are required for design and implementation of a test automation framework than are required to execute regression test cases manually over the lifecycle of a product. The type of testing framework used will generally depend on the type of testing being done, the types of testing tools available, and the skill level of the testing group charged with creating the framework.

1.2. The problem with having different automation frameworks

To illustrate the problems which arise when creating a unique test automation framework for each product, let us consider an example in which a tester is attempting to automate a test case which involves both web user interface (UI) and a public switched telephone network (PSTN) endpoint.

Assume the system allows the user to save contacts in a web-based address book, and displays a list of outgoing call logs made from the telephone associated to this user's account. If the outgoing call is placed to a telephone number (TN) which is associated with one of the Address Book contacts, then the outgoing call log should display the contact's First and Last name.

A basic test case for this system might include a scenario where the user adds a contact to the web UI, and then places a call to that contact. The success criteria would include (a) was the contact successfully added to the address book, and (b) was the call received by the contact, and (c) does the outgoing call log properly reflect the Address Book contact's First and Last name.

In order to automate this test case, we would need two frameworks:

- Web automation framework: This framework would run an automation script to add the contact via the web UI, and verify that the contact was added successfully. Additionally, after the test call is made, the web automation framework must have a separate script to navigate to the outgoing call logs page, and verify that the call log shows the address book contact's First and Last Name appropriately.
- Telephony automation framework: This framework would execute an automation script to place a call to the contact' telephone number, and verify that the call was placed and received successfully.

In order to execute even this simple test case, we must use two automation frameworks, and check four result conditions in order to determine whether the test case passed or failed.

0730-3157/09 $25.00 © 2009 IEEE
DOI 10.1109/COMPSAC.2009.143

Problem #1: Many automation frameworks operate in silos, leading to several test scripts needing to be run in a certain order, and many results needing to be verified in a specific order.

This is not to say that we could not integrate the two automation frameworks together – this can and has been done many times. However, this soon leads to additional problems:

Problem #2: As automation tools evolve, new automation frameworks are explored and adopted. When this occurs, the automation framework and test cases must be refactored.

Problem #3: For a large organization with different groups testing different cross-sections of devices, at various stages of new tool integration, the problem becomes even more complex.

These are the technical problems with automation frameworks, but there are also some organizational challenges. As automated test cases require fewer human resources to accomplish the same objectives of a manual tester, it is in the best interest of the project to automate test cases as early and as often as possible.

Due to the complexity and range of both individual project requirements, and the intricacies of each automation tool, it is common for organizations to have manual test engineers (responsible for test case development) and automation test engineers (with expertise in test automation tools). At some point in a project lifecycle, the manual tester and automation test engineer come together to automate the test cases. This interaction can be problematic, in that unless all product knowledge is transferred, there is a risk that the integrity of the initial manual test case may be lost to some degree upon automation.

Problem #4: Transition from manual to automated testing runs risk of diminishing integrity of test case.

These problems are solved by Photon.

1.3. What is Photon?

Photon is a unified automation framework, which defines a standard method of composing functional test cases, and supports custom implementations of these interfaces for any automation tool. Test cases developed in this framework are independent of the tools used for automation; such that test case refactoring is not needed should a new test tool be introduced.

To illustrate the advantage of Photon, consider the following diagram. A variety of third-party testing tools exist today, with each vendor generally implementing a different automation scripting language and domain-specific syntax (e.g. VBScript, TCL, proprietary languages).

Photon provides an adaptor layer, which translates Photon syntax to a particular tool's vendor-specific syntax, therefore allowing a test engineer to interact with a variety of testing tools with a single language. In addition, should a new tool become available, there is no need for a test engineer to refactor the test cases – he or she must simply implement a Photon Adaptor for the new tool.

1.4. Another automation scripting language?

Any team chartered with automating test cases will want to consider the investment spent in learning a new scripting language when evaluating a testing framework, as most testing tools support some scripting language. For the current implementation of Photon, the Ruby scripting language was chosen. As a result, a very basic knowledge of the Ruby programming language is required; however the selection of the Ruby scripting language was made with this consideration in mind, as it is:

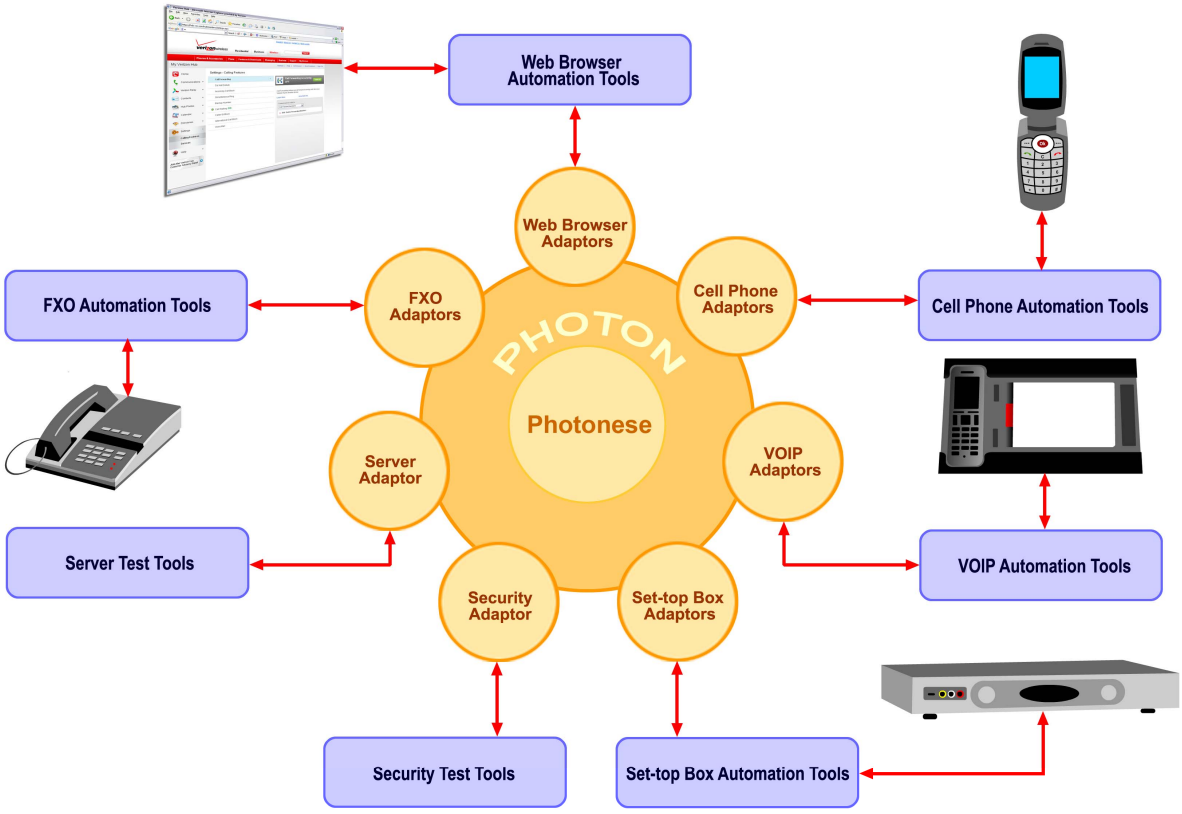

Figure 1. Photon adaptor layer

- Lightweight, interpreted
- Easy to learn
- Available on all major platforms
- Object-oriented
- Strong open source user community

The flexibility of the Ruby programming language also offers the tester and the testing group an open platform which can be modified to suit the testing organization's needs. In addition, the tester can create application-specific libraries, which allow abstraction of common application-specific actions, so as to reduce Testcase refactoring due to application evolution.

A tester who wishes to extend the Photon framework by integrating a new test tool and/or a new Resource type, would of course require a more in-depth knowledge of Ruby.

1.5. Photon Adaptor Layer

A tester need only learn the basics of Ruby, and "Photonese," a domain-specific language which allows the tester to execute certain common actions on different testing Resource objects. For example, the tester can execute "dialExternal(tn)" on a Telephone or "gotoURL(url)" on a Web Browser.

As Figure 1 illustrates, the tester does not need to learn any scripting language specific to any testing tool, as Photon will perform any translation required. The current implementation of Photon has found that the Ruby standard library provides the needed functionality to perform translation to any scripting language which a given testing tool may support. Photon currently translates to tools which use Visual Basic, TCL, Java, and simple shell scripting.

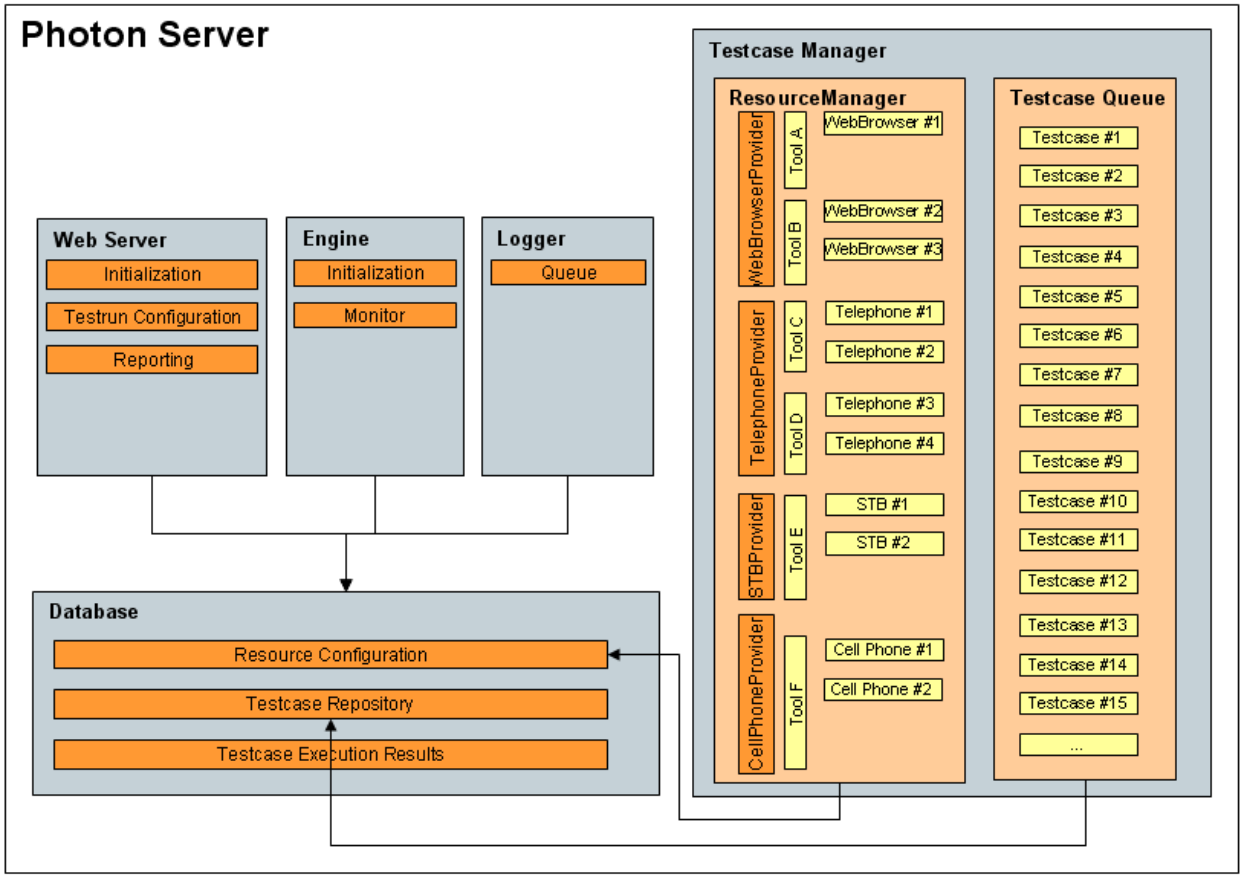

Figure 2. Photon architecture

2. Photon Architecture

The Photon Server runs as a stand-alone application on an individual tester's PC. As Figure 2 illustrates, Photon provides a web-based database-backed user interface, which allows the user to configure various testing tools, start the Photon Engine with a set of testing tools, and queue Testcases which are executed with a combination of testing tools required.

The Photon architecture consists of the following elements:

- **ResourceManager:** Manages Resources, which are provided by ResourceProviders, as configured at system startup.
- **ResourceProviders:** Each ResourceProvider instance corresponds to a particular testing tool. As illustrated by Tool A and Tool B in the figure, there can be multiple adaptors for a given ResourceProvider type.
- **Resources:** Resources are the objects which Testcases use to execute common actions which make up the test steps. Sample Resources illustrated in the figure include WebBrowser, Telephone, Set-top Box (STB), and Cell Phone.
- **TestcaseManager:** Queues Testcases which have been configured by user as Testruns, runs Testcases based on available Resources (per ResourceManager)
- **Logger:** Processes Log events which originate from the Photon framework, handling each Log according to system configuration
- **Server:** Launches Web Server, which supports launching of Engine, Testrun Configuration, and Logging.
- **Engine:** Launches Logger, loads ResourceManager, and launches TestcaseManager thread, monitors system, checks health of components, checks for newly configured Testruns

3. Writing Testcases

3.1. Testcase format

The Photon Testcase composer must create a separate class for each Testcase definition. As the sample Testcase definition in Figure 3 illustrates, each Testcase class must inherit from Photon::Testcase, and must define the following methods:

- **initializeTestcase:** Any initial setup, including assigning Resources in the resources Array argument to instance variables for ease of use within the runTestcase method
- **runTestcase:** This method should consist of a series of statements which execute common actions on the Resource(s) which the Testcase was passed in its initialize method.
- **reportTestcaseResults:** This method should be used to parse the results of the common actions in the runTestcase method, and uniquely define the success or failure of the Testcase.
- **class method getRequiredResourceMaps:** This method is called by the TestcaseManager in order to check out the Resources which are needed by the Testcase. Because each Testcase will require a unique combination of Resources, each class must define this method separately.
- **class method getRequiredParameters**: This method is called by the TestcaseManager in order to check what parameters are required by the Testcase. Because each Testcase will require a unique combination of Parameters, each class must define this method separately. Testcases which do not require any parameters are not required to define this method.

When composing test cases, the tester should consider that there is no guarantee that test cases will be executed in order. In addition, the tester should make no assumption that a given resource object passed to a test case will either (a) be the same for each execution of the test case or (b) be the same for test cases executed successively.

3.2. Application-specific libraries

.

While writing a set of Testcases for an application, the user will inevitably come across sequences of test steps which are repeated. As such, the user may find it convenient to abstract these application-specific actions, so that the same code is not repeated in several Testcases. This abstraction will also aid the tester as the application inevitably evolves over the course of testing.

For example, when testing a web-based application which requires authentication, the user might want to abstract the Login and Logout functionality. Similarly, the user may find it useful to abstract certain constants, such as Area (when cropping screenshot images), Point (when clicking on a particular X-Y coordinate), etc.

The user can define such convenience methods and constants in a separate library file, and simply import it into each Testcase file. The Testcase class provides a load_library method which allows the user to do this.

3.3. Application Evolution

A well designed library file will also help to minimize the impact of Testcase modification due to application evolution. Using the example of the Login and Logout abstraction, should the user interface components involved in these functions change due to user interface design change, the tester need only update the application library file, rather than updating every Testcase which logs in and out of the web based user interface.

4. Conclusion

Implementation of the Photon automation framework has been invaluable in automated testing of converged applications. Photon offers a common interface to the tester for a variety of testing tools, thus enabling the tester to automate Testcases involving several different testing tools.

In addition, because the tester can write Photon Testcases in a domain-specific language which is independent of the tools used for automation, reuse of the testing asset is possible even when the testing tools change.

Finally, because the automation framework is not application-specific, it is reusable across multiple projects, and can provide the user with a level of abstraction so as to minimize Testcase refactoring due to application evolution.

```ruby
class CallLogsTestcase < Photon::Testcase

  def CallLogsTestcase.getRequiredResourceMaps
    rmaps = [Photon::ResourceMap.new({:type => :account}), Photon::ResourceMap.new({:type => :web_browser}),
         Photon::ResourceMap.new({:type => :telephone})]
    return rmaps
  end

  def CallLogsTestcase.getRequiredParameters
    return [:callFromExpected, :callToTN]
  end

  def initializeTestcase
    @account = @resources[0]
    @webBrowser = @resources[1]
    @telephone = @resources[2]
    Photon::Testcase.load_library(__FILE__, "calllog_lib.rb")
  end

  def runTestcase
    @telephone.dialExternal(@parameters[:callToTN])
    CallLogLib::Application.login(@webBrowser, @account, false)
    @webBrowser.link(:id => "CALLSANDMESSAGESLnk").click
    Photon::Testcase.assert(
      @webBrowser.table(:column_header => "Status").getCellValue(5,3) == @parameters[:callFromExpected],
      "Call from does not match expected: #{@parameters[:callFromExpected]}")
  end

  def reportTestcaseResults
      Photon::TestcaseResultLog.new(@testcaseID, :pass, :pass).log
  end

end
```

Figure 3. Sample testcase