March 2010

A Comprehensive Web Deployment Strategy for a Component Based Java Framework

Ben F. Anderson
Worcester Polytechnic Institute

Stephen Franceschelli
Worcester Polytechnic Institute

Follow this and additional works at: https://digitalcommons.wpi.edu/mqp-all

Repository Citation
A COMPREHENSIVE WEB DEPLOYMENT STRATEGY FOR A COMPONENT BASED
JAVA FRAMEWORK

A Major Qualifying Project Report
Submitted to the Faculty
of
WORCESTER POLYTECHNIC INSTITUTE
in partial fulfillment of the requirements for the
Degree of Bachelor of Science
By

___________________________
Ben F. Anderson

___________________________
Stephen Franceschelli

9 March 2010

____________________________________
Professor George T. Heineman, Major Advisor
# Table of Contents

Abstract........................................................................................................................................... 4

1 Introduction..................................................................................................................................... 5
   1.1 Project Requirements .................................................................................................................. 6

2 Background ..................................................................................................................................... 8
   2.1 CompUnit................................................................................................................................... 8
   2.2 Deployment Technology Research ............................................................................................... 12
      2.2.1 JavaScript ............................................................................................................................... 13
      2.2.2 Servlets .................................................................................................................................... 14
      2.2.3 Java WebStart ......................................................................................................................... 14
      2.2.4 Java Applets ............................................................................................................................ 15

Methodology/Technology ................................................................................................................ 17

3 Design/Project Design.................................................................................................................. 20
   3.1 Manual Construction of Web-Hosted CompUnit Application ....................................................... 20
   3.2 Design of System (Batch) ............................................................................................................ 25
   3.3 Design of System (GUI) ................................................................................................................ 25
   3.4 SlingShot Scenario ....................................................................................................................... 26
   3.5 UML Diagram .............................................................................................................................. 28
   3.6 Sample Execution ......................................................................................................................... 28

4 Evaluation .................................................................................................................................... 32
   4.1 Unit Testing ................................................................................................................................. 32
   4.2 Statistics ..................................................................................................................................... 33
4.3 Limitations .......................................................................................................................... 34
5 Future Work/Conclusion ........................................................................................................ 35
6 Self Evaluation ........................................................................................................................ 37
7 Works Cited............................................................................................................................... 38
Abstract

It can be complex to deploy to the Internet a Java application requiring a framework that is not part of the Java Runtime Environment (JRE). Developers must modify their code to make it conform to the Java applet model and construct specific HTML tags. In addition, the required framework and code files must be uploaded to a web server. This project seeks a way to simplify this process. Specifically, this project enhanced the Java-based CompUnit component framework, which was developed by Professor Heineman of Worcester Polytechnic Institute, and created code that enables the deployment of a CompUnit application to the Internet with only minimal modifications needed by the programmer. The result is the Slingshot utility, which enables programmers to deploy their CompUnit applications to the Internet. Slingshot is now included as part of the CompUnit framework.
1 Introduction

A new paradigm shift in computing has led us to a software economy dominated by the Internet. Users want to access their applications from anywhere. Thus, many applications are now being hosted on the Internet for easier accessibility. Many technologies such as Java applets and Java WebStart are utilized to host and deploy applications on the Internet; however, there are limitations that developers face when deploying these applications. The inherent nature of the Internet is that Web-based client applications within a Web Browser have no client-side file system access, which is a significant limitation.

It can be complex to deploy to the Internet a Java application that requires a framework that is not part of the Java Runtime Environment.\(^1\) Developers must first modify their code to make it conform to the Java applet model and construct specific HTML tags. In addition, the required framework and code files must be uploaded to a web server. Writing large applications from scratch can be a very daunting task, so programmers often incorporate frameworks when designing large systems. This reduces the overall complexity of the initial problem and provides a reliable foundation for the application.

This project sought a way to simplify this process. Specifically, this project enhanced the Java-based CompUnit component framework, which was developed by Professor Heineman of Worcester Polytechnic Institute, and created code that enables the deployment of a CompUnit application to the Internet with only minimal modifications needed by the programmer. The

\(^1\) The Java Runtime Environment (JRE) consists of the Java Virtual Machine (JVM) in addition to other underlying and related technologies. The JRE is commonly preinstalled on most computers. The JVM “enables a set of computer software programs and data structures to use a virtual machine model for the execution of other computer programs and scripts. The model used by a JVM accepts a form of computer intermediate language commonly referred to as Java bytecode. This language conceptually represents the instruction set of a stack-oriented, capability architecture.” [http://en.wikipedia.org/wiki/Java_Virtual_Machine]
The resulting Slingshot utility enables programmers to easily deploy their CompUnit applications to the Internet and is now included as part of the CompUnit framework.

The challenge we faced in creating Slingshot was how to “re-host” the CompUnit framework so that it would enable the execution of previously stand-alone desktop applications, assembled from CompUnit components, to be run-able from within a Browser. The goal was to design an automated process for deploying suitable CompUnit applications to the web, which users can subsequently access and run. Slingshot accomplishes the original goal we had in mind, as it can take any component that conforms to the Java applet model, and deploy it to the Internet where it can be loaded and run within a Web Browser as a normal Java applet.

1.1 Project Requirements

The goal was to design an automated process for deploying “suitable” CompUnit applications to the Internet, which users could subsequently access and run without any other requirements. This necessitated writing a utility application for the CompUnit framework. This utility application has been named Slingshot, and can be run by programmers to deploy their applications based on the CompUnit framework to the Internet.

With the CompUnit framework, components can be packaged along with resources, such as images and sound files. These resources need to be accessible from within the web-deployed application, so they must be packaged and included with it. Additional complications arise when an application has external dependencies, such as the need for file system access. If an application that needs these dependencies is deployed to the Internet, it will not run properly. While the CompUnit framework does not have these external dependency issues, we wanted to ensure that our final Slingshot program could handle such issues for applications that
programmers may write for CompUnit. Handling of external dependency issues could help flag areas of CompUnit applications that would need to be rewritten to conform to the Java applet model. We felt this was critical because if an application cannot be rewritten to conform to the model, it is unsuitable for Internet deployment.

Another goal of this project was to ensure that after the automated process was run, the resulting “deployed” application could be hosted and immediately run within a Web Browser as a standard Java applet. We did not want the user to be responsible for hand-writing the HTML fragments necessary to embed a Java applet within an HTML web page. Additionally, we wanted to develop code to deploy the generated HTML file and corresponding code files to a remote machine, typically a Web Server, using existing file transfer protocols such as the File Transfer Protocol (FTP).
2 Background

The Java™ language was designed to enable software developers to write code that can be executed on any hardware platform with a working Java Virtual Machine (JVM). Starting in the late 1990’s, the JVM also supported running Java applications within Web Browsers. In the original model, a Java applet contains the code that is rendered within the browser and which controls the user interaction via keyboard and mouse. By convention, these applets were initially small code segments downloaded from the Web Server. They were executed within a protected “sandbox” inside the client Web Browser to prevent unauthorized access to the file system. However, as more complex applications appeared over time, the applet model has been unable to fully support these application needs. One particular instance that highlights this weakness is when a Java applet is constructed using third party code frameworks that are not part of the built-in Java Runtime Environment (JRE). If the required code frameworks aren’t installed on the computer before running the applet, it will not run. Developers have tried to solve this problem by packaging the required third party frameworks as part of the applet, but this has not been an ideal solution as the size of some frameworks can be prohibitively large to be loaded as an applet. The larger the size of an applet, the longer the end-user will need to wait for the applet to load and run. During the loading period before an applet is executed, a Web Browser can appear as if it has frozen. If this period is too long, a user may give up and close their Browser, as they may have assumed the applet doesn’t work properly.

2.1 CompUnit

CompUnit is a framework for building applications that conform to the component based software engineering cycle. The CompUnit framework is written in Java, is built from components itself, is open source and does not require external dependencies such as additional
underlying frameworks or resources. A CompUnit application is composed of one or more programmer-written components in addition to the standard CompUnit framework components. The CompUnit framework begins executing an application with a programmer-written component that has been flagged as the “main” component. Components utilize interfaces to communicate with other components, and are used when components want to connect to other components to exchange data. A component may have both provided and required interfaces. A provided interface means that the component implements the specified interface and provides functionality that is declared by the interface. Required interfaces are interfaces needed by components in order for them to run and utilize functionality of another component that provides the required interface. When a CompUnit application is run, each component must be provided with the required interfaces and must be “connected” to them before it can successfully run.

The first version of the CompUnit framework accessed resources by performing operations on the local hard drive at the file system level and is shown as follows:
Initially this was sufficient as CompUnit was still developing, and was not yet targeted at the Internet audience. As time went on, CompUnit’s developer, Professor Heineman, saw ways that CompUnit might be used in the future to distribute and run applications on the Internet. To facilitate this future need for Internet capabilities, subsequent versions of the CompUnit framework were restructured, changing the way resources were retrieved. Instead of dealing directly with the disk, resources are now accessed generically through the use of a ResourceRetriever component, and the IResourceRetriever interface. This was a big change and an excellent decision, as it allowed for additions such as this project’s utility application, Slingshot, which can easily deploy a CompUnit application to the Internet. As a result, when
resources such as graphics or sounds are needed by a component, a generic request is made to the ResourceRetriever component through the provided IResourceRetriever interface. This generic request is translated to properly access the requested resource with respect to how, or where, the application is running. If a CompUnit application is running as a desktop application, a resource request will be translated into direct file system calls. If a CompUnit application is running as an applet, the request will be translated into URL calls. This enables a programmer to write code once, and have it work in multiple locations without any other modifications. The new layout is shown as follows:
During the initial phase of investigation of deployment methods, several technologies were found as possible candidates for use in the deployment of a CompUnit application to the Internet.
These candidates included JavaScript (client side within a browser), Servlets (server side), Java WebStart (client side, though outside of a browser) and Java applets (client side within a browser). Each of these candidates had plusses and minuses for the purpose of this project and these are discussed below in more detail. We kept in mind a problem we envisioned; that is, if a component relied on writing to, or reading from a file system, other than for resource loading, it might not be a suitable candidate for Internet deployment. Eventually, Java applets were decided upon because they offered the cleanest and most secure way for end-users to run CompUnit applications from the Internet.

2.2.1 JavaScript

JavaScript is mainly a client-side programming language, which means that the code executes on the users machine. However, for security reasons, it is unable to write or read files locally. Additionally, JavaScript cannot write to or read from files on a server without the help of a backend script written in a language such as Perl or PHP. The process to deploy a CompUnit component utilizing JavaScript would likely be long and complicated, if not outright impossible. It would require that the CompUnit component framework be completely rewritten in JavaScript. It would also require the ability to interface with the server side script to write and read data. In addition to these complications, component developers who wanted to deploy their application would need to rewrite their components in much the same manner as the CompUnit framework would need to be rewritten. JavaScript was never seriously considered as a deployment platform because of the numerous limitations and the monumental amount of work that would have been needed.
2.2.2 Servlets

Servlets, in combination with Java applets, were closely considered for this project since Java applets can easily interface with a Servlet, providing access to the server file system. This method could have been used, since almost any CompUnit application can easily be translated to work as a Java applet. Furthermore, file system operations could be translated as HTTP requests to interact with a Java Servlet. This method for deploying components was not chosen, however, because the requirements for the hosting server were too great. The decision was made that the CompUnit component developers should only have to meet minimal server requirements, and need as few code changes as possible to deploy their components.

2.2.3 Java WebStart

Java WebStart (JavaWS) was also considered as a technology for deployment of CompUnit components. JavaWS provides a platform-independent deployment technology that allows application deployment by having applications hosted on a web server. It was designed to be a secure and robust system to allow different magnitudes of programs.

There are many benefits to having an application hosted on a web server. One benefit is that users don’t have to install applications, since they run in the browser easily with ‘one click’ activation. Coupled with this point is that JavaWS is platform agnostic, meaning any OS or Browser can be used. When the application is run it is cached locally on the users’ machine, and during any subsequent runs, it will load almost instantaneously since it already exists on the computer. After executing the application again, JavaWS will check and download an update if a newer version is available. JavaWS functionality is included in the JRE and provides its management capabilities. JavaWS became available with JRE version 2.
For this project JavaWS has significant advantages over other methods since it can access the local file system. However it can only do so if developers code-sign their applications.\(^2\)

Although JavaWS was a feasible possibility for this project it was not chosen for a number of reasons. One of the main reasons JavaWS was vetoed as a possibility is due to the fact that JavaWS applications may be self-signed.\(^3\) Because developers can self-sign their code, the whole purpose of code-signing is defeated. By allowing for self-signing, the security of local file system access can be compromised. The other reason we chose not to use JavaWS was that we decided that CompUnit applications deployed to the Internet should be as easy as possible to use, in addition to leaving minimal leftover traces on the client-side system after unloading an application. JavaWS can leave installed files that require cleaning up or deletion if a user is not going to run an application again.

### 2.2.4 Java Applets

The technology and method chosen to deploy CompUnit components to the Internet was based on the Java applet model. We chose this method because it is easy for a programmer to utilize, and offers the cleanest and most secure way for end-users to run applications from the Internet.

Java applets are generally small applications that have a certain specific task. They are often run within a larger program such as a Web Browser. The most common use of Java applets is to provide features to a web page that standard HTML cannot, such as interactive graphical content.

---

\(^{2}\) “Code signing is the process of digitally signing executables and scripts to confirm the software author and guarantee that the code has not been altered or corrupted since it was signed by use of a cryptographic hash.”  [http://en.wikipedia.org/wiki/Code_signing]

\(^{3}\) “In cryptography and computer security, a self-signed certificate is an identity certificate that is signed by its own creator. That is, the person that created the certificate also signed off on its legitimacy.”  [http://en.wikipedia.org/wiki/Self-signed_certificate]
Java applets must conform to the Browser Sandbox model that is imposed on them. The main premise of the Java Browser Sandbox model is to have an environment in which a program can run with restricted access to sensitive resources. The sandbox is responsible for protecting and limiting access to these resources. It has several built in security measures. First is a security manager whose sole purpose is to evaluate the code to be run. It looks for dangerous or malicious operations that are about to be performed, and has the power to accept or deny the running operation. There is also a byte code verifier that verifies code from unknown sources before it is allowed to run. In addition, it includes the applet class loader, which checks the runtime environment classes to make sure they haven’t been replaced.

The Browser Sandbox model can protect a wide range of computer resources including RAM, the keyboard or screen, or even the local file system. There are many levels of access that the sandbox can impose. Sometimes a program can have a wide range of access with minimal limitations, or it may be limited to certain things, like a specified maximum amount of RAM. Since the sandbox model has certain things it does and doesn’t allow, there are special guidelines that must be followed when creating applets. These guidelines limit the programmer to operations that conform to the sandbox model, since an applet may crash or not run if a certain operation is tried, such as accessing the file system. Any programs that run as an applet are run in the ‘sandbox’. Programs run at the command line can be optionally run inside or out.
Methodology/Technology

The initial task we had to accomplish for this project was identifying which technology to use. There were many possibilities, but in the end we decided to utilize the Java applet model to build off of the Java Sandbox model. While this still had its limitations, we accepted the fact that there was no perfect solution.

When utilizing Java applets as a deployment platform for applications, it is necessary to make any needed code modifications to components being deployed so that they conform to the Java applet model. In addition to possible modifications, a simple HTML file needs to be created and must include the `<applet>` tag, which contains the necessary information required to launch the applet in the web page. We created a simple CompUnit application to facilitate testing this process. The application, HelloWorld, is basic: it opens a window that shows some text. With HelloWorld, we did not need to make any code modifications to enable it to run as an applet because it already conformed to the Java applet model. We did, however, need to hand craft the HTML file that enables HelloWorld to run as an applet. This HTML file is shown below.

```
<html>
<head><title></title></head>
<body>
<applet code="org.compunit.FoundationApplet.class"
archive="/foundation/foundation.jar,/metadata/hello.World/1.1/code.jar"
height="150" width="300">
<param name="-ca" value="/applications/world.ca">
</applet>
</body>
</html>
```

Figure 3, sample HTML file with applet tag
Once we decided to use the Java applet deployment method, we immediately recognized the need for a tool to simplify the whole deployment process. For this reason we created *Slingshot*, which helps by creating a working HTML file based on a Component Assembly (CA) file, and then by uploading the correct files to a web server.

When working on the Slingshot application, it was much more practical for us to work separately and code on our own. In order to do this, we first had to design our system. We decided it would be mutually beneficial and more productive for us to split up the program into two main parts: the Graphical User Interface (GUI), and the back-end batch processor. We invested some initial time to design our system into these two components so that we could work on different things simultaneously. Once this was done, we each coded one part of it. We coded and tested our respective pieces separately, ensuring that each of our parts at least partially functioned. As the project progressed we worked together to connect each of our parts and test them as one.

The initial time that was spent designing our components to work together paid off, since the process of connecting our components was fairly seamless. A few defects had to be worked out, but within a reasonable amount of time we were able to get the program fully working.

To transfer the files to the web server, we needed to use File Transfer Protocol (FTP) and required an FTP server to test with. Since there is a lot involved in writing an FTP library, we used a freely available third party resource. First we researched and found some possible FTP libraries. We tested each of them, and compared and contrasted their capabilities in order to choose a particular one. Each FTP library ended up having very similar functionalities, and all of them were able to do the basic transfer operations that we needed, although some offered

---

4 Credit for providing an FTP server to utilize for testing goes to Stephen Franceschelli, project team member.
different advanced functionalities. Our initial choice ended up not working properly when we tried a simple test with it, so we picked a second one, the edtFTPj/Free Java FTP library, which did work properly. After we had decided on the FTP library we were going to use, we wrote a wrapper class to encapsulate all of its functionalities. Doing this allowed us to continue with the rest of the program while testing the FTP, because we abstracted its functionality into an interface.

To integrate all of the programming we did into the CompUnit environment, we had to use several of the frameworks’ developer tools. First we had to use Packager to package the CA files we had created. Then we used Café, a GUI CompUnit utility application, to connect the necessary components, and finally Installer to install the program. These tools were used often because frequent changes to the Java code required us to update specific component files to test them.

Once the program was written and thoroughly tested, we integrated the Slingshot utility into the CompUnit Sourceforge repository. It is now an integrated and fully functional part of the framework and is available for component developers to use.
3 Design/Project Design

3.1 Manual Construction of Web-Hosted CompUnit Application

During the first phase of this project, we needed to manually perform every operation that we wanted our Slingshot program to do. To help achieve these testing operations, we created a sample CompUnit ‘HelloWorld’ application composed of a single component to test with. A detailed description of CompUnit applications, that uses our HelloWorld application as an example, is written in the following paragraphs.

Every CompUnit application has one or more Component Assembly (CA) files, which describe how each component is constructed. A CA file is created by Café and is done semi-automatically; a programmer will need to connect component interfaces together, but using the easy to understand GUI makes it almost trivial. A CA file is a standard Extensible Markup Language (XML) file; our HelloWorld CA file looks like this:

```xml
# world.ca
1  <?xml version="1.0" encoding="UTF-8" standalone="no"?>
2  <APPLICATION SCHEMA="1.0" name="world">
3      <PROPERTIES name="META">
4          <PROPERTY name="main" value="World"/>
5      </PROPERTIES>
6      <PROPERTIES name="CAFE">
7          <PROPERTY name="compUnit-x" value="624"/>
8          <PROPERTY name="compUnit-y" value="181"/>
9      </PROPERTIES>
10     <COMPONENTS>
11         <COMPONENT class="hello.World" name="World" version="1.1">
12             <PROPERTIES name="org.compunit.CafeCafeEditor">
13                 <PROPERTY name="x" value="414"/>
14                 <PROPERTY name="y" value="200"/>
15             </PROPERTIES>
16         </COMPONENT>
17     </COMPONENTS>
18     <CONNECTIONS>
19         <CONNECT destination="org.compunit.interfaces.IShutdown" 
20             source="World" target="CompUnit"/>
```
Line 1 is the opening tag of an XML file. Line 2 is the beginning of an application with the opening tag containing two properties, a ‘SCHEMA’ version number which pertains to the version of the CA file structure itself, and the name of the CA file, ‘world’. Line 3 is a properties opening tag for ‘META’ properties, and can contain one or more property tags. Line 4 is a ‘META’ property that defines the ‘main’ component to be ‘World’. Line 5 is the closing tag for the ‘META’ properties. Line 6 is the opening properties tag for ‘CAFÉ’ properties. Line 7 and 8 denote GUI positioning properties for the CompUnit framework within Café. Line 9 is the closing tag for ‘CAFÉ’ properties. Line 10 is the opening tag within which all the components in the application are listed; in this case, the HelloWorld application consists of a single component. Line 11 is the opening tag for the ‘World’ component. Also defined in this tag are a version number, and the main Java class of the component. Lines 12-15 are more layout properties for the position of the component in the GUI Café window. Lines 16 and 17 are closing tags for the component and the wrapper for all components. Line 18 is the opening tag for interface connections. Line 19 is the only interface connection for HelloWorld and has three properties; a source, a destination and a target. The destination is the name of the actual interface, while the source and target denote the connected components’ names. Line 20 is the closing tag for interface connections. Line 21 is the opening tag within which components that need to be activated are listed. Line 22 is an order for the ‘World’ component to be activated. Line 23 is the closing tag for component activations. Line 24 is the closing tag for the application.
The layout of the files that make up HelloWorld are as follows:

![Diagram of HelloWorld's file layout]

Figure 4, the layout of an installed CompUnit application on disk

The top-level folders in this picture are located within an “environment” directory, which is where our application, HelloWorld, is installed. The first top-level folder is the applications folder, which contains CA files pertaining to applications that are installed within this “environment”. With this application, HelloWorld, the application file is ‘world.ca’. The next top-level folder is metadata. It contains the code for components that have been installed to the “environment”. Within each of the component directories there are folders pertaining to different versions that may exist. In this specific example, HelloWorld has two installed versions. Within each version of a component, there are two important files, the ‘code.jar’ file and the Component Definition (CD) files. The code.jar file is an archive of the executable code, the compiled Java ‘.class’ file, for the component. The CD file defines the interfaces of the component and provides some other useful information, such as the component author, the file size and the release date. This layout differs slightly from the layout of an exported or deployed application. The layout of an exported/deployed application would look as follows:
Figure 5, the directory layout of an exported or deployed application

The main differences to note are the ‘foundation.jar’ and HTML files. The ‘foundation.jar’ file is the entire CompUnit framework code, without which an applet would not run. The other file is a simple HTML file, which contains code telling a Web Browser how to run the applet.

After programming the HelloWorld application, we needed to manually export the files we needed for deployment. These files are shown above in figure 5. We also needed to hand-create the HTML file to properly load the application as an applet. This file is shown below:

```html
<html>
<head><title></title></head>
<body>
<applet code="org.compunit.FoundationApplet.class"
  archive="/foundation/foundation.jar,/metadata/hello.World/1.1/code.jar"
  height="150" width="300">
  <param name="-ca" value="/applications/world.ca">
</applet>
</body>
</html>
```

Figure 6, sample HTML file with applet tag
This is the simplest possible HTML file for an application/applet. The main thing we were concerned with was the `<applet>` tag. The `<applet>` tag, shown above in figure 6, has some important features. The first of these are the ‘code’ and ‘archive’ properties. The ‘code’ property has a value that pertains to the CompUnit framework Java class for execution as an applet. The ‘archive’ property contains comma-separated values that tell a Web Browser which code archives to look in for executable code files. There are also height and width properties that tell a Web Browser how big the applet is within a Browser window. Another important feature is the `<param>` tag. This is a parameter for the applet. The parameter in this case is a ‘-ca’ parameter. It has a value that is the relative path to the ‘world.ca’ file. The CompUnit framework reads this parameter when it is executed. It specifies to CompUnit the location for the CA file of the application to be run. An example of HelloWorld being executed as an applet within a Web Browser is shown below (it spawns its own window):

![HelloWorld applet window](image)
3.2 Design of System (Batch)

Our program, *Slingshot*, consists of two main components, GUI and Batch. The Batch component processes a CA file to determine what resources need to be transferred. It does the actual processing of the CA file. The Batch component takes the file and parses it to get the necessary information. Once this is done, Batch generates an applet tag and writes it to a temporary HTML file. Then it returns a list of files, including the HTML file, to the GUI so that the files can be moved.

![Diagram showing the required and provided interfaces of the Batch component.]

Figure 8, Batch’s required and provided interfaces

3.3 Design of System (GUI)

The GUI component is the driver for the program and deals with user input. The GUI portion is used to get information from the user. This includes the location that the files should be written to (the local file system or the FTP server), and if required, the FTP login information. It guides the user through a step-by-step process of gathering the required information and configuring the desired options. Once the list of files is returned from the batch component, the GUI either
moves them to a local folder, or transfers them to a server via FTP.

![Diagram of Slingshot's required interfaces](image)

**Figure 9.** Slingshot’s required interfaces

### 3.4 SlingShot Scenario

The following scenario describes the system behavior.

1. User installs their application with CompUnit installer.ca.
2. User starts SlingShot application in a similar manner to FoundationGUI.
3. Initial window comes up with a small intro text and a drop-down selection for the application being deployed.
4. User selects which application (and probably which version) to deploy, and then clicks next.
5. Either a popup box, or the next screen, will display an alert message asking if the user would like to run a quick code review to identify possible problem areas. (E.g. no file system access.)
6. If the user selects to perform a code review, a report will be generated identifying which files and lines have potential issues. Suggestions might be made as to how the problems can be fixed.
7. If no issues are found, or the user did not run a code review, the user can continue. If there are errors, the user can either quit and fix them, or can choose to ignore them. If the
user chooses to ignore them, a warning will appear. The user will also be warned if no
code review was performed, and a recommendation will appear prodding the user to do
so.

8. The next screen presents a few options for packaging/deployment. The user will select
an option to export locally or remotely, as well as the option to generate a template
index.htm file or merely the needed applet tag. (This could be a .txt file, or just presented
in a copy-able format at the final confirmation window.)

9. If the user selects to export remotely, the next window will have radio buttons to select
from FTP, SFTP, SCP or another way to upload. Depending on which radio button is
selected, the relevant form fields will enable (UN-grey out). The user will enter the
relevant information of username, password, server address, server path and server port.
Upon clicking next, Slingshot will attempt to connect to the server. If testing the
connection/authentication is unsuccessful, the user will be prompted to double-check the
settings.

10. If exported locally, the user is asked for a destination folder with a normal file browser
window.

11. After the files are uploaded or exported, the user is prompted to test the applet/URL/file-
path in their browser. If the index.htm page was generated, the URL or file-path will be
‘intelligently’ guessed and provided to the user. Otherwise the user will be prompted by
some short instructions with how to go about inserting the applet tag into their HTML
file.

12. At the final window the user will be given a report of the actions performed and files
exported or uploaded.
3.5 UML Diagram

![UML Diagram](image)

Figure 10, UML diagram showing the layout of Slingshot

3.6 Sample Execution

Once deployed, the main CompUnit framework begins execution as a Java applet and instantiates the main component for a given application via the custom `<param>` tag. This hybrid component/applet is able to do anything that the original CompUnit component could do, with the exception of local file system access. The full process of exporting an application with Slingshot is shown as follows.
Figure 11, the CA file-selection screen

Figure 12, the export type-selection screen
Figure 13, the local export folder-selection screen

Figure 14, the FTP export information-entry screen
Figure 15, the final confirmation screen

After deployment, the directory structure of the deployed files is as shown in figure 6 on page 23.
4 Evaluation

To evaluate the quality of the produced software, we relied on both unit testing and integration testing as we now describe. We also used available code coverage tools to determine the extent to which the test cases were properly executing the underlying code system.

4.1 Unit Testing

When testing applications, many programmers follow a process known as unit testing. Unit testing allows a programmer to verify that code is acting according to its specification. JUnit is the main unit-testing framework used in Java, and is open source software. JUnit is linked with the code to test at compile time, and runs a set of tests, developed by the programmer, against the code. This testing allows programmers a way to test that the functionality they expect from their code is accurate. Unit testing is aimed more towards testing small parts of code, rather than the entire program itself. Unit testing also allows the programmer to get statistics, such as code coverage. This provides a programmer with accurate information about what code has been tested, so that they can verify that the majority of their code has been tested.

Our initial testing was very basic. We made test classes and methods to see if sections of code functioned properly. This ranged from checking return values to inputting values into the GUI window and seeing if the correct files were being exported or uploaded correctly. This was a way of determining if the necessary functionality was present and working how we had assumed it would without having formal test cases.

---

5 “JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development, and is one of a family of unit testing frameworks collectively known as xUnit that originated with SUit.” [http://en.wikipedia.org/wiki/Junit]
We also tested the code by loading in the sample “HelloWorld” application. This was used to give us our first full run of Slingshot to see if it could run from start to finish correctly.

After we connected each part we started more formal testing. First we designed unit tests that could run automatically. These were used to test things like simple accessor methods and other pieces of the code that could be tested individually. After this was done we created a test harness and were able to test the component sections of the code.

The testing was split up into two parts: automatic and manual testing. The automatic is mainly simple tests and the manual is used for things like FTP where we have to input information. The goal for testing was 80% code coverage using EclEmma.6

4.2 Statistics

The testing and code coverage statistics of Slingshot are as follows:

![Code Coverage Table]

Figure 16, the statistics of Slingshot’s code coverage after running all automated and manual tests

---

6 “EMMA is an open source toolkit for measuring and reporting Java code coverage. EMMA is distributed under the terms of Common Public License v1.0.” [http://en.wikipedia.org/wiki/EMMA_(code_coverage_tool)]
4.3 Limitations

Upon starting the project, we had to recognize and accept the limitations of what we were doing. There are certain actions that simply are simply not allowed in an applet within the Java Browser Sandbox. The main limitation is that the application cannot read or write files from the client’s file system. This means that when running in the sandbox the programs can only access files that are read only. In order to accommodate this, our program will be distributed with a document stating common limitations and methods, which will allow the user to get around them as best as possible.
5 Future Work/Conclusion

At the beginning of this project we knew we would be somewhat time limited, and this partly influenced our decision to go with Java applets. The process to modify a component to fit the Java applet model is by far the easiest and least complicated of the researched methods. If we had had more time, we may have chosen to go with JavaWS, since it would offer exactly what applets can, but could offer local file system access as well. Given local file system access, the range of components suitable for deployment would be more widely covered, though the level of security would decline from the applet level.

Another idea we only briefly looked into was that of developing either an NPAPI based browser plug-in, or a Firefox® browser extension. Such a plug-in or extension would contain the complete Java bytecode for the CompUnit framework. This would further simplify the deployment of components as Java applets, since developers would only need to upload their code to a website. This would slim down the size of files that need to be downloaded, as well as reducing the load time.

We believe there may be a way to store files on the local file system since plug-ins and extensions both tie into the browser at lower layers. If this is the case, this method could be superior to all the others. The plug-in would be the sandbox for what would basically be a Java applet (but called a CompUnit applet), and as such we could develop a method to store an object locally, much the same way Adobe® Flash® does. This would likely be a significant amount of work, but it is something we would have explored had we had more time available to complete this project.
At the beginning of this project we had one main goal, which was to develop a method to deploy CompUnit components to the web. We met one half of this goal, which was the development of Slingshot. Slingshot takes a modified, ready for deployment component, and deploys it, doing some behind the scenes work to configure an HTML file properly so the applet will load. The other half that Slingshot does not cover is the code modifications that might be needed to make an application conform to the applet model. Some components may require little to no modification to be deployment ready, but some may take significant modification. For any component that loads resources locally, each call to a resource would need to be modified to look on the web instead. Developing an application to help automate the component modification process would complete the solution to the original problem. This would require advanced techniques such as static code analysis and was not feasible for the scope of this project. However, we believe it would be worth exploring in a future project.
6 Self Evaluation

Our initial goal was to create a web deployment application that could transfer files to an FTP or save them locally for deployment. The important things we had to take into consideration were the implicit limitations of the project, and the time we had to do the project. Based on this we would say that we met our goals and were successful in doing what we set out to.
7 Works Cited

   http://www.ucertify.com/article/what-is-the-sandbox-model.html

   http://docstore.mik.ua/orelly/java-ent/security/ch01_02.htm

   http://www.webopedia.com/TERM/s/sandbox.html


   http://javascript.about.com/od/reference/a/cannot.htm

   http://java.sun.com/products/servlet/overview.html


   http://en.wikipedia.org/wiki/Java_Servlet


    http://en.wikipedia.org/wiki/Self-signed_certificate


    http://en.wikipedia.org/wiki/EMMA_%28code_coverage_tool%29