NetBeans to Eclipse GlassFish Project Converter

A Major Qualifying Project Report:

submitted to the faculty of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by:

Michael Tidd

March 7th, 2011

Approved:

___________________________
Professor Gary F. Pollice, Major Advisor

1. GlassFish
2. Eclipse
3. NetBeans
Abstract

Many developers utilizing the GlassFish™ Enterprise server use both the Eclipse™ and NetBeans® Interactive Development Environments. Prior to this project there was no way to import a GlassFish project that was created in NetBeans into Eclipse. Automating this conversion required an understanding of the NetBeans and Eclipse file structures. It also required an understanding of various XML parsing techniques. The encapsulation of converter within an Eclipse plug-in required an understanding of the Eclipse plug-in architecture and how it interacts with files within the plug-in bundle. The project resulted in the creation of both a stand-alone conversion tool and the integration of that tool in an Eclipse plug-in.
Acknowledgements

Ludovic Champenois- Oracle GlassFish Team

Vince Kraemer- Oracle GlassFish Team

Gary Pollice- Worcester Polytechnic Institute, Advisor
# Table of Contents

Abstract.............................................................................................................................. i
Acknowledgements.......................................................................................................... ii
Table of Contents............................................................................................................ i
Introduction......................................................................................................................... 1
Background.......................................................................................................................... 3
Methodology......................................................................................................................... 8
Results and Analysis ........................................................................................................... 17
Future Work and Conclusions ............................................................................................ 19
Glossary................................................................................................................................. 20
References............................................................................................................................. 21
**Introduction**

The goal of this project was to develop a means to convert a GlassFish™ project that had been created in NetBeans® into an Eclipse™ compatible project. NetBeans and Eclipse are Interactive Development Environments (IDEs) that assist developers in writing, building, and running code. The need for such a converter is essential to the success of collaborative development on a GlassFish project. Often developers use different IDEs than their coworkers, or may use different IDEs themselves, and by providing an easy way to import these projects into the Eclipse workspace, it allowed developers working on the same project to utilize both IDEs. The project was completed in three phases:

1. The manual conversion of a set of GlassFish projects, which were created in NetBeans, into Eclipse compatible projects.

2. The development of a stand-alone converter to automate the process.

3. Encapsulation of the converter within an Eclipse plug-in.

The manual conversion process revealed the basic translations that were necessary in converting projects from NetBeans to Eclipse. The process also revealed that simple projects could be run using the simple run command in Eclipse. It also revealed that more complicated projects required being built and run via the Ant® build tool.

The second phase involved the creation of a stand-alone converter that allowed users to automatically convert a NetBeans GlassFish project into an Eclipse compatible project, and then import it using the standard Eclipse importer. During this phase the
decision was made to execute the conversion utilizing a standard conversion, adding the necessary files within the NetBeans project rather than copying the project into the Eclipse workspace. The plug-in was designed in a way to allow for future extensions that could add functionality outside of the scope of this project.

The final phase involved the creation of an Eclipse plug-in. By integrating the converter into a plug-in increased the ease of use and the potential impact of the converter. Although this was not a mandatory phase, it was critical to the success of the converter. The plug-in was designed to run within the GlassFish Tools Bundle for Eclipse. The bundle contained the Eclipse Java EE IDE, GlassFish v3 and Java EE 6 pre-installed. In order to apply the best pairing with the bundle’s Eclipse environment the converter needed to exist as a plug-in. In order to have it run seamlessly with the other aspects of the bundle it had to be developed utilizing the Eclipse plug-in architecture. By designing the GlassFish Importer to work similar to the standard Eclipse Importer made its use intuitive to most developers familiar with Eclipse. The goal of the project was to provide developers with a simple means of converting a GlassFish project created in NetBeans into an Eclipse project, and was achieved through the development of the Eclipse plug-in.
**Background**

This major qualifying project was done in collaboration with GlassFish Application Server team at Sun Microsystems. Work was specifically done with Sun engineers Ludovic Champenois and Vince Kraemer. Sun is a subsidiary of Oracle Corporation and is both a hardware and software vendor. Sun is well known for its creation of the Java platform. The Java platform consists of the Java programming language, the Java Virtual Machine, and the Java Application Interfaces. Sun is also a sponsor of the NetBeans IDE.

Project GlassFish is an open source application server based on Java System Application Server Platform Edition 9. This project focuses on GlassFish version 3 that was designed to include the capabilities of Java Enterprise Edition 6 within a GlassFish server. GlassFish v3 incorporates many features such as Java Database Connectivity 4, Enterprise JavaBeans 3.1, Java Architecture for XML Binding and Java API for XML Web Services 2.2, and JavaDB 10.5.3.

The project involves the interaction between two interactive development environments: Eclipse and NetBeans. In order to create an Eclipse project from a NetBeans project, the project information structure needs to be understood. Figure 2.1 displays the first layer of the NetBeans project structure. Within the `nbproject` directory of any NetBeans project is a `project.xml` file as seen in Figure 2.2. The `project.xml` file contains project information such as the name, type, Ant build version, and any libraries required.
NetBeans projects are built and run using Ant, which is a Java-based build tool. A build tool is one that is able to complete the following ten steps:

1. Get the source.
2. Prepare the build area.
3. Configure the build.
4. Validate the source code.
5. Compile the source code.
6. Build the compiled code into libraries.
7. Validate the build.
8. Build the documentation.
9. Package the components.
10. Deploy the software.
Similar to NetBeans, Eclipse has its own file structure, however the Eclipse file structure contains a few more reference files. All Eclipse projects consist of the .project and .classpath files. These files can be seen within Figure 2.3. The .project file contains the basic project information. The .classpath file contains information on libraries that are utilized by the plug-in. The Eclipse file structure also contains a .settings subdirectory. The .settings contains much of the additional information in relation to a project’s type. The files within this directory will differ according to the project type.

![Figure 2.3: Eclipse Project Structure](image)

Eclipse is built on a layer-based framework. The base layer is Equinox, which represents the Platform Runtime layer. Equinox acts as the plug-in control for Eclipse. The base layer for the user interface is the Standard Widget Toolkit (SWT). SWT is a UI toolkit that provides a foundation for widgets built upon it to be platform independent. Its role is not to provide a rich UI but rather a foundation for which a rich interface can be
built and then utilized across different platforms and be OS independent. Built on top of the SWT is JFace. JFace provides the structure for rich interface components such as dialogs, preference inputs, and wizard frameworks among others. Finally built on top of JFace is the Workbench. The workbench provides the basic UI structure for Eclipse, responsible for the presentation and organization of the various tools used within the framework.

![Eclipse Plug-in Architecture](image)

**Figure 2.6: Eclipse Plug-in Architecture**

The Eclipse Framework also allows plug-ins to be developed which extend the core layers of the framework. A single plug-in can utilize multiple layers of this architecture, in order to provide integration with multiple extension points in the Eclipse Architecture. Within Eclipse a set of multiple plug-ins is typically called a feature. By bundling plug-ins within a feature, the extension can utilize those extensions points. For example, the plug-in created by this project could be encapsulated within the larger GlassFish plug-in to provide additional functionality by extending the workbench layer of the Eclipse
architecture. These plug-ins are tools compiled into Java Archive files coupled with a plugin.xml file which the Eclipse framework parses and integrates the tools incased the Jar file into the Eclipse framework. Plug-ins may also collaborate with other plug-ins. This relationship is considered a dependency. For example, for the converter to be distributed separately from the GlassFish plug-in it required that the GlassFish plug-in already be installed in order to run successfully thus forming a dependency.

Java is an object-oriented programming language that is designed to limit the system dependencies of languages. The five goals of the Java programming language are:

1. Simple, Object Oriented, and Familiar
2. Robust and Secure
3. Architecture Neutral and Portable
4. High Performance
5. Interpreted, Threaded, and Dynamic

It is designed to be “architecture neutral, portable, and dynamically adaptable.”

(java.sun.com)
Methodology

The task was approached with bi-weekly iterative development with three major deliverables. Iterative development is an alternative to the waterfall method; in which to project takes one continuous flow. The waterfall method does not backtrack from phase to phase but rather follows a linear progression. There is also the modified waterfall method, in which a review of previous phases occurs as new information is collected and insights gained. (Software Engineering p20) However, even the modified waterfall method, would not suffice as the development process for the entire project. As there we three core deliverables, each phase of the project required the knowledge derived from the previous phase. Thus detailed requirement gathering could not take place until the previous iteration had completed. The limitations of the waterfall and modified waterfall methods were not shared with another design process: the iterative design process (Software Development p83). The iterative design process requires a full cycle of the design process during each iteration.

Figure 3.1: Waterfall Development Process
The three deliverables of the project included producing manually converted samples, creating a stand-alone converter, and creating an Eclipse plug-in. The first phase, manually converting each sample, was intended to provide the background information needed to translate NetBeans project information to Eclipse project information. The knowledge of both the mappings and the complexity of the translation was critical the architectural design design of the converter. The process revealed the translations for both project name and context root. It also revealed two other critical factors: the conversion of projects that required others and how more complex projects could be run.

The second deliverable, a stand-alone converter, was intended to be proof-of-concept. The implementation of the stand-alone converter raised some design considerations. The first was regarding the conversion method to be utilized, of which there were two options: standard and copy. A standard converter modified the existing NetBeans project to make it Eclipse compatible, whereas the copy converter created a duplicate project in the Eclipse workspace. The final decision was to implement the converter utilizing a standard conversion process. The standard process was selected...
because it is easier to extend the functionality of a standard converter rather than limit the copy converter. By implementing the converter as a standard converter it allows the developer to work on the same project within both NetBeans and Eclipse, where as a copy conversion would have created two instances of the project, creating redundancy issues. Figure 3.3 displays the standard Eclipse plug-in, which in the lower section there is a check box for performing a copy import. However the default import, does not import the project from the current directory, but rather links it to the existing project directory. By implementing a standard converter our importer could duplicate the flow of the standard Eclipse importer.

![Projects:](image.png)

**Figure 3.3: Standard Eclipse Importer**

In order to establish better cohesion a Factory pattern was implemented. It creates the necessary Eclipse project files. A factory is a singleton object for creating other objects(Head First Design p160). The Factory pattern provides a means of creating different objects at runtime. For the converter, basic functionality was included within an abstract factory class name `FileFactory` that included the ability to get the name and the XML for a given project.
The `FileFactory` class forces any concrete factory to implement the create method which handles the actual copying of the files. The `dotClasspathFactory` directly implements this abstract factory, as it does not need to do any extra functionality. The `dotProjectFactory` and the `dotSettingsFactory` both extend this Abstract Factory pattern. The `dotSettingsFactory` adds the functionality to handle the translation of the context root where the application will launch on the server.

```java
@override
public void create() throws TransformerException, IOException,
    ParserConfigurationException, SAXException {

    //load the nb context root
    String contextRoot = getContextRoot(path);

    //copy the ecl file to the nb location
    //FileCopier.copyDir(url + ".settings", path+"/.settings");

    //edit the ecl project file
    File contextFile = new File(path+"/.settings/org.eclipse.wst.common.component");
    Document context = getXML(contextFile);
    NodeList nodes = context.getElementsByTagName("property");
    Node contextNode = nodes.item(1).getAttributes().item(1);
    contextNode.setTextContent(contextRoot.substring(1));

    nodes = context.getElementsByTagName("wb-module");
    contextNode = nodes.item(0).getAttributes().item(0);
    contextNode.setTextContent(getProjectName(path));

    //save the ecl project file
    Util.saveDoc(context, path+"/.settings/org.eclipse.wst.common.component");
}
```

Figure 3.4: `dotSettingsFactory` Create Method

The `dotProjectFactory` includes that ability to handle the translation of the project name from the Netbean’s `project.xml` to the Eclipse `.project` file.

```java
public void create() throws TransformerException, IOException,
    ParserConfigurationException, SAXException {
    //Gets the project name using a SAXish Parser
```
String projectName = getProjectName(path);
IPath location = Platform.getLocation();
File file = location.toFile();
File[] files = file.listFiles();
file = files[0];
files = file.listFiles();
//edit the ecl project file
File projectFile = new File(path + ".project");
Document project = getXML(projectFile);
NodeList nodes = project.getElementsByTagName("name");
nodes.item(0).setTextContent(projectName);
//save the ecl project file
Util.saveDoc(project, path + ".project");

Figure 3.5: dotProjectFactory Create Method

The prototype version of the converter contained a standard document XML parser. A document XML parser is easy for developers to implement as it handles all of the parsing itself. In order to retrieve a given node the developer merely needs to pass the parser a given node’s name. It provides a quick and simple way of reading from and writing to XML documents. However, the document parser is fairly inefficient in cases where only a small portion of information available needs to be extracted. This inefficiency lies in the fact that it parses the entire document. In a case when it only needs to parse a few nodes, it ends up wasting time and resources parsing nodes that will never be read. In the case of this converter only the project name and context root needed to be translated, and they were in separate XML files, so the document XML parser was very inefficient means of parsing.

Thus for the beta version of the stand-alone converter a SAX parser was implemented. SAX stands for the Simple API for XML, however the name may be deceiving. Rather than being simple with regards to development it is actually in reference to the simplicity on the system level. It only provides low-level functionality, and the actual parsing is the responsibility of the developer. However, by implementing the parsing yourself, only the nodes that need to be parsed are parsed. With regards to the
project.xml file only one node was parsed while utilizing a SAX parser, where as the
document parser was parsing 36.

SAX parsers are not always ideal though, and in this case utilizing a SAX parser may
have some implications with regards to maintaining the code. As the SAX parser does not
parse the entire document, if the layout or organization of the documents being parsed
changes, the parser may too break. The risk inherited by implementing a SAX parser is
probably worth it with regards to the projects.xml file, however it may not have
been for extracting the context root where the sun-web.xml file is fairly small and the
return on investment by using a SAX parser isn’t worth the risk of the plug-in failing in
the case that the file standard was changed.

The third deliverable, the plug-in, was the ideal solution, providing users with a
natural and intuitive way to import NetBeans GlassFish projects into Eclipse. With the
initial approach it was considered to be an optional phase only to be complete if time
permitted. This phase required the wrapping of the stand-alone converter within an
Eclipse plug-in. Included with the stand-alone code; the plug-in also contained manifest
files, images, and template files. All these files are to be bundled within a jar pictured in
Figure 3.6. This jar also contains an activator, which exposes certain resources within the
plug-in to the Eclipse IDE. The Glassfish plug-in was built as an extension to the Eclipse
Import Wizards.
Figure 3.6: Plug-in Jar Bundle

Figure 3.7 shows the plugin.xml file, which contains the basic information about the plug-in. It includes the extension point, the name of import category, name of the plug-in, and the primary class for the import plug-in. Each of these pieces of information has been highlighted in the figure.

```xml
<?xml version="1.0" encoding="UTF-8"?
<!eclipse version="3.4"?>
<plugin>

<extension
  point="org.eclipse.ui.importWizards">
  <category
    name="Glassfish"
    id="com.sun.importWizards.sampleCategory">
    <wizard
      name="Import Glassfish Project from Netbeans"
      icon="icons/images.jpeg"
      category="com.sun.importWizards.sampleCategory"
      class="com.sun.importWizards.ImportProjectWizard"
      id="com.sun.importWizards.ImportProjectWizard">
      <description>
        Import a file from the local file system into the workspace.
      </description>
    </wizard>
  </category>
</extension>

</plugin>
```

Figure 3.7: Plug-in.xml File

The project also requires a manifest.mf file, which is shown in Figure 3.8. This manifest file contains information such as the Activator class and any packages that need
to be exposed. This proved to be the main setback, as the standard Java file access cannot access files that are contained within a bundle. Rather these files need to be exposed by including them within the plug-ins exported packages. Only after exporting them can the template files be accessed through a URL object. A URL object is then able to connect to the plug-in and to access the template files.

The samples provided at the beginning of the project provided for a test-first development process, where test cases are generated before any code is written.
Figure 3.9: Test-First Development Process

This test-first process allowed for complications to be discovered quickly, allowing for discussions with the sponsors on whether or not various outliers were within the scope of the project. The development process also helped during phase transitions by revealing certain issues that arose during the iterations. For example, they helped during the transition from a document XML parser to a SAX XML parser, the test cases provided a means of double-checking whether or not utilizing a SAX parser would break the existing architecture and helped to predict what types of changes would affect the SAX parser.
Results and Analysis

All three of the deliverables were completed. At the beginning of the project the sponsors, provided a set of project that served two purposes, the first being for manual conversion and the second for providing a platform for test-first development. As expected the manual conversions helped to indetify the various translations. The prototype was based on what was discovered in this phase. The stand-alone converter accomplished the goal of providing a “proof of concept” converter. The prototype also provided a basis for several design decisions.

These design decisions were implemented for the final version of the converter. Based on the sponsors impressions of the prototype, it was completely refactored for the final version. The refactoring fixed issues with the code’s structure and style and helped to make the code clean and concise. Also, per another design decision, the functionality for projects requiring external projects was removed at this point. Based on that design decision, the number of applicable test projects was reduced for the scope of this project. The stand-alone converter was completed and committed to the open source converter project created by the GlassFish team. The converter project is a sub project of the GlassFish plug-ins project.

Once the refactoring of the stand-alone was completed the next phase was to develop a working Eclipse plug-in. It was more important to encapsulate the converter within a plug-in rather than add some of the missing functionality. Based on the priorities set forth by the sponsors, and the time available at the end of the project the plug-in was developed and finished. It provided a simple and intuitive way to import GlassFish
projects that had been created in NetBeans. When running it against the set of test projects it successfully converted and imported twenty out of the twenty-three sample projects. The Eclipse plug-in runs seamlessly that the developer importing the project is not aware that a conversion is occurring, but rather is under the impression that the project is just being imported. The final version of the Eclipse plug-in was also committed under the GlassFish converter project in the open-source repository provided by the GlassFish team.
**Future Work and Conclusions**

The NetBeans GlassFish Project Importer has many areas for future work. The importer was designed with specific design requirements, intended to speed up the process with the hopes of having time to implement a plug-in. Several possible features were left out in order to expedite this process.

The first of these features was the implementation of the extension to allow for copy conversion. This would require adding the copy step to conversion process, and also modifying the import screen on the plug-in. The second possible feature is the ability to import multiple projects. This would require the converter to parse through given directories and then allow the user to select which projects to import. This could work just like the current “import existing projects” in the Eclipse IDE.

This project utilized and demonstrated many software engineering techniques including iterative development, test-first development, and pattern-based design. The project delved into two forms of XML parsing, prototype development, and plug-in development. At the end of the process all goals were fulfilled, including the development of a “proof of concept” stand-alone converter and the creation an easy to use Eclipse plug-in to support the conversion and import. The project can be accessed at [http://glassfishplugins.java.net/converter/index.html](http://glassfishplugins.java.net/converter/index.html).
Glossary

**Copy Conversion**- Process of copying a project into another workspace and then performing the conversion.

**Document XML Parse**- A method of XML parsing in which the entire document is parsed and then nodes may be extracted by tag names.

**Eclipse**- An open-source, community-driven IDE.

**Factory Design Pattern**- An object-oriented pattern for overseeing the creation of objects at runtime.

**GlassFish**- An open-source enterprise application server maintained by Oracle.

**Iterative Development**- A development process model, in which projects are broken into short one or two week iterations, each going through its own development cycle.

**NetBeans**- An open-source IDE that is maintained by Oracle.

**SAX**- The Simple API for XML Parsing, A pull-based XML parser in which a document is parsed node by node rather than the entire document.

**Standard Conversion**- The process of converting a project within an existing workspace.

**Test First Development**- The development process in which tests are created prior to the production of code.

**Waterfall Development**- A development process model where the process follows a sequential phase structure rather than a cyclic one.
References


