Project Number: GFP 0902

WPI Suite OSGi Framework Conversion

A Major Qualifying Project
submitted to the faculty of
Worcester Polytechnic Institute
in partial fulfillment of the requirements for the
degree of Bachelor of Science

by
Elliot Pennington

Date:
December 18, 2009

Approved:

Gary F. Pollice
Worcester Polytechnic Institute

Abstract

In D term of 2009, a group of software engineering undergraduate students created a particularly solid product called WPI Suite. The original project created a software engineering requirements manager, with a solid but flexible foundation so that additional modules and functionality could be added to the application. WPI Suite was built with its own internal module system.

This MQP project transformed the WPI Suite application into an OSGi-compliant application. OSGi (Open Services Gateway initiative) is the most advanced standard for flexible and modular applications. OSGi allows bundles, which can publish and consume services for other modules, to be dynamically loaded within an application. The OSGi standard helps to minimize inter-module dependencies that often cause problems for large applications. The MQP project built the frameworks and interfaces to the application modules from the outer OSGi layer. Now, WPI Suite's internal module system is fully modular, and no longer suffers from the limitations of old fashioned monolithic applications.

This project serves as a guide for future teams who desire to transform a large project into an OSGi application. Existing documentation focuses primarily on building small projects from scratch into the OSGi engine from the start. For a large existing project, the challenges are quite different.

Contents

1 Introduction

In software projects, engineers always strive to decrease dependencies among the different sections or modules of a project. Dependencies in code cause problems when the code needs to be changed. Code is constantly changing in software projects for many reasons. Often customers request new features, or they find bugs, or they decide they dislike they way something came out. Other times, separate pieces of software, with which the project interacts, will change their public interface, and the project needs to be adapted to work with the new interface.

Internal code dependencies add difficulty when code requires change. Usually, the programmers need to change all the dependent code in addition to the problem spot. This takes time, and modifying working code can introduce bugs into the system.

Future aspirations for WPI Suite are high. Many new plugins can be added for increased functionality. Future teams may add a web interface to WPI Suite while using the same back end. These goals require a more modular WPI Suite that supports a standardized plugin system.

Enter OSGi, formerly the Open Services Gateway initiative, a standard for dynamic, modular programs. Using an OSGi engine for WPI Suite allowed us to build modules that publish or consume services. Modules can be loaded dynamically at run time, for more efficient resource usage. For this project, we chose to use the Eclipse Equinox OSGi engine. The engine is just that, a core. Our application is implemented in bundles which communicate through the core. In the original WPI Suite, the code was already partitioned into modules, but they were not truly separate. This MQP attempts to separate all of the original packages into OSGi compliant bundles. At the end of the project, support for converted OSGi modules was not achieved.

2 Background

2.1 Choices

While we chose to use OSGi to solve the modularity problem, there were other options. The original WPI Suite student group built the program with a plugin framework, that would allow future developers to build plugins for the application. Such an approach is a valid, common, and historically a rather successful way to solve the problem.

A plugin framework provides the plugin developers with a series of objects and public interfaces that allow them to see and use certain pieces of code from the original application. The original application then applies the plugin by accessing it through the plugin interfaces. Good plugin systems are challenging to develop. Plugin frameworks can still cause code dependencies.

OSGi runs each module as a completely separate entity, and has a standardized plugin interface. To handle information transfer between modules, OSGi services are used. A module that requires information from another module will simply look up its service, and consume it. If the other module's service has not been started, the original module can start it, or wait for it, depending on how OSGi is configured.

2.2 OSGi Details

Before an OSGi application can be started, the OSGi core must be started first. There are several OSGi core implementations. This project uses the Eclipse Equinox core (available at http://www.eclipse.org/equinox/). Equinox is started from the command line. Once started, a shell prompt appears. From here one can install, uninstall, run and stop the bundles for your application. The OSGi shell can also give you information about the installed and running bundles. More on this later.[1]

For each OSGi bundle, there must be a manifest file which defines the settings for the bundle. Settings include the Java version in which to run the bundle, and the bundle's name and version. The manifest also defines dependencies for the bundle in the form of both external required bundles and imported packages, which must be supplied, or exported, by at least one of the other bundles running in the same session. The manifest also defines the location of the desired bundle activator, a specific java class of which the `start()` method is called when the bundle is asked to start, and of which the `stop()` method is called when the bundle is asked to stop. The two primary requirements of an OSGi bundle over an ordinary Java project are the manifest file and the bundle activator class.[2]

3 Methodology

This section documents the successes, failures and workarounds of the project.

3.1 Pax Construct

Pax Construct is a set of scripts that comes in a zip file that can be downloaded from http://www.ops4j.org/projects/pax/construct/. Pax is built to work in conjunction with Apache Maven. Maven is a tool built to automate builds for large Java projects. Its features include the ability to download jar dependencies from its repositories dynamically at build time, and link everything together.[3] Maven requires your project to have a specific directory layout and certain files that specify the exact parameters for the magic that Maven will perform. This differs considerably from the more old fashioned Java build tool, Ant, in which the programmer specifies all the build parameters directly. Maven assumes a certain set of specific build parameters to which the program must conform. This means that if the program is not structured the way that Maven wants it to be structured, Maven will not build the program.

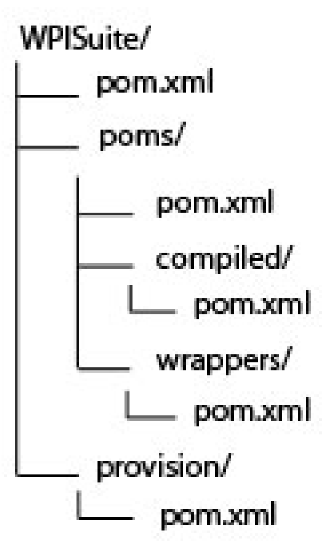
Maven uses xml files called POM (Project Object Model) files to download and link dependencies during the build. The programmer must write the POM files for the program.[4] The POM files require information nearly identical to the information required by the manifest files of OSGi. In fact, it is entirely possible to generate an OSGi manifest from the POM files for Maven. And this is exactly what the Pax Construct scripts attempt to do. Pax cannot be used without Maven.

Note that in most cases, it one cannot generate POM files from OSGi manifest files. POM files require more specific information about each dependency that manifest files do not. It is somewhat unfortunate that it works out this way, as manifest files are considerably easier to write.

3.2 Adventures With Pax Construct

My primary OSGi guide during my adventures with Pax Construct was written by Craig Walls and is entitled *Modular Java: Creating Flexible Applications with OSGi and Spring.* Walls' examples in the book use Pax-Construct, so I chose that as my primary tool to generate working OSGi bundles.[5]

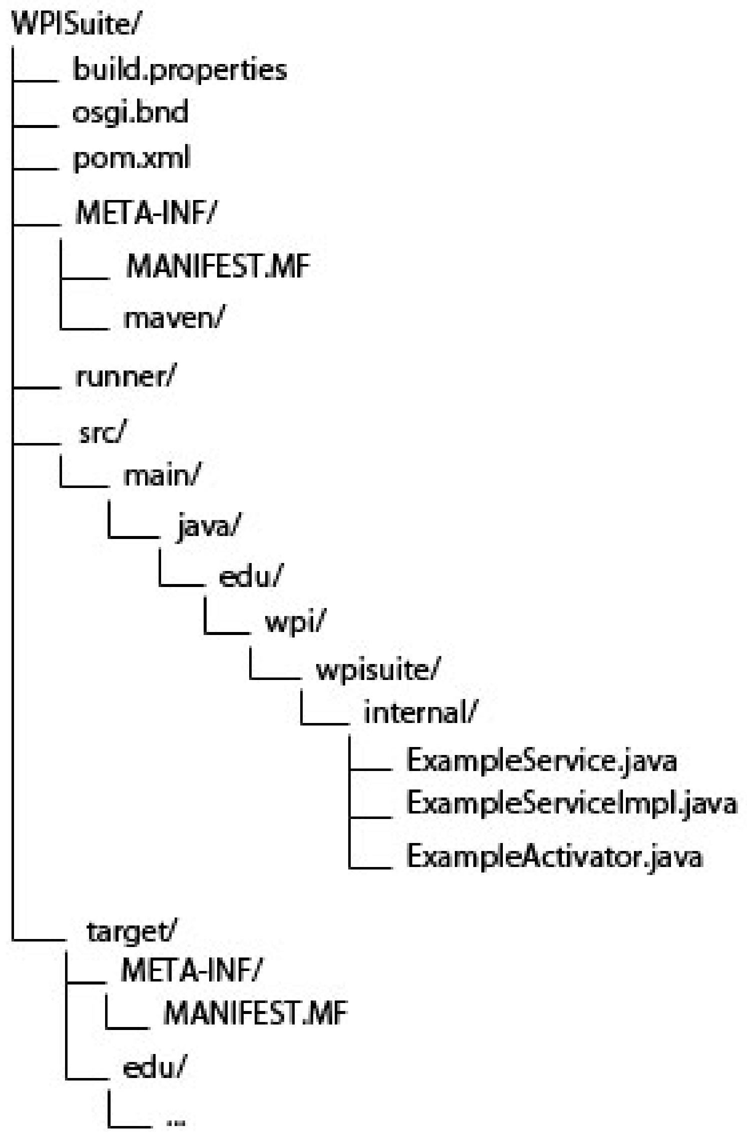
I checked out a fresh copy of the WPI Suite code from the subversion branch I had created earlier for this MQP. Within that directory, I ran `pax-create-project`. This script asks the user for some Maven-specific information about your project, the artifact ID, the group ID, and the version number. Then it sets up the project, and generates a structure that looks like the following:[6]



Next, I created a bundle for WPI Suite. I decided not to create bundles right away, but simply to get WPI Suite both to build and to run within OSGi. To create the bundle, I used `pax-create-bundle`. It needed to know the same type of Maven information about the bundle that it needed to know about the project. The image on the next page shows the structure it generated underneath my project directory. It did provide a few example classes in the `internal/` directory it created. Note the large number of configuration files it generates. The `osgi.bnd` file is used by the BND[2] program to generate the OSGi manifest files from the pom.xml file. The `runner/` directory is recreated each time the program is run in OSGi via the `pax-provision` script.[7]

Next, I set about moving all the WPI Suite packages and classes into their appropriate places in the bundle's `src/main/java/` directory. WPI Suite uses a lot of external jar files that it keeps in a `lib/` directory off the top project level. I emulated this, but I added the `lib/` folder under the project level directory. I copied all of WPI Suite's required jars there.

I tried to run the `mvn install` command from the bundle level next. This is Walls' next step. It builds the bundle and deploys it into the local Maven repository. The build failed. I had not set any of the bundle dependencies. But it did give me a very long readout of all of the compiler errors from the code and the packages it was expecting to find but could not. Maven found the code, and attempted to compile it.[8]



```
WPISuite/
|___ build.properties
|___ osgi.bnd
|___ pom.xml
|___ META-INF/
|    |___ MANIFEST.MF
|    |___ maven/
|___ runner/
|___ src/
|    |___ main/
|         |___ java/
|              |___ edu/
|                   |___ wpi/
|                        |___ wpisuite/
|                             |___ internal/
|                                  |___ ExampleService.java
|                                  |___ ExampleServiceImpl.java
|                                  |___ ExampleActivator.java
|___ target/
     |___ META-INF/
     |    |___ MANIFEST.MF
     |___ edu/
          |___ ...
```

Next, I added the dependencies for the jar files that came with WPI Suite. Dependency entries go in the bundle's top level POM file and look like this:[9]

```
<dependency>
  <groupId>snaq.db.dbpool</groupId>
  <artifactId>dbpool</artifactId>
```

```
    <version>4.8.3</version>
  </dependency>
```

I added one entry for each of the required jars. The `<groupId>` and `<artifactId>` entries are not arbitrary. The programmer has to find the correct group ID and artifact ID for each required jar file. Maven needs this information so it can try to download the files. The version numbers are specific too, but they are usually in the filename of the jar in question. After I added I all the dependencies, I ran a new `mvn install` with more success. The build still failed, but I got more concrete messages about what exactly was missing.

Maven failed to find some of the jar files that I had asked it include, but it told me to download them myself and it also gave me the command I should use to install them. I already had the missing jars in my lib folder, so I just used Maven's command and pointed it at that directory. I do not know where Maven stores the information I gave it about the file's location. There is no sign of it in any of the POM files. Another `mvn install` brought me closer, but it now was asking for several more jars that I didn't have. Maven had decided that some of the jars I was including had their own jar dependencies. Finding some of the jars that Maven wanted was difficult. One more `mvn install` gave me yet another jar dependency for one of the new jars. After I installed that, the build was finally clean.

Pax Construct includes a script to run a project in an OSGi session; `pax-provision`. My bundle built, so I ran `pax-provision` on the project, the next step. I got a lot of errors. All of my jar files were missing. Walls explains that all of the jars need to be converted to bundles so that they can export their packages for use by the application bundles. The next task, therefore, was to run `pax-wrap-jar` on all of my jar files. Pax keeps track of the locations of the jar files in a project. The `pax-wrap-jar` script asks only for the name of the jar and the version, but not the location.[10]

I re-ran `pax-provision` and got no errors, but WPI Suite was not running. Walls talks about this problem in his book; the bundle's explicit dependencies are resolved, but that a dependency further in the program is not met. He explains that the stack traces and logs are unlikely to point to the problem, and so the solution is often hard to find. In his example, his project had a jar that required Jakarta Commons Logging. Commons Logging is not available as an OSGi bundle, though. He used a different logging library called Pax Logging, which supports not only Commons Logging, but also other common logging libraries. So I used the command he provided to install the Pax Logging bundle. It was available from Maven.[11]

This helped. WPI Suite ran. I had to set up a bundle activator first to call WPI Suite's main method. The `MANIFEST.MF` file that Pax generated insisted upon pointing to their generated `ExampleActivator` for the bundle's generator. Since Pax generates the manifest file each time `pax-provision` is run, I chose to modify `ExampleActivator` to suit my needs.

I got a `NullPointerException` in the `ModuleLoader` class in WPI Suite. Digging around revealed that WPI Suite knows which external modules to load by reading its `wpisuite.module` file from the bundle root directory. With Pax, it was not clear what the root directory for the bundle is; it was unable to find it in the top level of the bundle directory tree. I modified the code to print the directory in which it was looking, and saw that the root is actually inside the bundle's generated `runner` folder. This provided some difficulty, because no file can be placed in `runner`, since it is regenerated every time a `pax-provision` is is executed. I modified the WPI Suite code to look one level higher, which worked.

This is where progress came to a halt with Pax Construct. I had a lot of errors in the project. Many classes were not being found. All of the dependencies in the manifest file were marked as `optional`, and it was unclear if that was correct. OSGi was giving me errors at startup, saying that it was `skipping` all of my installed bundles, even though some of them seemed to work. There was a lot of inconsistent behavior, and so much happening behind the covers of the Pax scripts that I could not isolate the problems. I chose to start over with a new tool.

3.3 Starting Over With Eclipse

Eclipse worked. Eclipse gave me the transparency I needed to configure WPI Suite properly to work in OSGi. This section explains how I resolved each problem I encountered along the way.

I created a new Plug-In project in Eclipse, set to run as a framework in Equinox, not as an Eclipse Plug-In. I created all the packages manually, and gave them the same names as they had in the original WPI Suite. Then I moved all the source files into the appropriate packages. Eclipse automatically compiled the project, and it gave me an error on a call to construct `com.sun.ssl.internal.ssl.Provider()`. I read the Javadoc for this class and for others related to it. It did not appear to be necessary to generate a new `Provider` object. It was assigned to an unused variable, so I commented out the line.

I opened up the Manifest file. Eclipse has a tabbed view for this file with a graphical interface that automatically write your Manifest depending on what you tell it to do in the GUI. In the dependencies tab, I added all of the packages that came in all of the required jar files into the Import Package section. Many of these gave me errors. Eclipse could not find any bundles that were exporting those packages.

I created a new plug-in project from the Plug-In Development folder of all possible project types. I needed a "Plug-In from Existing JAR Archives."[12] I loaded all of the jars into this one project. I set my WPI Suite bundle manifest to require that bundle, and the Eclipse builder found all of the packages required in the WPI Suite bundle manifest. Note that under ordinary circumstances, it would be better to put each jar into its own bundle, for additional flexibility. I was seeking the least possible resistance to a working application.

I tried to run the bundles from Eclipse. I set it to run in a separate OSGi session from the one in which Eclipse was running. It didn't work. Eclipse tried to run the bundle in the Eclipse OSGi session regardless. This failing me, I made a new directory and copied in an `org.eclipse.osgi.jar`. I exported both the WPI Suite bundle and the bundle with the required jars as separate jar bundles into that directory. I started Equinox with the following command:

```
$ java -jar org.eclipse.osgi.jar -console
```

Once in the OSGi shell, I installed both bundles into the OSGi session with the OSGi `install` command.[13] The WPI Suite bundle depends on the required jar bundle, so I started the required jar bundle first. No problems. I started the WPI Suite bundle next, and got errors. WPI Suite was missing a lot of packages that it needed. All the packages it was missing were exported from the required jar bundle. I ran the OSGi `bundle` command on the required jar bundle to check its status. It was exporting all the packages that were missing, but they were all set to version number `0.0.0`. The WPI Suite bundle's manifest was set to require specific version numbers of all of the missing packages. It had been set that way by the graphical tool I had used to set the package imports. In order to see the version number of the imports, it is necessary to view the raw `MANIFEST.MF` file. The format is easy to read, so this is not undesirable. I deleted all the version requirements from the import settings, allowing any version of the imported packages. I was not planning to use Maven at any point for this build, so I was content with this solution, though it is not entirely ideal.

My next attempt to start the WPI Suite bundle yielded errors as well. I was missing a few more packages. After inspecting my exported packages from the required jar bundle versus my imported packages, I noticed that there were a few extra packages that I had included by accident with the graphical tool. I removed the extra packages, re-exported, and started up the WPI Suite bundle. This time the startup was clean. I had managed to satisfy all of the explicit dependencies of my bundle.

When all the explicit dependencies are satisfied, the OSGi session starts a new thread for the bundle and calls the bundle activator. My bundle activator just does all of the things the WPI Suite `main()` method does, finally creating a new `WPISuite()`. I got a lot of `NoClassDefFoundError`s when OSGi ran the code. Most notably, these were errors on common Swing classes. I set the WPI Suite manifest file to include the Swing packages. This solved the immediate problems with Swing.

I got a series of errors next that were fairly straightforward. I had forgotten to move the `wpisuite.module` file into the new directory. It had to go into the top level directory on the same level as the `org.eclipse.osgi.jar` Equinox file, as that's the root of the OSGi session. There was another file that WPI Suite used its database configuration information called `database_config.ini`; it needed to go into the root OSGi directory as well.

I started to get WPI Suite's code to advance far enough to get some graphics. The EclipseStarter opened up and ran WPI Suite. Before anything happened, I got two java alert windows, both informing me of a new `NoClassDefFoundError` on `javax.crypto.spec.PBEKeySpec`. After clicking through those, I got the login window for WPI Suite, but it quickly brought up the database configuration window. It was supposed to load the database information from `database_config.ini`, but all of the fields were blank. Clicking cancel on the database configuration window brought up a second login window, so now there were two. Both the login window and the database configuration window were failing to display their button icons.

I opted to try to fix the missing `PBEKeySpec` first, since it was occurring first. I used my typical method of including the package in the manifest file, but to no avail; I still received the `NoClassDefFoundError`. This is a very important error in the project and leads to some very important facts about OSGi.

Unsure of where to go with the missing PBEKeySpec, I searched the original WPI Suite project to find the location of the icons. They sit in a folder called resources in the project root directory. I copied this into my OSGi root folder and tried again – still no icons. I checked the permissions with an ls -l and discovered that they were very limited; only the owner could access this file. A chmod -R 755 on resources fixed the problem. WPI Suite now finds the icons.

I tried the same trick on the database configuration file, since the fields in the login window were failing to populate, but to no avail. Next, I tried entering the data into the fields myself; I just copied it from database_config.ini. Nothing happened, but I got some new errors in the stack trace in the terminal; some new NoClassDefFoundErrors on classes from javax.naming and from com.mysql.jdbc. The JDBC package is already included and exported from my required jar bundle, so I added javax.naming to the WPI Suite manifest as an imported package; it is exported by the Equinox bundle as part of the standard Java library.

Restarting WPI Suite yielded no changes. Both javax.naming and com.mysql.jdbc were still missing. I got a better view of the login window as it was starting this time. The database fields started populated, but they cleared almost instantly as the java alert popped up to inform me of the missing PBEKeySpec. This was a good sign. WPI Suite was reading the database configuration file, but the code to use it failed because it couldn't find the class definition for PBEKeySpec. My two problems had become one.

Meanwhile, in the terminal, I got messages that some of the modules were failing to initialize. The primary cause of this was a NoClassDefFoundError on the Swing class JPanel. My bundle was now very clearly finding some classes and ignoring others, regardless of configuration. If it were unable to find all Swing classes, I would still have the missing UIManager problem from right after the first clean start. Since inconsistent behavior is difficult to debug, I needed to try to make it consistent.

I decided to convert one of the failing WPI Suite modules into an OSGi bundle. I chose the ProjectAdministration module to convert. The conversion was not difficult. I wrote an OSGi activator for this new bundle to publish a service of an instance of a Module, which the ProjectAdministration class extends. All classes extending Module require an instance of an ActiveState object as an argument to their constructors. In the WPI Suite system, ActiveState is a singleton which keeps track of the state of the system, so ActiveState.getInstance() is called on instantiation of every module.

Since the ActiveState class is located in the WPI Suite bundle, I made a quick service to publish it, and registered it in the WPI Suite activator. Since the original ActiveState was not implemented as a thread safe singleton, I also made that change, since each bundle runs in a separate thread. To get the ActiveState in the ProjectAdmin bundle, I used a ServiceTracker to find the service so I could use it in the constructor for the ProjectAdmin class. I ran the bundle a few times correcting some basic configuration errors, and wound up with the exact same stack trace as the WPI Suite module. The primary error was a NoClassDefFoundError on JPanel. The new bundle had a very minimal configuration setup, unlike the WPI Suite configuration, which by this point, had lots of imported packages and special options. This led me to believe that the root of the problem was not the WPI Suite configuration.

The stack traces I was getting for this did not make sense. Below is an example. Notice that the trace clearly finds and moves through several methods within the JPanel class, only to give a NoClassDefFoundError on it later on.

```
Caused by: java.lang.NoClassDefFoundError: javax/swing/JPanel
        at java.lang.ClassLoader.defineClass1(Native Method)
        at java.lang.ClassLoader.defineClass(ClassLoader.java:700)
        ...
        at java.lang.ClassLoader.loadClass(ClassLoader.java:254)
        at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:399)
        at java.lang.Class.getDeclaredMethods0(Native Method)
        at java.lang.Class.privateGetDeclaredMethods(Class.java:2427)
        at java.lang.Class.getDeclaredMethod(Class.java:1935)
        at java.awt.Component.isCoalesceEventsOverriden(Component.java:6033)
        at java.awt.Component.isCoalesceEventsOverriden(Component.java:6022)
        at java.awt.Component.isCoalesceEventsOverriden(Component.java:6022)
        at java.awt.Component.access\$500(Component.java:169)
        at java.awt.Component\$3.run(Component.java:5987)
        at java.awt.Component\$3.run(Component.java:5985)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.awt.Component.checkCoalescing(Component.java:5984)
        at java.awt.Component.<init>(Component.java:5953)
        at java.awt.Container.<init>(Container.java:251)
        at javax.swing.JComponent.<init>(JComponent.java:581)
        at javax.swing.JPanel.<init>(JPanel.java:65)
        at javax.swing.JPanel.<init>(JPanel.java:92)
        at javax.swing.JPanel.<init>(JPanel.java:100)
        at edu.wpi.wpisuite.components.editor.EditorPanelView.<init>(EditorPanelView.java:63)
```

I decided to set up a completely separate application as a bundle to run in the OSGi session. This application would not depend on or interact with any of the other bundles, but it would use the JPanel class, which the other bundles were unable to find. For some quick code, I went to the java.sun.com website for the Swing tutorial, and found an example application that used `JPanel`. I copied the code into a new project, wrote an activator to start it up, and loaded it into my OSGi session. This bundle started up right away. No problems with `JPanel`.

There was only one thing left to check: my bundle with the required jars that I had not changed since I made it. I opened the manifest and included a few of the missing packages, including the Swing packages and `javax.crypto.spec`. It worked. I no longer got the errors for those missing classes. There were a few more packages I needed to set as included in the required jar bundle, and I went through all of those until I got a new error that could not be solved by that method.

```
java.lang.NoClassDefFoundError: Could not initialize class com.mysql.jdbc.ConnectionImpl
        at com.mysql.jdbc.NonRegisteringDriver.connect(NonRegisteringDriver.java:282)
        at java.sql.DriverManager.getConnection(DriverManager.java:582)
        at java.sql.DriverManager.getConnection(DriverManager.java:185)
        at snaq.db.ConnectionPool.create(ConnectionPool.java:95)
        at snaq.util.ObjectPool.checkOut(ObjectPool.java:178)
        at snaq.db.ConnectionPool.getConnection(ConnectionPool.java:200)
        at snaq.db.ConnectionPoolManager.getConnection(ConnectionPoolManager.java:571)
        at net.java.ao.db.DBPoolProvider.getConnectionImpl(DBPoolProvider.java:62)
        at net.java.ao.DatabaseProvider.getConnection(DatabaseProvider.java:734)
        at edu.wpi.wpisuite.database.Database.testConnection(Database.java:283)
        at edu.wpi.wpisuite.database.DatabaseConfig.validateConfig(DatabaseConfig.java:217)
        at edu.wpi.wpisuite.database.DatabaseConfig.validateConfig(DatabaseConfig.java:229)
        at edu.wpi.wpisuite.app.LoginController.<init>(LoginController.java:43)
        at edu.wpi.wpisuite.app.WPISuite.<init>(WPISuite.java:61)
        at edu.wpi.wpisuite.app.WPISuite.startprogram(WPISuite.java:178)
        at osgi2.Activator.start(Activator.java:32)
        ... 32 more
```

Note that the method called before the exception is thrown is in the same package as the class for which no definition was found, so it cannot be a dependency problem. I looked, and the `ConnectionImpl` class was present in the package being exported by my required jar bundle. These `com.mysql` packages are part of the MySQL connector for java that the WPI Suite team chose to use for their database connections. They are all in the required jar bundle, and there is no reason for a `NoClassDefFoundError` on any of those classes. When I got this problem, WPI Suite's graphical interface no longer started; no more login screen.

I searched the Internet for any information regarding `jdbc` and OSGi. Google yielded this blog: http://hwellmann.blogspot.com/2009/04/jdbc-drivers-in-osgi.html. The blog entry is asking for a solution to the same problem I am having, only he also provides a workaround that, in his opinion, is ugly.

Putting my own concerns for aesthetics aside, I used his solution, which was to call `Class.forName(driver)` in the activator class of the bundle containing `driver`, where `driver` is the JDBC driver being used for the connections. The blog author was using numerous drivers, so he disliked having to specify all of them, but WPI Suite uses only one driver.[14]

I located the driver and put a call to `Class.forName(com.mysql.jdbc.Driver)` in the activator class for my required jar bundle. Note that this cannot be done to the stock project that Eclipse creates from the existing jar files. The `.classpath` file will show a line that looks like:

```
<classpathentry exported="true" kind="lib" path=""/>
```

The `path=""` indicates that the root of the project is set to kind `lib`, and Eclipse will not allow a source package to be nested within a folder set to `kind="lib"`. The solution to this is to move to the Navigator view in Eclipse, create a new folder in the project, and move all of the packages into the new folder. Next, open the `.classpath` file and change that `classpathentry` line so that the `path=` points to the new folder. Then make a new folder in the project root. Call it `src`. Add a line to the `.classpath` that looks like this:

```
<classpathentry kind="src" path="src"/>
```

Now it is possible to make a package in the `src` folder to hold the activator.

This was the last hurdle. With this change, WPI Suite started right up. I logged in and connected to the test database. I could see other projects, and create my own project. I tried a bunch of actions in WPI Suite and it did not break.

4 Results and Analysis

The work resulted in a working WPI Suite application contained within a single bundle, with the required jars in a different bundle. My attempts to convert the WPI Suite modules into OSGi bundles was unsuccessful; the configuration difficulties were unsurmountable in the remaining time for this project.

4.1 Why Pax Construct Failed

Pax Construct used an additional project level layer on top of the bundles. Note that Eclipse does not use this. When using the Pax Construct scripts to wrap the jar files, there was no way to set dependencies for those bundles in any sort of specific way. There was also no way to see and start those bundles manually. The vast majority of the jar files that I supposedly turned into bundles were never even installed into the project's OSGi session. Some were though, most notably the Pax Logging utility jars, which already came as bundles. Pax had clear problems loading the jars that it had moved into bundles.

In hindsight, I think that I would have needed to modify the POM files for each of the jars that had become bundles. At the time, this did not occur to me. Why would I add Maven instructions to a jar file that was already compiled and built? That still does not explain why the jar bundles were never installed into the project's OSGi session, though.

Using Pax Construct allows very little visibility into what OSGi actually needs and uses to work. It adds a lot of complexity because it incorporates Maven into it as well. I had never used either Maven or OSGi before, so I had some problems figuring out exactly where the line was between Maven and OSGi. The answer to that question is not very clear. As best I can tell, the line is blurry and varies depending on the operation. For certain stages of the project, Pax takes much of its information for OSGi from the Maven configuration. At other times, the Pax scripts are needed to set OSGi specifics and do not alter Maven settings at all.

Because of my initial limited understanding of OSGi, I was unable to see through Pax Construct well enough to get the project working. In addition, I was unable to learn much about OSGi while struggling with Pax because of its opacity and its confusion with Maven. When I started setting up the project in Eclipse, OSGi became a lot more clear.

4.2 OSGi

OSGi is an impressive technology. This section will explain some details about it that I left out of the Methodology section.

4.2.1 Stack Traces and Dependencies

The stack traces from OSGi configuration errors often point to the root of the problem indirectly at best. Below is a stack trace I encountered during the conversion. The important pieces are italicized.

```
Caused by: java.lang.NoClassDefFoundError: javax/swing/JPanel
```

```
at java.lang.ClassLoader.defineClass1(Native Method)
at java.lang.ClassLoader.defineClass(ClassLoader.java:700)
...
at java.lang.ClassLoader.loadClass(ClassLoader.java:254)
at java.lang.ClassLoader.loadClassInternal(ClassLoader.java:399)
at java.lang.Class.getDeclaredMethods0(Native Method)
at java.lang.Class.privateGetDeclaredMethods(Class.java:2427)
at java.lang.Class.getDeclaredMethod(Class.java:1935)
at java.awt.Component.isCoalesceEventsOverriden(Component.java:6033)
at java.awt.Component.isCoalesceEventsOverriden(Component.java:6022)
at java.awt.Component.isCoalesceEventsOverriden(Component.java:6022)
at java.awt.Component.access$500(Component.java:169)
at java.awt.Component$3.run(Component.java:5987)
at java.awt.Component$3.run(Component.java:5985)
at java.security.AccessController.doPrivileged(Native Method)
at java.awt.Component.checkCoalescing(Component.java:5984)
at java.awt.Component.<init>(Component.java:5953)
at java.awt.Container.<init>(Container.java:251)
at javax.swing.JComponent.<init>(JComponent.java:581)
at javax.swing.JPanel.<init>(JPanel.java:65)
at javax.swing.JPanel.<init>(JPanel.java:92)
at javax.swing.JPanel.<init>(JPanel.java:100)
at edu.wpi.wpisuite.components.editor.EditorPanelView.<init>(EditorPanelView.java:63)
```

Note that Java find `JPanel` early in the trace, and several methods in its constructor run. Later in the same trace, a `NoClassDefFoundError` is thrown on `JPanel`. Why? `JPanel` is included as a dependency for part of the trace, but not for other parts of the trace. If the trace switches to use packages from a different bundle, the new bundle's configuration is used to determine dependencies.

The misleading stack traces make OSGi configuration difficult. In addition to pointing to problems in manifest files, stack traces can also point to problems in a bundle's `build.properties` file. Often such problems are characterized by a bundle being unable to find its own member packages. This sort of behavior from a stack trace is confusing unless the programmer has experience.

4.2.2 Services

OSGi services allow an object to be visible to bundles other than the one in which the object was created. When bundles "publish a service," they are submitting an object to a centralized OSGi repository of services. Other objects can then "consume the service," whereby they call a `ServiceTracker` to retrieve the desired object for use.

The OSGi service system is a standardized plugin system. Building a poorly designed set of interfaces with lots of interdependencies is still possible under OSGi, but it is not easy. Using OSGi urges programmers to build loosely coupled interfaces and modular plugins, if only because it takes more effort to do otherwise. There is a significant amount of overhead code involved in publishing and consuming services. Ordinarily considered objectionable, this attribute of OSGi enforces good code design.

5 Future Work and Conclusions

OSGi is a valuable system because it offers standards that enforce modular design. However, conquering the configuration difficulties requires much development time. OSGi makes sense as an engine choice for new projects. Converting existing projects is not an effective solution for an application built with poor modular design; for such an application to utilize the advantages of OSGi, it would need to be heavily restructured. The strengths of OSGi show mostly in the application building process because it enforces good design; it is harder to configure OSGi to work for an application built without modular code design.

I do not recommend future work on the OSGi-enabled WPI Suite. Pre-OSGi WPI Suite features its own plugin system which works quite well. Unfortunately, neither the user interface nor the application core is implemented in plugin form. This means that building a web based front end (or extending any other part of the non plugin core) would require significant restructuring of the original design. Even for an OSGi-enabled WPI Suite, the same degree of restructuring would be needed in order to use OSGi advantages for a web front end, only the programmers would would be hindered by the OSGi configurations as well. Other than experiencing OSGi, such a project would have little more educational value than doing the same restructuring for the original WPI Suite, and possibly less, since OSGi configuration problems can halt project progress.

Should future work be done to further the conversion of WPI Suite to an OSGi application, the team should not use the conversion made during this MQP. The team needs to restructure WPI Suite to be fully modular in small pieces, and do the OSGi conversion as though the application were being built from scratch. The first step would be to define an application core, a runnable program implemented as an OSGi bundle, capable of coordinating other OSGi bundles as plugins. The second step would be to create a bundle to support user login. Then maybe to build the core of the desktop interface as a bundle. The rest of the functionality would need to be broken into bundles as well, and added piece by piece.

Continuing to use the current conversion will yield only frustration as future teams try to maintain OSGi compatibility. This method of working backwards will be less effective, because the focus will be on working with OSGi to conform to an existing WPI Suite. This approach does not utilize OSGi's strengths. This MQP was an attempt to make OSGi work for WPI Suite, not the other way around, and the progress was slow. Should the conversion effort continue in this direction, WPI Suite will be lost in a myriad of stack traces.

6 References

Notes

[1]Walls, Craig. *Modular Java.* Dallas Texas: The Pragmatic Bookshelf, 2009.
[2]Ibid.
[3]Ibid.
[4]"Introduction to the POM". Apache Maven. 17 November 2009
<http://maven.apache.org/guides/introduction/introduction-to-the-pom.html>.
[5]Walls, Craig. *Modular Java.* Dallas Texas: The Pragmatic Bookshelf, 2009.
[6]Walls, Craig. *Modular Java.* Dallas Texas: The Pragmatic Bookshelf, 2009.
[7]Ibid.
[8]Ibid.
[9]"Introduction to the POM". Apache Maven. 17 November 2009
<http://maven.apache.org/guides/introduction/introduction-to-the-pom.html>.
[10]Walls, Craig. *Modular Java.* Dallas Texas: The Pragmatic Bookshelf, 2009.
[11]Ibid.
[12]Varttinen, Robert. "Adding a Third Party JAR to your Eclipse Plugin". November 16
2009 <http://robertvarttinen.blogspot.com/2007/01/adding-third-party-jar-to-your-eclipse.html>.
[13]Vogel, Lars. "OSGi with Eclipse Equinox". November 16 2009 <http://www.vogella.de/articles/OSGi/article.ht
[14]Wellmann, Harald. "JDBC Drivers in OSGi". 11 December 2009 <http://hwellmann.blogspot.com/2009/04/jdb
drivers-in-osgi.html>.