Essential OSGi

Winning Strategies for Software Evolution

Gilbert Carl Herschberger II

Essential OSGi

Winning Strategies for Software Evolution

Gilbert Carl Herschberger II

This book is for sale at http://leanpub.com/essentialosgi

This version was published on 2014-02-27

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my best friend and wife, Yvonne.

Contents

I Winning Strategies **1**

1 Start Here **2**

I Winning Strategies

1 Start Here

1.1 Ten Winning Strategies

These strategies are essential for evolving from a large, monolithic Java class library to a coherent collection of OSGi modules.

1. Create a large, monolithic *wrapper* module. Immediately *provide* existing Java packages to other modules. Immediately *hide* implementation packages.
2. *Move* Java source code from Java project to plug-in project.
3. Create exactly one *API* module. Create many *implementation* modules.
4. Use Test-Driven Development. For every production module, create a corresponding test module. Create the test module *first*.
5. Replace custom plug-in mechanism with the OSGi service mechanism. Instead of polling for plug-ins, let plug-ins register themselves. Replace custom class loaders with OSGi class loaders.
6. Rewrite code incompatible with OSGi. Eliminate polling for resources. Split a package between interface (API) and implementation. Split a package between application programming interface (API) and service provider interface (SPI).
7. Create a custom Eclipse command. Create batch programs for production and test environments.
8. Create custom OSGi URL stream handler services.
9. Release a coherent collection of OSGi modules.
10. Refine collection of OSGi modules with module refactoring.

Essential strategies are both important and necessary. Without these strategies, you can introduce unnecessary delay. Let's say that you have a perfectly adequate Java package from an existing Java project. What prevents you from using the package inside an OSGi container?

1.2 Only one issue

This document focuses on one issue: Evolving existing software to embrace OSGi. It takes you through a long-term migration process, from having zero (0) modules to having a coherent collection of modules.

- When I have so many Java packages, where do I start?

- Understanding four kinds of modules: API, Dependency, Implementation and Wrapper. How many modules do I need? How many Java packages should a module provide? How many API modules do I need?
- Understanding module refactoring, such as combining two modules and splitting a module. How do I improve module design without starting over? How do I refactor a module, while maintaining backward compatibility with a previous module design *and* multiple target environments?
- Understanding Java refactoring and its effect on modules, such as splitting a package between API and implementation. How do I identify and fix defects in my existing API design?

1.3 Assumption

I'm making the following assumptions.

- JDK 1.6 or higher is already installed and available on your computer. I'm not going to explain how to locate, download and install a JDK.
- Eclipse Indigo or higher is already installed. I'm not going to help with the installation of Eclipse.
- This book is focused on the long-term, high-level strategy and tactics of building modules. It is not a step-by-step tutorial on writing Java code so it doesn't have the typical exercise format of other books in the *Essential Training* series.
- If you're not using Eclipse, that's fine. I'm confident that you'll be able to translate between the Eclipse-specific instructions and what you need for your favorite IDE.

1.4 Featuring CjOS Project

In this document, the primary examples come from volunteer work on the free license and open source CjOS Project (http://cjos.sourceforge.net). CjOS is a platform-independent operating system. It is a *co-operating system*.

1.5 Internet-facing production application

The pressure is on. When working on a production application, you need to be able to keep it up and running, 24x7x365. That's twenty-four hours per day, seven days per week, three hundred and sixty-five days per year. You don't have time to play around with the theoretical, pure application of OSGi across all of our production applications at the same time.

- Continue to use your favorite JARs. Using these strategies you can bring a collection of your favorite JARs into the world of OSGi.
- Continue everyday without business interruption. Using these strategies you can evolve your large, monolithic Java class library to a coherent collection of OSGi modules without business interruption.

1.6 Limitations

We have at least two limitations when maintaining a flagship product in production environment:

- Existing Java code, and
- testing.

1.6.1 Existing Java code

Since we're talking about a large body of existing Java code, a lot depends on how well the existing Java code is compatible with the OSGi model.

- **Some Java packages are easy for a module to provide**. Sometimes, it is a simple matter of adding entries to a manifest file.
- **Some Java packages are difficult**, if not impossible, to use inside an OSGi container. A Java package might work outside but not inside an OSGi container.
- **A Java package may need to be refactored**. When a Java package doesn't work inside an OSGi container, it may need to be refactored. What is package refactoring? We explain in a later chapter.

1.6.2 Testing

While you're under pressure and stress, we want to make sure that you test what needs to be tested. You need to test a feature in its old context, a classic Java application. You need to test a feature in its new context, the OSGi container. Just because a feature works in one context does not mean that it will work in the other.

As the purpose of the object-oriented infrastructure is to increase the chance of success, the purpose of using OSGi is to increase the chance of success when running software.

General terms

In this book, I decided to use general terms instead of OSGi-specific terms.

- A module is also known as a bundle. At runtime, a module is a JAR file. In Eclipse terms, a module project is a plug-in project.
- When I say that a module "provides" a Java packages, others will say that a bundle "exports" a package.

1.7 Assumptions

I'm going to assume that you have some knowledge of Java programming and you're starting out with OSGi. Maybe you've read a few books on OSGi and can't find traction to get started on a migration.

1.8 Are you stuck?

After reading about the benefits and opportunities of OSGi, I was sold on the idea. I have designed and written custom class loaders. I have designed and written custom plug-in mechanisms. I understand why classic JARs are bad, really bad.

So, why couldn't I go out and start writing a bunch of new OSGi modules? I was stuck. I asked myself:

1. Where do I start?
2. Do I have to rewrite everything?
3. Do I understand OSGi well enough to make meaningful modules?
4. How many modules should I plan?
5. How do I decide which capability goes into which module?
6. It's going to be embarrassing if I make a mistake on the internal details of an OSGi module for everyone to see. Am I prepared for failure?

I sit down at my computer. I stare at a blank page and nothing comes out. I do not have a working theory on how to get to where I want to go.

It is as if I have the dreaded disease called writer's block. I am compelled to write something; but, when I sit down with a blank sheet of paper, I've got nothing. I have no idea how to proceed.

1.9 What this book is not

I don't want to write a book about how great and wonderful OSGi can be. I don't want to sell you on the idea of using OSGi for all of your future projects. While I'm enthusiastic about getting OSGi to work at all, I hope that you've already been sold on the idea of using OSGi and you want to take the next logical step: Evolving to OSGi.

On the other hand, you may have been given a directive from your boss to "OSGi-ify" some of the existing libraries for your ultra-conservative corporate organization. They want to retain their investment in the existing Java source code and take advantage of the feature of OSGi. They may not know exactly what the features are. OSGi sounds good to someone so they want you to get it to work.

In either case, there are successful and unsuccessful strategies in evolving and embracing OSGi.

What are some of the unsuccessful strategies? We'll talk about that, too.