Hand-in-1

Lucas Sass Rósinberg

General information

In general, have I been having some trouble getting to get most of the lab exercises working on its own. I have been trying to make it from scratch but had troubles with each of the labs and got suggested from one of the instructors before easter break to use the examples in the GitHub repository and as I was told it was more important that I could show my understanding of the underlying technique which we were supposed to work with. Therefore, will most of the source code be examples code provided in the SB4-KOM-F20 repository adjusted for me to run it.

Java Service Loader (JavaLab) – project name in zip: AsteroidsServiceLoader

In the JavaLab I primarily used the provided example AsteroidsServiceLoader from SB4-KOM-F20 repository but did add an enemy module which in short. I added an implementation of an enemy, which is based on the player module. Instead of moving with the keyboard it moves from left to right and from top to bottom randomly.

The Java service loader work by using a service which is a well know set of interfaces and usually abstract classes. To loading a service, a service provider class which is used to proxy the needed information to a provider so that the provider can for fill a certain request together with the code so that it can create the actual provider on demand.

A service provider is identified by placing a provider configuration file in the resource directory known as META-INF / services. The name of the files is the binary name of the service type. If a particular provider class is mentioned more than once or are named in another configuration file, the duplicate is ignored.

Providers are located and instantiated lazily this means that they are loaded on demand, and lazily. A service loader handles the cache of the providers that have been located.

This means in our case that the SPILocator.java which is in the common module dk.sdu.mmmi.cbse.common.util package is used to instantiate a service provider interface which in this case is used to locate service providers in the META-INF configuration files so that they can be loaded and unloaded. This is implemented in the Game.java file (located in core dk.sdu.mmmi.cbse.main) which is then instantiated when main.java is running.

NetBeans Asteroids game (NetBeansLab1) – Project name in zip: AsteroidsNetbeansModules2

In the NetBeansLab1 exercise I used the provided example AsteroidesNetBeansModules (From GitHub repository) as I could not get my modules to work, and tried to get help from one of the instructors which after some time not being able to help me suggested that I use the provided example and try get that running as I was told I wasn't as much the programming itself as it was showing the understanding of the underlying techniques.

To make the application modular, I used modules on top of the NetBeans Platform which allows the application to be extended by additional modules and adapted for specific needs. This also insures that they are able to communicate with each other without being dependent on each other and therefore achieve loose coupling of modules. This is achieved by encapsulation code within modules, so that the NetBeans Platform provided class loader System can be loaded by its class loader, and in the process make them independent unit. Because of this independent unit a module can explicit make its packages specific functionality available to other modules. To use this functionality from other modules a module must be declared a dependency on other modules. A Dependency is declared in the modules manifest file and is then resolved by the NetBeans runtime container, this ensures that the application always starts up in a consistent state. NetBeans Runtime Container consist of 5 modules Bootstrap, start up, Module System API, Filesystem API, and Lookup & Utilities API. Bootstrap is responsible for executing all registered command line handlers and create a boot class loader which loads the start-up module, and then executes them. Start-up deploys the application by initializing the module system and the file system .The Module System API is responsible for management of modules and their dependencies. The File system API provides a virtual filesystem which provides a platform with independent access, which is mostly used for loading resources of the modules. Lookup & Utilities PI provides an important base component which is used for intercommunication between modules. The runtime container is the minimal form of a rich client application and can be executed as such without further modules. Because of this if there are no task the runtime container would directly shutdown again after starting up.

So, the major difference from this to the earlier lab is that instead of using pom files and using the service loader class we get NetBeans to do must of the underlying work for us using the NetBeans Runtime Container.

Dynamic NetBeans Asteroids Game (NetBeansLab2) – Project name in zip: AsteroidsNetbeansModules1

In the netbeansLab2 exercise I used the same repo as the template, as I was not able to make bundles and get them to talk to each other. And therefore, focused on getting them to work, using the needed implementations. So, for this exercise I implemented the update center needed to work with the silent updater so that the modules could be loaded dynamic. The NetBeans Platform provides a tool which can be used to install, uninstall, activate, deactivate, or update. These functionalities are accessible by Auto Update Service. This is possible because of the class Update Manager. One module is represented by a class UpdateUnit this class is a kind of wrapper for different elements. The named elements are each represented by UpdateElement. The class provides the name category author, and other things of the module. For actions such as installing or uninstalling an OperationContainer instance. The instance comes with a factory method to each action. In short you can access all functions which your plugin manager provides using the Update Service API which is called Silent Updater in the project. Therefore, when you need to remove or add a module while running your program all you need to do I add or remove the module from the udates.xml layer which your update center uses to load components.

The video of dynamic load/unload - https://www.youtube.com/watch?v=k_r7kAw-wg0

I have not been able to get the program to continue running the NetBeans Platform after uninstalling my enemy, and therefore not been able to show the loading other than it loads all the

starting modules. I have not been able to figure out why it shuts down other than it gives me some error when handling the file as it seems to be a problem with it loading the file correctly after changes made to the file. It is however able to show that it can uninstall the enemy module after I delete the module in the updates.xml file.

OSGi Asteroid Game (OSGi Lab) – Project name in zip: PaxAsteroids (OSGi Project)

As with the NetBeans Module system I was not able to create an OSGi bundle and get them to talk with each other in time. However, I know that it is possible as I am using it in my semester project, and we have been able to get it to work though with a lot of hardship (we got it to work just before assignment deadline). So, for the purpose of this assignment and making it in time without going to much overboard I decided to use the PaxAsteroids example that was provided on the GitHub repository to be sure that I had something to show. As I also had troubles alone with the PaxAsteroids to work correctly as will be show in the video of dynamically load/unload as I am only able to load and unload enemy even though it finds all the other modules does it not find it as a listed bundle in the gogo shell. I believe the problem was that the dependency of the apache Felix wasn't added correctly or changes somehow. The video of dynamic load/unload - https://www.youtube.com/watch?v=OvpXOAkTq-Y

OSGi implement a complete dynamic component module, component comes in form of bundles which can remotely be installed, uninstalled, started, stopped, and updated. OSGi also uses provided service interface and the required service interface. The bundles have specific meta data which becomes packaged in specific way using the meta data so that you can manage, install, update, etc. your modules dynamically. The OSGi framework manages the life circle of the bundles in the order in which they are loaded. The OSGi framework is separate into separate layers the module, life, service, security, and declaratively services. The module layer defines the modularity framework on top the standard java platform, the life layer handles the life circle of the module, the service layer handles the service registry along with the lookup layer for exposing service for a module. This layer is also called the whiteboard model. The security layer build on top the java security layer and provides security to the bundles. The declarative services make it possible to declare dependencies between modules using dependency injection. This means that the OSGi Framework supports to 2 component modules the whiteboard and declarative dependency model.

When running the OSGi framework with Apache Felix I use Pax Provisions which is the OSGi service platform. Felix uses the configuration from a properties file and then scans two deployment-pox.xml files for bundles that it should download and install. Finding none will present a simple prompt and await orders. The first time running pax provision the pax plugin creates a runner directory under the OSGi-project which captures configuration of the runtime environment. This runner contains the deploy-pom.xml files the bundles which have been downloaded and runtime files required for Apache Felix. This means we can use the provided command line shell for OSGi called gogo, to control our bundles as can be seen in the dynamically load/unload video for OSGi.