Casper Fenger Jensen
Casje19@student.sdu.dk

Hand-in: 10-04-2022

A Component-based 2D Asteroids Game

The purpose of these exercises is to learn how to take apart a monolithic application and divide it into individual part called modules and how doing so makes it easier to remove, update and add modules to the system.

JavaLab

Resume of GameLab: in the GameLab exercise the original monolithic code base were divided into modules such as Core, Common, Player, etc. and when module A depends on module B, module B will be a dependency imported in the module A's pom.xml. The instantiation of a module B is done by calling the constructor directly in module A with "new ClassName();".

In JavaLab this instantiation/assembly is automated with Java's native service loader, to use this service loader the SPILocator class is provided. The SPILocator allows with the use of these three methods *(figure 1)* to get a collection with the implementations of a specific interface, instead of accessing an instance by referring to the value at which the instance from the "new ClassName();" call's return value is stored like done in the GameLab exercise.

```
private Collection<? extends IGamePluginService> getPluginServices() {
    return SPILocator.locateAll(IGamePluginService.class);
}

private Collection<? extends IEntityProcessingService> getEntityProcessingServices() {
    return SPILocator.locateAll(IEntityProcessingService.class);
}

private Collection<? extends IPostEntityProcessingService> getPostEntityProcessingServices() {
    return SPILocator.locateAll(IPostEntityProcessingService.class);
}
```

Figure 1 Game.java within the Core module.

These collections returned by the methods *(figure 1)* can then be loop thou with the use of a for loop *(figure 2)* and the individual implementations of these interfaces can then be accessed.

```
private void update() {
    // Update
    for (IEntityProcessingService entityProcessorService : getEntityProcessingServices()) {
        entityProcessorService.process(gameData, world);
    }
    for (IPostEntityProcessingService postEntityProcessorService : getPostEntityProcessingServices()) {
        postEntityProcessorService.process(gameData, world);
    }
}
```

Figure 2 Game.java within the Core module.

To publish an implementation of an interface for the SPILocator to include, one file per implemented interface is placed within the following path "src > main > resources > META-INF > services" in the module

that provides the implementation, this file is then named the path to the java class file *(figure 3)* that implements the interface and the context of the file is the same as the name.

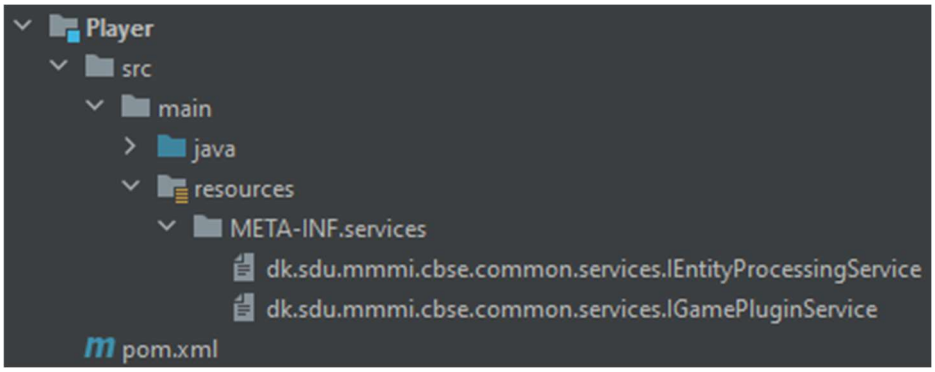

Figure 3 The structure of the Player module.

NetBeansLab1

There are one major issues with the code in JavaLab, that is in the way of load-/unloading dependencies in runtime. This being that the Core module depend in the pom.xml directly on the modules that implements the interfaces used in the Core module, this is a problem because the module needs to be build and reran for any changes in the pom.xml to take effect.

One way to get around this problem is to make the project use the NetBeans module system, and then depend on the "org-openide-util-lookup", from this dependency the previous SPILocator methods *(figure 1)* can be replaced with ones that uses the "Lookup" class instead *(figure 4)*.

```
private Collection<? extends IEntityProcessingService> getEntityProcessingServices() {
    return lookup.lookupAll(IEntityProcessingService.class);
}

private Collection<? extends IPostEntityProcessingService> getPostEntityProcessingServices() {
    return lookup.lookupAll(IPostEntityProcessingService.class);
}
```

Figure 4 Game.java within the Core module.

These look just like the SPILocator once but using the "Lookup" instead allows in the Core module's pom.xml file for the dependencies on the implementations of the interfaces to be moved into the application's pom.xml file meaning that the Core module does not have to be rebuild each time a new implementation of say the IEntityProcessingService is added to the project for this to be featured in the Core module.

To register implementations of an interface the implementation must use the "org.openide.util.lookup.ServiceProvider" above the class that implements an interface which other modules should be able to use this implementation. This is done by adding "@ServiceProvider()" just above the class and then parsing in "service = <implemented_interface>" *(figure 5)*, the implemented_interface here being the interface that the class implements.

```
import org.openide.util.lookup.ServiceProvider;

@ServiceProvider(service = IGamePluginService.class)
public class PlayerPlugin implements IGamePluginService {
```

Figure 5 PlayerPlugin.java in Player module.

Furthermore, are the interfaces placed in "Common*" modules, this allows multiple modules to require the same module that provides and implementation of the interface without the modules that requires the implementation needing to be aware of each other. These interfaces are made public for other modules to require/provide by listing so in the "Common*" modules pom.xml *(figure 6)*.

```
<plugin>
    <groupId>org.apache.netbeans.utilities</groupId>
    <artifactId>nbm-maven-plugin</artifactId>
    <version>4.3</version>
    <extensions>true</extensions>
    <configuration>
        <publicPackages>
            <publicPackage>dk.sdu.mmmi.cbse.commonenemy</publicPackage>
        </publicPackages>
    </configuration>
</plugin>
```

Figure 6 pom.xml in CommonEnemy module.

The implementations of an interface in a module that provides is made a providing module as shown in *(figure 7)*.

```
<plugin>
    <groupId>org.apache.netbeans.utilities</groupId>
    <artifactId>nbm-maven-plugin</artifactId>
    <version>4.3</version>
    <extensions>true</extensions>
    <configuration>
        <useOSGiDependencies>true</useOSGiDependencies>
    </configuration>
</plugin>
```

Figure 7 pom.xml in Enemy module.

NetBeansLab2

To make new implementations of an interface available it is no longer needed to list the module as a dependency in the module that requires it, it is only the "Common*" module and "Lookup" that are dependencies. To be able to load/unload modules in runtime, they also need to be removed from the application's pom.xml file and then injected into the system in some other way than within a pom.xml fil.

To do so an update center “updates.xml” list all the dependencies *(figure 8)* and a “SilentUpdate” module reads this file and loads/unloads dependencies in the “Lookup” as dependencies are added/removed in the “update.xml”

```
<module codenamebase="dk.sdu.mmmi.cbse.Player" distribution="Player-1.0-SNAPSHOT.nbm" downloadsize="0" l
    <manifest OpenIDE-Module="dk.sdu.mmmi.cbse.Player" OpenIDE-Module-Display-Category="dk.sdu.mmmi.cbse

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Enemy" distribution="Enemy-1.0-SNAPSHOT.nbm" downloadsize="0" lic
    <manifest OpenIDE-Module="dk.sdu.mmmi.cbse.Enemy" OpenIDE-Module-Display-Category="dk.sdu.mmmi.cbse"

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Asteroid" distribution="Asteroid-1.0-SNAPSHOT.nbm" downloadsize="
    <manifest OpenIDE-Module="dk.sdu.mmmi.cbse.Asteroid" OpenIDE-Module-Display-Category="dk.sdu.mmmi.cb

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Bullet" distribution="Bullet-1.0-SNAPSHOT.nbm" downloadsize="0" l
    <manifest OpenIDE-Module="dk.sdu.mmmi.cbse.Bullet" OpenIDE-Module-Display-Category="dk.sdu.mmmi.cbse

</module>

<module codenamebase="dk.sdu.mmmi.cbse.Collision" distribution="Collision-1.0-SNAPSHOT.nbm" downloadsize
    <manifest OpenIDE-Module="dk.sdu.mmmi.cbse.Collision" OpenIDE-Module-Display-Category="dk.sdu.mmmi.c

</module>
```

Figure 8 updates.xml

The syntax for declaring new implementations and requesting them is still the same as in the NetBeansLab1 exercise.