Casper Fenger Jensen
Casje19@student.sdu.dk

Hand-in: 09-05-2022

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.

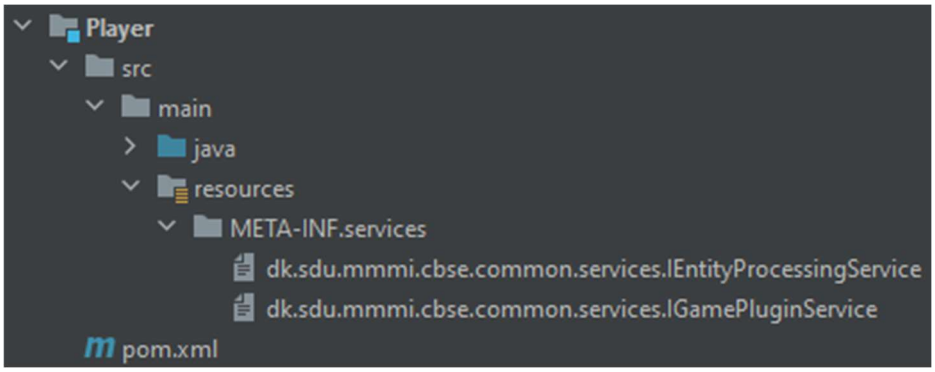

Figure 3 The structure of the Player module.

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.

DesignLab

The objective of the exercise is to learn about the couplings in the code architectures here by the “IntroLab” and “JavaLab”. Said with other words, it is learning about how the different parts of the two programs/systems is used together and how many different parts needs to be refactored to ably changes the parts that makes the program/system.

IntroLab

Number	Class	Depends on	Dependency Depth
1.	SpaceObject	Game	5
2.	Enemy	Game, SpaceObject	6
3.	Player	Game, SpaceObject	6
4.	PlayState	Player, Enemy, GameKeys, GameStateManager	3
5.	GameState	GameStateManager	2
6.	GameStateManager	GameState, PlayState	1
7.	GameKeys	-	0
8.	GameInputProcessor	-	0

9.	Game	GameInputProcessor, GameKeys, GameStateManager	4
10.	Main	Game	5

JavaLab

Number	Component/Library	Depends on	Dependency Depth
1.	Collision	Common	1
2.	Asteroid	Common	1
3.	Enemy	Common	1
4.	Player	Common	1
5.	Common	-	0
6.	Core	Common, Player, Enemy, Asteroid, Collision	2

Reflection

In the monolithic application "IntroLab" the dependencies are all over the place and the depth goes far were the "JavaLab" has a much flatter depth which looks a lot simpler to apply changes to since only one or a few files needs modification when remove, updating or added things to modules or completely new modules.

OSGILab

The code outcome from NetBeansLab2 makes it possible to load and unload modules in runtime by modifying a file, specifically the "update.xml" file. Another way to do the same task of load- and unloading modules is thou a command line interface (CLI). OSGI is standard for making such CLI and for this exercise Apache's implementation of the OSGI standard called Felix will be used.

The major changes that have been made is making a new OSGI project and then generating all the modules again because in the OSGI standard these are referred to as bundles. Then after creating all the bundles need the code were mostly just move over with a few changes such as removing the "Lookup" implementations that can be seen on *(figure 5)* and wrapping the "LibGDX" dependencies, this wrapper is then referred to in the bundles instead of directly importing "LibGDX".

Another major change is the none porting of the "SilentUpdate" as this is no longer needed and last thing beside adding the OSGI dependencies to the "pom.xml" files of each bundle, is to refracture the "*.xml" files in the "META-INF" folder two important things in the new files is the "service" tags where within the interface that this bundle contains an implementation of is listed *(figure 9)*.

```
<service>
    <provide interface="dk.sdu.mmmi.cbse.common.services.IEntityProcessingService"/>
</service>
```

Figure 9 enemyprocessor.xml dk.sdu.mmmi.cbse.enemy bundle

The other important thing in these files is the "reference" tag where the need of an implementation of a specific interface is listen along with the relation *(figure 10)*.

```
<reference bind="addPostEntityProcessingService" cardinality="0..n"
           interface="dk.sdu.mmmi.cbse.common.services.IPostEntityProcessingService"
           name="IPostEntityProcessingService" policy="dynamic" unbind="removePostEntityProcessingService"/>
```

Figure 10 core.xml in dk.sdu.mmmi.cbse.core bundle

To load and unload modules the CLI is used the “lb” (list bundles) command is used to display the status of all bundles. The “stop” and “start” commands followed by a bundle’s id is then used to load and unload bundles.

SpringLab

In this lab spring is used to manage the modules. The exercise builds on the code from “JavaLab”. The only module that has been modified is the “Core” module. Instead of using the “SPILocator” this has been removed along with the files in the “resource” folders for alle modules. The “Core” module has gotten an “ApplicationContext” from the Spring framework *(figure 11)*, within this instance the other modules is stored, and they can be accessed thou here.

```
public Game() {
    ApplicationContext context = new ClassPathXmlApplicationContext( configLocation: "Beans.xml");
    for (String name: context.getBeanDefinitionNames()) {
        System.out.println(name);
    }

    this.entityProcessors = new ArrayList<>(context.getBeansOfType(IEntityProcessingService.class).values());
    this.entityPlugins = new ArrayList<>(context.getBeansOfType(IGamePluginService.class).values());
    this.postEntityProcessors = new ArrayList<>(context.getBeansOfType(IPostEntityProcessingService.class).values())
```

Figure 11 Game.java in Core module

To add modules the “bean” tag is used in the “Beans.xml” file in the “resource” folder within the “Core” module where a name and a path to the java class are listed *(figure 12)*.

```
<bean id="playerPlugin" class="dk.sdu.mmmi.cbse.playersystem.PlayerPlugin" />
<bean id="playerControlSystem" class="dk.sdu.mmmi.cbse.playersystem.PlayerControlSystem" />

<bean id="enemyPlugin" class="dk.sdu.mmmi.cbse.enemysystem.EnemyPlugin" />
<bean id="enemyControlSystem" class="dk.sdu.mmmi.cbse.enemysystem.EnemyControlSystem" />

<bean id="asteroidPlugin" class="dk.sdu.mmmi.cbse.asteroid.AsteroidPlugin" />
<bean id="asteroidControlSystem" class="dk.sdu.mmmi.cbse.asteroid.AsteroidControlSystem" />

<bean id="collider" class="dk.sdu.mmmi.cbse.collision.Collider" />
```

Figure 12 Beans.xml in Core module

TestLab

This exercise is about unit testing and for this the code from the OSGILab and some dependencies from “junit” are used. First a test folder is made in the module that is going to be tested. The functions that needs to be ran is labeled with a “@Test” from “org.junit.jupiter.api”, the same as overriding a function *(figure 13)*.

```
@Test
public void testPlayerStartPosition() {
```

Figure 13 PlayerControlSystemTest in test in dk.sdu.mmmi.cbse.player

For this exercise two tests were made, one for testing the playership’s starting position and another for testing the movement.

In the first test where the starting position is tested, everything is cleared and then the playership is created, and then the function “assertEquals” from the “junit” dependency, is used to match the playership’s expected position for the X-value, Y-value and the rotation of the ship.

The second test where to test the movement of the ship, first everything is reset and the playership is created, the Y-value is then stored because it is known that the ship is facing in the vertical direction. A button press is then simulated over a time period and the new Y-value is matched with the old one to see if the ship has moved. The same is then done with the rotation.

Looking back at this, this last test seems to be fitting better as an input test, the thing that should have been done different is instead of just checking if the starting and ending values where different, to know if the ship has moved or not. The actual distance should have been compared instead to check if the ship moved correctly in comparation to the input given.

When the code is executed, an output is displayed in the console, as seen in the snippet *(figure 14)* both tests passed as there were no tests failed and none skipped.

```
-------------------------------------------------------
 T E S T S
-------------------------------------------------------
Running dk.sdu.mmmi.cbse.player.PlayerControlSystemTest
Running testPlayerMovement!
Running testPlayerStartPosition!
Tests run: 2, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.5 s - in dk.sdu.mmmi.cbse.player.PlayerControlSystemTest

Results:

Tests run: 2, Failures: 0, Errors: 0, Skipped: 0
```

Figure 14