Table of Contents

JavaLab

The source code for this project can be found under the folder “AsteroidsServiceLoader”.

What was the purpose

The JavaLab consisted of automating the implementation of player, Enemy, Asteroid, Bullet and Collision. Using the provided ServiceLoader helper class, SPILocator, to help with reducing the manual instantiations.

What was done in the code to access and register the components

The process of automating the instantiating of any component can be written as following:

1. In the Game class under Core where one previously instantiated by manually typing the **new** keyword for **every new instance** was replaced with a **Collection** where the **SPILocator.locateAll** method was used (This is how one access a given component)
2. For the SPILocator to know that a component uses a given service, a file in the **META-INF.service** package must be created (This is how one register a given component)

All in all, by using the ServiceLoader we can minimize the glue-code that are needed to instantiate new instances of classes. Though, this lab was not designed to be fully component-based and is only a step in the “right” direction.

One thing to point out on this lab implementation, the **bulletEntityPart** in the **Bullet** class should have been moved to its own Common class, like **CommenBullet**. That way it would also ensure that modules like **Asteroid** does not have any knowledge to bullet. As the saying goes *provide no more, expect no less.*

Code-snippet

The following code-snippet describes how the registration and instantiating works

```
(...)
public class Game
        implements ApplicationListener {

    (...)
    private final GameData gameData = new GameData();
    private List<IEntityProcessingService> entityProcessors = new
ArrayList<>();
    private List<IPostEntityProcessingService> postEntityProcessors = new
ArrayList<>();
    private World world = new World();

    @Override
    public void create() {

        (...)

        // Lookup all Game Plugins using ServiceLoader
        for (IGamePluginService iGamePlugin : getPluginServices()) {
            iGamePlugin.start(gameData, world);
        }
    }


        {...}

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

    {...}

    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);
    }
}
```

DesignLab

To get a clearer picture of the benefits of the component orientated architecture vs the monolithic architecture a dependency analysis will be performed on both implementations. The monolith will be on class level and the component orientated on component level. Both will be seen at build time.

AsteroidLibGDX – Monolith application

Firstly, a couplings table has been developed on the monolithic architecture game implementation, AsteroidsLibGDX. The analysis is performed on build time

Number	Class	Depends on	Dependency Depth
1	GameKeys	-	0
2	GameInputProcessor	-	0
5	Bullet	SpaceObject	4
6	Enemy	SpaceObject, Bullet	5
7	GameState	GameStateManager(C)	1
9	GameStateManager	GameState (C), PlayState(C)	1
10	Game	GameStateManager, GameKeys	2
3	SpaceObject	Game	3
4	Player	Game, SpaceObject	4
8	PlayState	GameStateManager(C), Player, Enemy, Bullet, GameKeys,	4

Most of the classes in the implementation have a circular dependency. Those that have one directly are marked with a 'C' and the depth is seen as one for the involved classes. The depth written in the analysis can be seen as infinite due to the circular dependency but for the purpose of understanding I've tried to find a finite number. The result of the above analysis is that the implementation has a high coupling due to the high dependency.

JavaLab – Component orientated implementation

The next analysis is performed on the more component orientated implementation from *JavaLab*.

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

One key aspect of the component orientated implementation is that there are no circular dependencies. In fact, a component-bases system cannot have a circular dependency due to how the architecture is designed (With the use of contracts and interfaces). The benefit of using

interfaces to communicate with other classes is first of all to limit the dependency but also to create contracts between the communicators thus ensuring that every part know what to expect of each other.

NetBeansLab1

The source-code can be found under the folder "NetBeansLab1"

The effect of the change

In NetBeansLab1 I've changed how we start the application. From preciously where we had a main-class we now have an installer-class that extends **ModuleInstall**. The difference here is that without a main class we enable the ability to hook into the life cycle of the runtime container at the startup. That means that the runtime container looks for any module with a class that extends the **ModuleInstall** and through there create something at the startup/runtime.

Another change that've been made compared to the precious lab (JavaLab) is how modules are registered. Instead of doing it manually through the **manifest-file/META-INF.service** it is now done with just an annotation, **@ServiceProvider**. The main benefit is the automation it provides. The tag is placed just before the start of a given class and contains one or more used interfaces.

What was done in the code itself

The following bullet points covers roughly what I've changed in this lab:

- Changed main to installer (A class that extends ModuleInstall)
- Added the @ServiceProvider annotation on components which removes the need of the META-INF.Service folder.
- The pom.xml file in the application folder needs dependencies on EVERY module
- The manifest file under nvm in the Core module needs to be "updated" for the Installer class
- The parent pom-file needs to have all modules
- To access the components annotated with the **@ServiceProvider** annotation a **lookup** instance provided by java.

Code-snippet

The following code-snippet describes how the **@ServiceProvider** annotation works.

```
{...}

ServiceProviders(value = {
    @ServiceProvider(service = IEntityProcessingService.class),
    @ServiceProvider(service = BulletSPI.class)})
public class BulletControlSystem implements IEntityProcessingService,
BulletSPI
{

    //Some Code
}
```

The following code-snippet describes how components are accessed

```
{...}

public class Game implements ApplicationListener {

    //Some attributes


    private final Lookup lookup = Lookup.getDefault();//To retrieve the
default instance of the java Lookup
    private List<IGamePluginService> gamePlugins = new
CopyOnWriteArrayList<>();
    private Lookup.Result<IGamePluginService> result;

    @Override
    public void create() {
        //Some code

        //The lookup for components using the IGamePluginService
        result = lookup.lookupResult(IGamePluginService.class);
        result.allItems();

        //To get and call the start-method on components using the
IGamePluginService
        for (IGamePluginService plugin : result.allInstances()) {

            plugin.start(gameData, world);
            gamePlugins.add(plugin);
        }
    }

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

    // Post Update
    for (IPostEntityProcessingService postEntityProcessorService :
getPostEntityProcessingServices()) {
        postEntityProcessorService.process(gameData, world);
    }
}


    (...)

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

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

NetBeansLab2

The source-code can be found in the folder "AsteroidsNetbeansModules"

Demonstration of NetBeans Dynamic Update

A video that demonstrates the dynamically install and uninstall feature can be found in the main folder submitted to ItsLearning. If it is unplayable, it can be seen through this link (A private YouTube link)

The effect of the change

In NetBeansLab2 the focus was on dynamically installing and uninstalling modules when the application was running. That was done through the **update.xml** file which were extracted after the application was build.

The dynamically installation and uninstallation of modules is, as said before, done through the **update.xml** but that is only to tell what modules should be affected. The actual "logic" that controls the functionality is through the **SilentUpdate** module.

What was done in the code itself

The following bullet points covers roughly what I've changed in this lab:

- Clean install with Development profile checked and deployment in profiles – to know the modules that might be needed
- Move netbeans_site from target to a static place. Take the file-path for update.xml to SilentUpdate's bundle.properties
- New clean install without development checked.
- Run the application – The update.xml can be used to dynamically update the project
- The **@ServiceProvider** annotation that was used in NetBeansLab 1 was reused here as well with the same benefits.

How the SilentUpdate works

The following describes how the dynamic update functionality works with a high degree of abstraction:

It is **silentUpdate** that deals with **updating**, **uninstalling** and **installing** through three appropriate methods in **UpdateHandler** class (A provided API through the NetBeansModule system).

The autoServiceAPI checks if anything new has been installed through the unique name on the module itself and its version number

Through the container for e.g uninstall it checks if the module can be uninstalled without affecting other modules (Other modules that dependends on that module)

The **lookup** method is "patiently listening" for any changes through the **lookupListener** and automatically updates the game logic if anything is installed or uninstalled.

Code-snippet

The following code-snippet describes how the **@ServiceProvider** annotation works.

```
{...}

ServiceProviders(value = {
    @ServiceProvider(service = IEntityProcessingService.class),
    @ServiceProvider(service = BulletSPI.class)})
public class BulletControlSystem implements IEntityProcessingService,
BulletSPI
{

    //Some Code
}
```

The following describes how components are accessed and handled if changes happen

```
{...}

public class Game implements ApplicationListener {

    //Some attributes


    private final Lookup lookup = Lookup.getDefault();//To retrieve the
default instance of the java Lookup
    private List<IGamePluginService> gamePlugins = new
CopyOnWriteArrayList<>();
    private Lookup.Result<IGamePluginService> result;

    @Override
    public void create() {
        //Some code

        //The lookup for components using the IGamePluginService
        result = lookup.lookupResult(IGamePluginService.class);
        result.addLookupListener(lookupListener);
        result.allItems();

        //To get and call the start-method on components using the
IGamePluginService
        for (IGamePluginService plugin : result.allInstances()) {

            plugin.start(gameData, world);
            gamePlugins.add(plugin);
        }
    }

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

    // Post Update
    for (IPostEntityProcessingService postEntityProcessorService :
getPostEntityProcessingServices()) {
        postEntityProcessorService.process(gameData, world);
    }
}



    (...)

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

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

OSGi

The source-code can be found in the "PaxAsteroids" folder.

In OSGi lab the focus was to get an understanding of how to register dependencies. One was through bundle context and the other was through dependency injection. Furthermore, the goal was to use Apache gogo shell to start and stop OSGi bundles.

Both versions of dependencies were used on the same component. Normally you use one "method" on one component but in this example, it was only to show the concept.

Bundle Context

Bundle context was implemented on the **player component**. More specifically on the **PlayerPlugin** class. An **Activator** class that **implements BundleActivator** was created in the player component. The Activator class is used to create a new instance of the given class, in this example a new instance of the **PlayerPlugin** and through the **BundleContext** datatype register the "service" (Interface)

This method removes the need for a **META-INF.service** file as seen in the previously examples. Though a new type of file is needed, the **osgi.bnd** file. The main use of this file is to specify where the "activator" class is located.

Dependency Injection through Declarative services

Dependency injection was also implemented on the **player component** but instead on the **PlayerControlSystem**. Here two methods were created; One to add and one to remove the **bulletService**. In contrast to bundle context with this approach there is a need for an **.xml** file where the needed service is **provided,** the "implementation class" is specified and any references are specified as well. References as being what to "inject". The service was **IEntityProcessingService** and the reference was **BulletSPI**.

Dynamically load and unload through Apache gogo shell

A video that demonstrates the dynamically install and uninstall feature can be found in the main folder submitted to ItsLearning. If it is unplayable, it can be seen through this link (A private YouTube link)

Code-snippets

The following code-snippet is how the bundle-context was implemented using an Activator class

```
public class Activator implements BundleActivator {

    @Override
    public void start(BundleContext context) throws Exception {

        PlayerPlugin playerPlugin = new PlayerPlugin();
        context.registerService(IGamePluginService.class, playerPlugin,
null);
    }

    @Override
    public void stop(BundleContext context) throws Exception {
    }

}
```

For the Bundle-context to work a **.bnd** file must be declared specifying the path to the activator class

```
#------------------------------------------------------------------
# Use this file to add customized Bnd instructions for the bundle
#------------------------------------------------------------------
Bundle-SymbolicName: OSGiPlayer
Bundle-ActivationPolicy: lazy
Bundle-Activator: dk.sdu.mmmi.osgiplayer.Activator
Service-Component: META-INF/entityprocessor.xml
```

The following code-snippet is how the dependency injection was implemented in the **PlayerControlSystem** class

```
public class PlayerControlSystem implements IEntityProcessingService {

    private BulletSPI bulletService;

        //Some Code

    //TODO: Dependency injection via Declarative Services
    public void setBulletService(BulletSPI bulletService) {
        this.bulletService = bulletService;
    }

    public void removeBulletService(BulletSPI bulletService) {
        this.bulletService = null;
    }

}
```

And then the specification in the **xml** file under **META-INF**

```
<?xml version="1.0" encoding="UTF-8"?>
<scr:component xmlns:scr="http://www.osgi.org/xmlns/scr/v1.1.0"
name="dk.sdu.mmmi.osgiplayer.processor">
    <implementation class="dk.sdu.mmmi.osgiplayer.PlayerControlSystem"/>
    <service>
        <provide
interface="dk.sdu.mmmi.cbse.common.services.IEntityProcessingService"/>
    </service>

    <reference bind="setBulletService" cardinality="0..1"
               interface="dk.sdu.mmmi.cbse.common.bullet.BulletSPI"
               name="BulletService" policy="dynamic"
unbind="removeBulletService"/>
</scr:component>
```

It is specified in the **implementation** tag what class is implementing the interface specified in the **service** tag. And lastly in the **reference** tag the method used to bind and unbind the BulletSPI.

SpringLab

The source-code for this project can be found in the "SpringLab" folder.

In SpringLab the focus was to get familiar with the Spring container and the component-oriented features of the Spring framework.

The implementation took starting point from *JavaLab's* implementation though only **Player, Core** & **Common** was ported to the Spring framework for simplicity.

Where we previously in *JavaLab* had to manually create the **META-INF-service** files, - that specifies which interfaces a given class uses - we use the **@Component** annotation provided by Spring instead. With the use of that annotation, we can also skip any manually **.xml** files.

To load any services, we use an instance of the **AnnotationConfigApplicationContext.** On that instance, through an foreach loop, the **getBeansOfType** method is called to get a given bean (A component).

What was done in the code itself

The following bullet points covers roughly what I've changed in this lab:

- **@Component** on each class that implements an SPI (from Common)
    - The annotation makes the class a Spring Bean
- In **Game**, instead of using the Lookup/serviceloader, we use spings version, **AnnotationConfigApplicationContext (Called context)** and on that instance we use the **getBeansOfType** and then the type of the interface to start or process the **bean**
- In **Game** we also specify what **bacePackage** the **Spring context** needs to look for. That ensures that Spring does not have to traverse through the entire project, but only where we know classes are located.

- When we have the **interface instance** through the map, we call the appropriate method for that **interface**, e.g., **start** or **process**.

Code-snippet

The following code-snippet shows how components are registered in Spring

```
import org.springframework.stereotype.Component;

@Component
public class PlayerControlSystem implements IEntityProcessingService {

        //Some code

}
```

So simply by adding the annotation **@Component,** Spring knows that this class should be treated as one.

The following code-snippet shows how registered components are registered

```
import
org.springframework.context.annotation.AnnotationConfigApplicationContext;

public class Game
        implements ApplicationListener {

        //Some code
    private AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext();

    @Override
    public void create() {

        //Some code

        context.scan("dk.sdu.mmmi.cbse");
        context.refresh();

        for (Map.Entry<String, IGamePluginService> iGamePluginServiceEntry :
                context.getBeansOfType(IGamePluginService.class).entrySet()){
            System.out.println(iGamePluginServiceEntry);

            iGamePluginServiceEntry.getValue().start(gameData, world);
        }
    }

        //Some Code

    private void update() {
        // Update
        for(Map.Entry<String,IEntityProcessingService>
IEntityProcessingService :

context.getBeansOfType(IEntityProcessingService.class).entrySet()){

            IEntityProcessingService.getValue().process(gameData,world);
        }

    }

        //Some code


}
```

An **AnnotationConfigApplicationContext** instance is declared as an attribute and serves as the 'keyhole' to access the registered components in Spring. As seen in the **create** method a **.scan** and **.refresh** method are called on the context which specified where spring has to search for components, the base package, and the update the context with the found components.

Then to access components an **entry set** from the **context** with a specified interfaced is requested and through there the instance of the component, he value in the map, can be accessed

TestLab

The source-code for this project can be found in the "TestLab" folder.

One key aspect of coding software is to verify that the code does what is intended. That is verified through tests.

The test lab was more about testing then registering and accessing modules. The test was developed on the Bullet-module's **BulletControlSystem** class and verified the creation of bullets when a **shooter** shoots and a bullet moves after its creation.

The below code-snippet shows the two implemented test. The first one, **bulletCreation**, tests the creation of a bullet using the method **createBullet**. As seen in the **createBullet** method a new player is created with a given position though as an **Entity** and not as the specified version, **player**. That is due to the limitation in terms of access and dependencies. The **second test** verifies that the bullet does indeed move and is not just created. Once again, the **createBullet** method is called to create an instance of the bullet, the initial position is extracted and after the **.process** method has been called once the position is extracted once again and compared to the initial position.

Code-snippet

```
Import …

public class BulletTest {
    private final GameData gameData = new GameData();
    private final World world = new World();
    private final Entity player = new Entity();

    private Entity bullet = null;
    private BulletControlSystem bulletControlSystem = null;

    BulletTest(){

    }

    public void createBullet(){
        //We cannot create a new Player instance due to the dependencies
        player.add(new PositionPart(33, 44, 55));
        //Creates a new bullet
        this.bulletControlSystem = new BulletControlSystem();
        this.bullet = this.bulletControlSystem.createBullet(player,
gameData);
    }

    @Test
    public void bulletCreation(){
        createBullet();
        this.bulletControlSystem.process(gameData, world);
        assertNotNull(bullet);
    }

    @Test
    public void bulletMovement(){
        createBullet();
        this.bulletControlSystem.process(gameData, world);
        //To test if the bullet has been drawn
        assertNotEquals(this.bullet.getRadius(), 0);

        PositionPart positionPart = this.bullet.getPart(PositionPart.class);
        float positionX = positionPart.getX();
        float positionY = positionPart.getY();

        world.addEntity(bullet);

        //To ensure that the bullet moves
        bulletControlSystem.process(gameData, world);
        assertNotEquals(positionX, positionPart.getX());
        assertNotEquals(positionY, positionPart.getY());
    }
}
```