Anders Kirsby Thygesen
anthy16@student.sdu.dk
11-04-2018

Software Components

Introduction

All assignments described in this report, are based on projects from the following repository: https://github.com/sweat-tek/SB4-KOM-F18. Elements have been added to each project, in accordance with the assignment descriptions found in the course plan.

Assignment 3: JavaLab - week 8

In this assignment, an SPILocator class is added to the project, to automate component assembly. The SPILocator class finds implementations of the projects main interfaces - IGamePluginService and IEntityProcessingService.

These interfaces are seen as services. The classes implementing these services - in this case classes that are part of Player and Enemy - are seen as service providers, because they provide an implementation of the service.

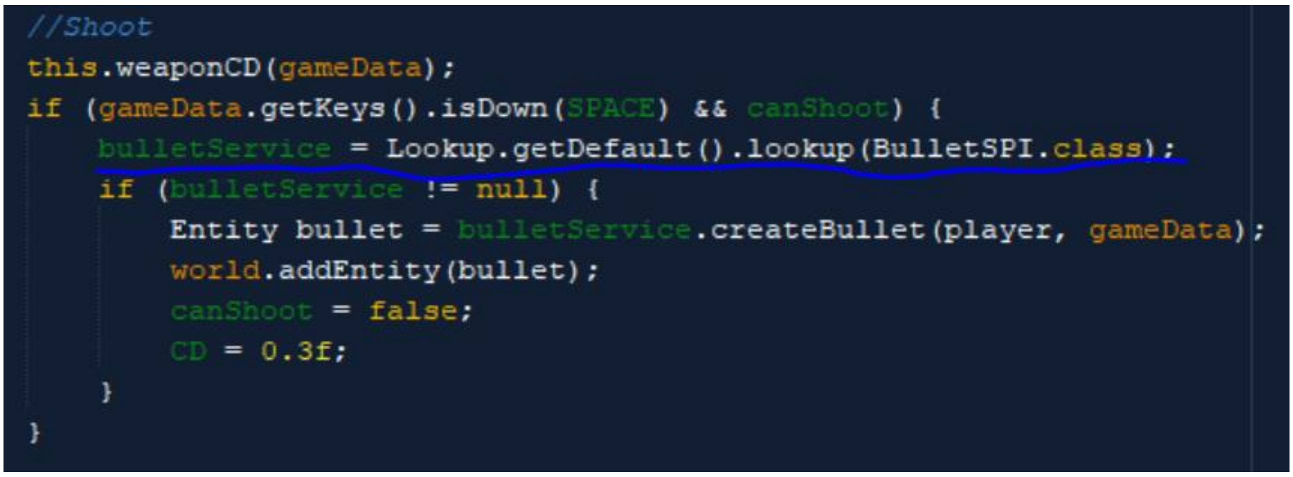
These service providers can be recognized by looking at their respective META-INF folders, located in ../src/main/resources/META-INF/services.

Anders Kirsby Thygesen
anthy16@student.sdu.dk
11-04-2018

Assignment 5: NetbeansLab 1 & 2 - week 9 & 10

Week 9 and 10 have been merged, as the assignment presented in week 10 essentially qualifies for all the necessary requirements presented in the assignment for week 9. A (not so) minimal run-time container is running in week 10, the previous modules have been ported over, and the lookup feature is also used, as described below.

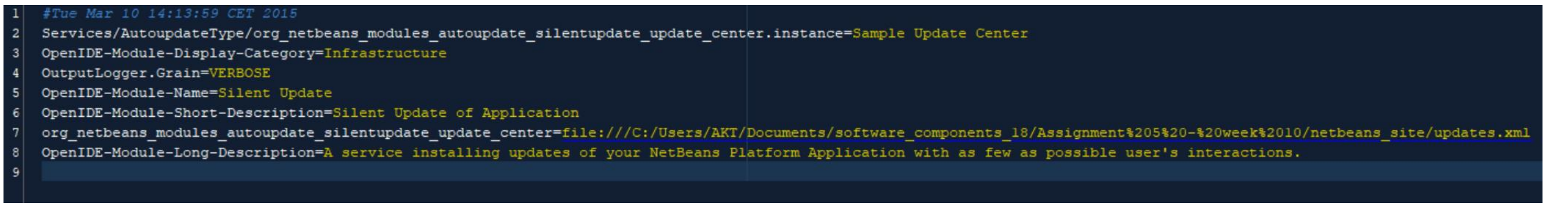
In this assignment, all classes from previous assignments, and a couple of new ones, have been ported over, and a Lookup feature has been used to find service providers. An example can be seen below:



```
//Shoot
this.weaponCD(gameData);
if (gameData.getKeys().isDown(SPACE) && canShoot) {
    bulletService = Lookup.getDefault().lookup(BulletSPI.class);
    if (bulletService != null) {
        Entity bullet = bulletService.createBullet(player, gameData);
        world.addEntity(bullet);
        canShoot = false;
        CD = 0.3f;
    }
}
```

Then, an update center Is generated. The following steps have been taken to do so:

1) AsteroidsNBModules-app is put in deployment mode.
2) SilentUpdate is removed as a dependency.
3) All necessary dependencies are added to AsteroidsNBModules-app. These include modules such as Core, Common, Player, Enemy, and so on.
4) Clean and build is run on AsteroidsNBModules-app.
5) The resulting netbeans_site folder is copied from ../application/target/netbeans_site and placed outside the root of the project.
6) AsteroidsNBModules-app is set back into default config.
7) The previous dependencies are removed from AsteroidsNBModules-app (Player, Enemy, etc.).
8) SilentUpdate is added to AsteroidsNBModules-app as a dependency.
9) The path to the updates.xml file (found in the netbeans_site folder) is added to Bundle.properties located in SilentUpdate:



```
#Tue Mar 10 14:13:59 CET 2015
Services/AutoupdateType/org_netbeans_modules_autoupdate_silentupdate_update_center.instance=Sample Update Center
OpenIDE-Module-Display-Category=Infrastructure
OutputLogger.Grain=VERBOSE
OpenIDE-Module-Name=Silent Update
OpenIDE-Module-Short-Description=Silent Update of Application
org_netbeans_modules_autoupdate_silentupdate_update_center=file:///C:/Users/AKT/Documents/software_components_18/Assignment%205%20-%20week%2010/netbeans_site/updates.xml
OpenIDE-Module-Long-Description=A service installing updates of your NetBeans Platform Application with as few as possible user's interactions.

```

10) Clean and build is run again.

And now, modules can be loaded/unloaded dynamically - while the app runs. This is done by removing (or commenting) the specific module from the updates.xml file found in the netbeans_site folder.

Anders Kirsby Thygesen
anthy16@student.sdu.dk
11-04-2018

Assignment 6: OSGiLab - week 11

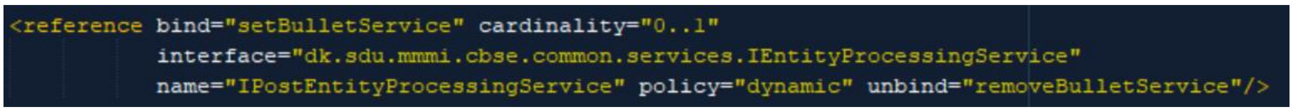
In this assignment, the same components from NetbeansLab 2 are ported over - this time as OSGiBundles instead of modules.

The functionality remains the same after the port is done - the only, or at least primary, difference is the way bundles are loaded.

Loading and unloading bundles at runtime does sometimes cause some issues. The OSGiExtender contains dependencies to all bundles, which presumably sometimes causes a reference error. Sometimes the wrong bundle gets unloaded and sometimes an error is thrown when reloading a bundle. A fix may include removing these dependencies so that the OSGiExtender only depends on the common libraries, Core and the OSGiUpdater (this has been tested a bit, but doesn't completely resolve the issue).

Declarative services and the BundleContext API are used, as described in the assignment.

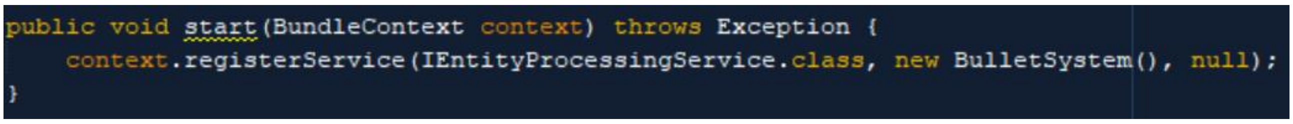
An example of the use of declarative services can be seen in the META-INF folder of OSGiEnemy. Here, a file called entityprocessor.xml is located - a snippet of it can be seen below:



```
<reference bind="setBulletService" cardinality="0..1"
           interface="dk.sdu.mmmi.cbse.common.services.IEntityProcessingService"
           name="IPostEntityProcessingService" policy="dynamic" unbind="removeBulletService"/>
```

This is dependency injection via declarative services.

An example of the BundleContext API in use can be seen in the Bullet class located in OSGiBullet:



```
public void start(BundleContext context) throws Exception {
    context.registerService(IEntityProcessingService.class, new BulletSystem(), null);
}
```

Here, the IEntityProcessingService service is registered with the BulletSystem acting as a service provider.

Anders Kirsby Thygesen
anthy16@student.sdu.dk
11-04-2018

Assignment 7: Design - week 12

- **Create a coupling table for the first monolithic Asteroids game implementation**

The first part of this assignment consists of creating a coupling table for the monolithic Asteroids game from IntroLab. External libraries such as LibGDX or Java libraries are not considered dependencies. The table can be seen below:

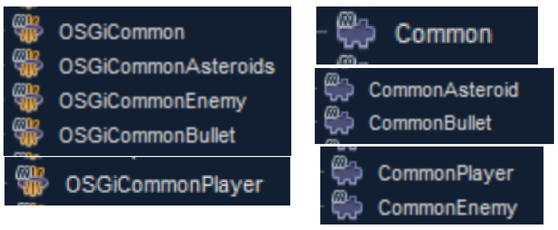
Number	Class	Depends on	Dependency depth
1.	GameKeys	-	0
2.	SpaceObject	Game	0
3.	Enemy	Game, SpaceObject	1
4.	GameInputProcessor	GameKeys	1
5.	Player	Game, SpaceObject	1
6.	Projectile	SpaceObject	1
7.	GameState	GameStateManager	2
8.	PlayState	Enemy, Player, Projectile, GameKeys, GameStateManager	2
9.	GameStateManager	GameState, PlayState	3
10.	Game	GameInputProcessor, GameKeys, GameStateManager	4
11.	Main	Game	5

Creating a coupling table for this system is tricky. Dependencies are so entangled, and coupling is so high that "loops" tend to happen.

For example - GameStateManger depends on GameState and PlayState. PlayState depends on the three entity classes (Player, Enemy, etc.), GameKeys AND GameStateManager. A dependency depth cannot be resolved from this. This obviously displays a problematic architecture.

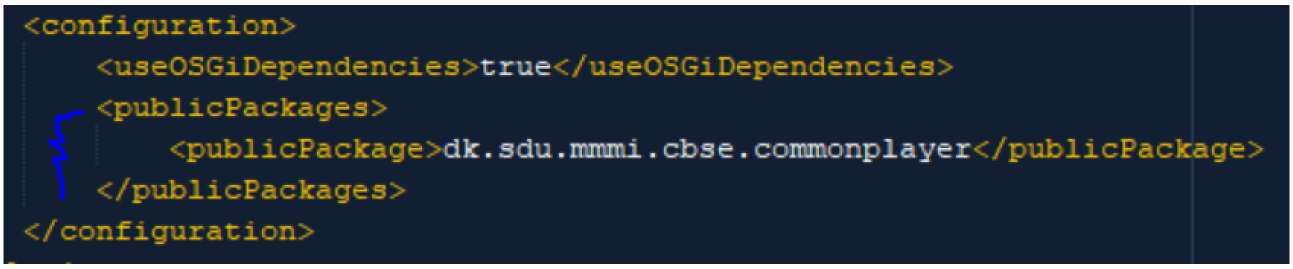
- **Refactor the entity classes into a separate common library**

Common libraries have been created in both NetbeansLab 2 and OSGiLab.



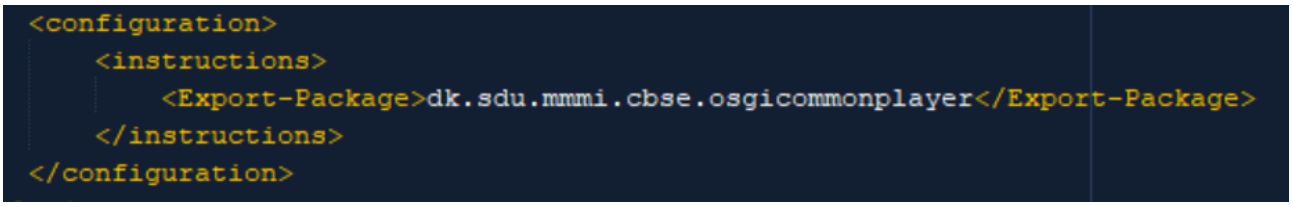
These are almost identical in setup and purpose. An example of this is the common library for player:

In NetbeansLab 2, CommonPlayer is used as a common library for the player entity. The library only contains one class: Player - a class that extends Entity. It should also be noted that for other modules to access packages within the common library, the following line is added to the pom.xml config:



```
<configuration>
    <useOSGiDependencies>true</useOSGiDependencies>
    <publicPackages>
        <publicPackage>dk.sdu.mmmi.cbse.commonplayer</publicPackage>
    </publicPackages>
</configuration>
```

In OSGiLab, the same common library is called OSGiCommonPlayer. This library also only contains an extension of Entity called Player. OSGi does not require packages to be made public, instead they are exported:



```
<configuration>
    <instructions>
        <Export-Package>dk.sdu.mmmi.cbse.osgicommonplayer</Export-Package>
    </instructions>
</configuration>
```

- **Group a set of classes that provide independent game functionality into separate components**

Components are grouped by functionality, so that no component contains functionality from different domains. For example, the enemy component does not contain classes related to bullet.

- **Expose game functionality through interfaces**

This is done via the provided interfaces. These are interfaces such as:

- IEntityProcessingService
- IGamePluginService
- IPostEntityProcessingService
- BulletSPI
- IAsteroidSplitter

- **Dependency analysis**

A coupling table has already been made for the monolithic Asteroids game. Now, a coupling table is created for the Asteroids game from NetbeansLab 2. Instead of looking at the depth of individual classes, entire modules are analysed:

Number	Class	Depends on	Dependency depth
1.	SilentUpdate	-	0
1.	Common	-	0
1.	CommonAsteroid	Common	1
1.	CommonBullet	Common	1
1.	CommonPlayer	Common	1
1.	CommonEnemy	Common	1

Anders Kirsby Thygesen
anthy16@student.sdu.dk
11-04-2018

1.	Core	Common	1
1.	app	SilentUpdate	1
1.	Enemy	Common, CommonBullet, CommonEnemy	2
1.	Bullet	Common, CommonBullet	2
1.	AsteroidSplitter	Common, CommonAsteroid	2
1.	Player	Common, CommonBullet, CommonPlayer	2
1.	Asteroid	Common, CommonAsteroid	2
1.	Collision	Common, CommonAsteroid, CommonBullet, CommonEnemy, CommonPlayer	3

The module dependencies are much cleaner, compared to the monolithic systems. First of all, dependencies only exist to common libraries. Second of all, the deepest dependency depth is 3 - which is found in Collision. This could likely be minimized to 2 with better implementation (the current dependencies on the common entity libraries are a bit redundant - the code could definitely be optimized).

No “looping” dependencies exist either. Instead, mostly everything leads back to the Common library.

This means that all modules, except the common libraries, can be removed (unloaded) and replaced without breaking the system itself. Compared to the monolithic system, this makes a big difference. If I removed the Player class from the monolithic system, PlayState would break - and with it GameStateManager, Game and essentially the entire application.

- **Reflection**

The level of abstraction that component-oriented design allows for, makes it much easier to “decode” large systems. The ability to “hot-swap” components by installing/uninstalling them during runtime is probably also a useful feature for a large system that may need to run 24/7.

That being said, the setup does seem large and tedious to do. There’s a lot of overhead in the shape of manifests, xml files, etc. and a lot seems like it can go wrong. This is likely something that will get easier with experience.

If I were to work with component-oriented design in Java again, I would likely work with OSGi.