Министерство науки и высшего образования Российской Федерации
федеральное государственное автономное образовательное учреждение высшего образования
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО»

Отчет
отчет по производственной технологической практике

Место прохождения практики
«Софтверке»

Авторы: Пушкарев Глеб Андреевич

Группа: M3335

Факультет: ФИТиП

Санкт-Петербург 2021

Ссылка на git репозиторий

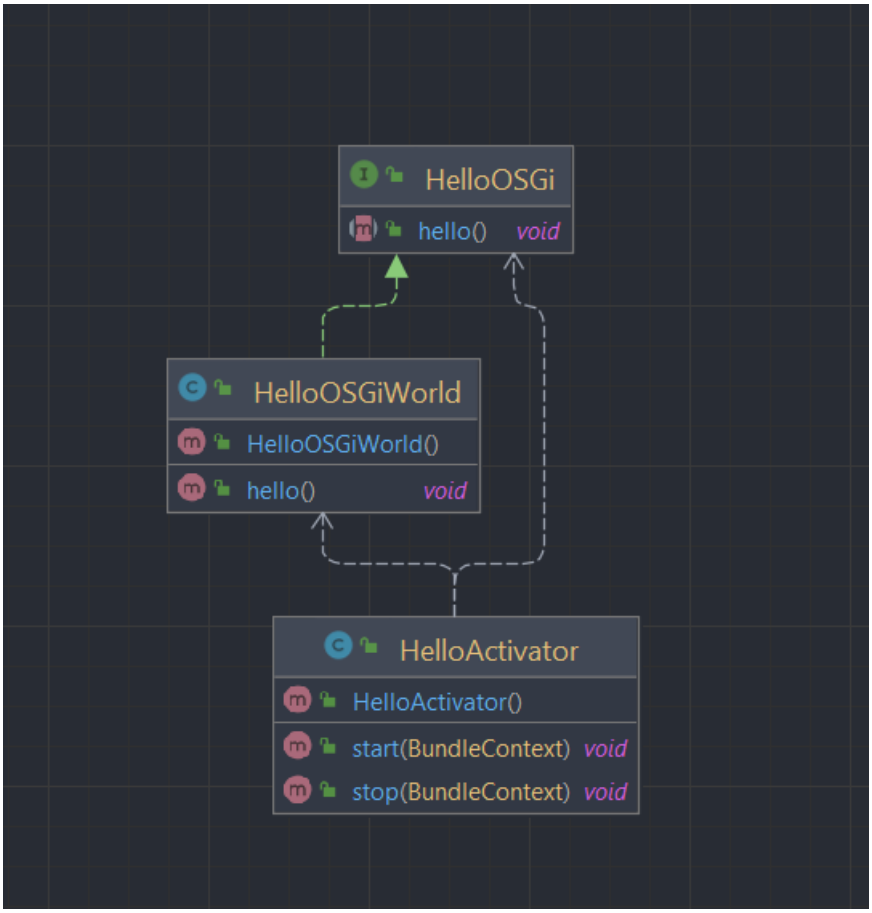
https://github.com/NelosG/softwerke-internship

Описание результатов по каждому этапу

- **Этап 1: Подготовительный**
  Были прочитаны первые 4 главы книги «OSGi in Action», в результате чего было получено общее понимание работы OSGi и информация о том, как с ним работать.

- **Этап 2. Реализация OSGi-сервиса**
  Был создан бандл содержащий интерфейс сервиса, активатор бандла и реализацию сервиса, которую активатор регистрирует в bundle context фреймворка.
  Интерфейс предоставляет только одну функцию: *hello()*
  Бандл экспортирует только сам интерфейс.
  Реализация сервиса при вызове функции *hello()* вывод на консоль «*Hello OSGi World!*».

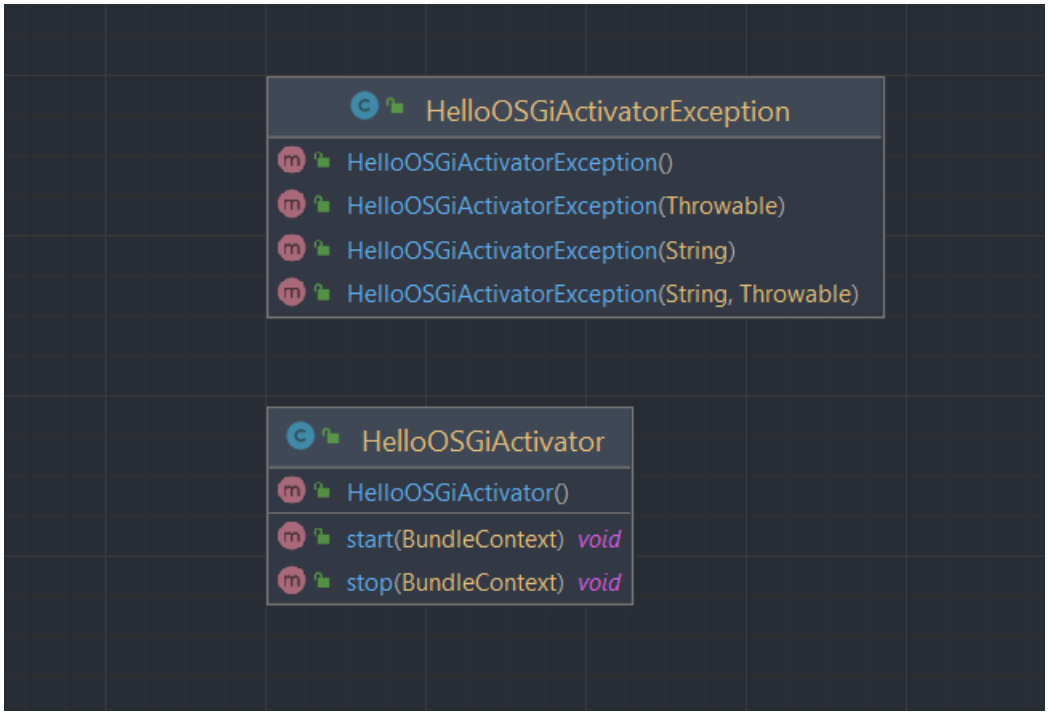
  Ниже приведен график данного бандла (здесь и далее на графиках зеленый цвет означает наследование, а серый использование).



модуль ru.ifmo.pga.hello.service
бандл pga_hello_osgi_service

Модуль *ru.ifmo.pga.hello.service* имеет зависимости:

1. *osgi.core* – необходим для использования классов OSGi (BundleActivator)
2. *maven-bundle-plugin* - Необходим для сборки (создает корректный манифест)

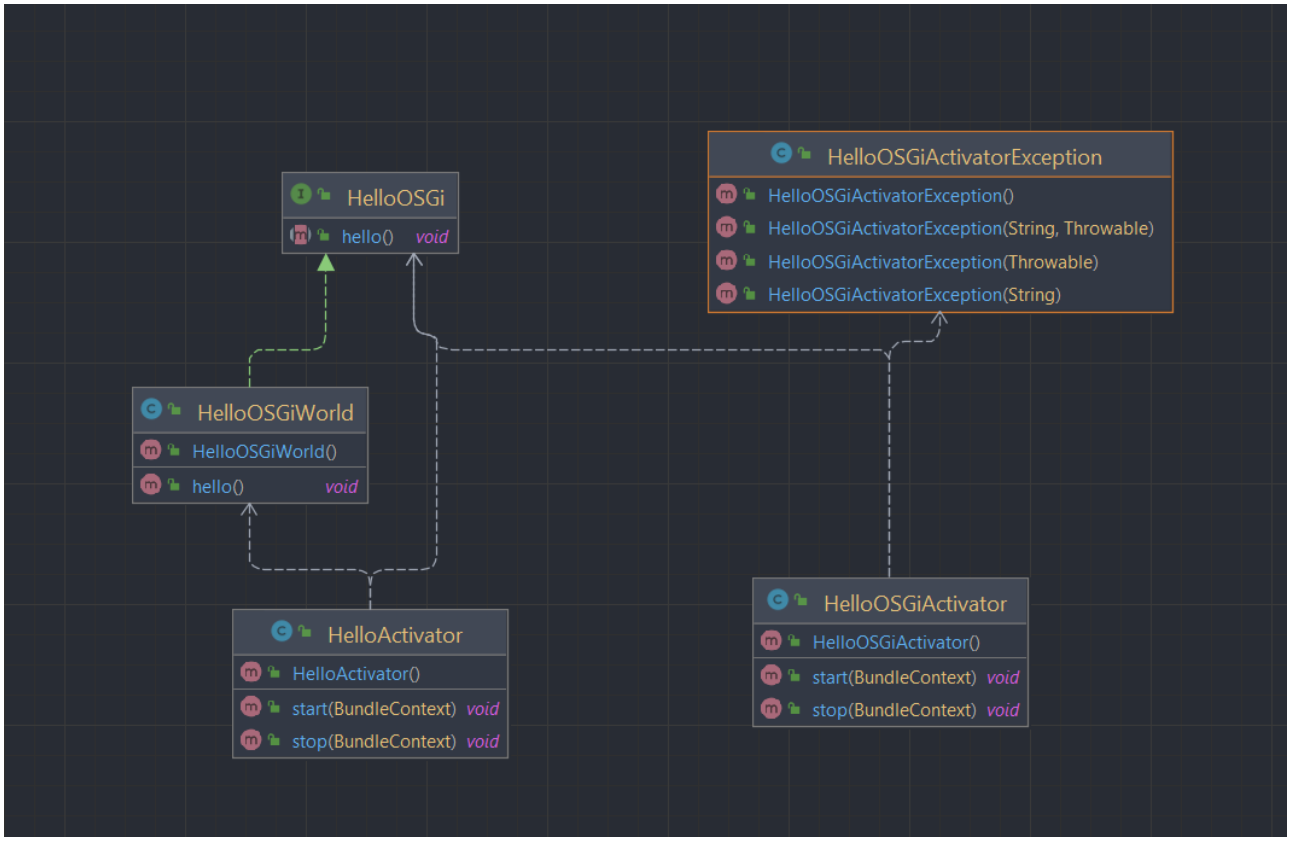


модуль ru.ifmo.pga.hello.activator
бандл pga_hello_osgi_activator

Модуль *ru.ifmo.pga.hello.activator* содержит реализацию клиента который вызывает метод *hello()* при вызове функции *start()* .

Модуль *ru.ifmo.pga.hello.activator* имеет зависимости:

1. *osgi.core* – необходим для использования классов OSGi (*BundleActivator*)
2. *maven-bundle-plugin* - Необходим для сборки (создает корректный манифест)
3. *ru.ifmo.pga.hello.service* – Предоставляет сервис *HelloOSGi*



Общая схема Этапа 2

- **Этап 3. Apache Felix Service Component Runtime**
  Аналогично этапу 2 был создан сервис, реализация которого выводит на консоль «Goodbye OSGi World!», правда уже не на моменте активации бандла, а во время активации сервиса.

  Были использованы SCR аннотации, которые заметно упростили написание кода.
  Если раньше нам было необходимо самостоятельно регистрировать компонент в бандл контексте, то теперь мы с помощью аннотаций говорим OSGi реализацией какого сервиса является данный класс.

```
/**
 * Singleton component, that implements {@link GoodbyeOSGi} service.
 */
@Component(
        service = GoodbyeOSGi.class,
        immediate = true
)
```

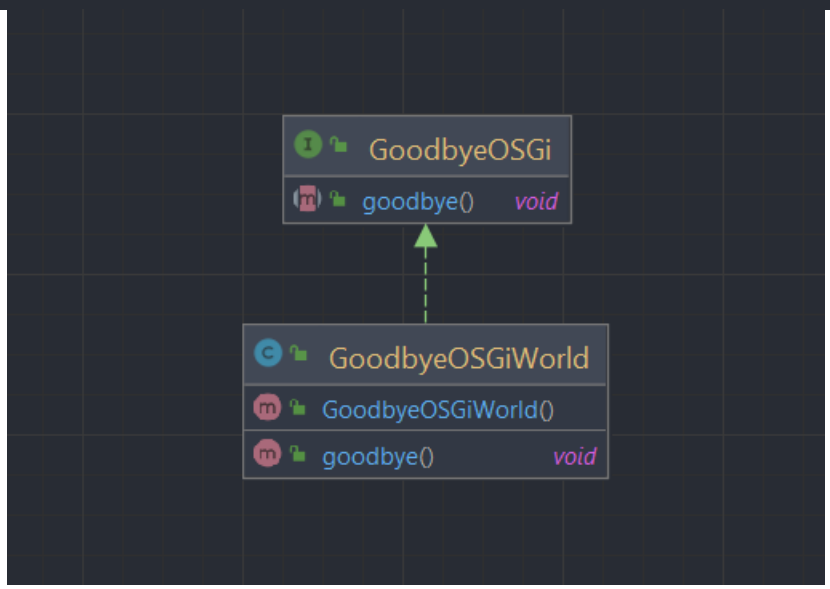


Схема модуля

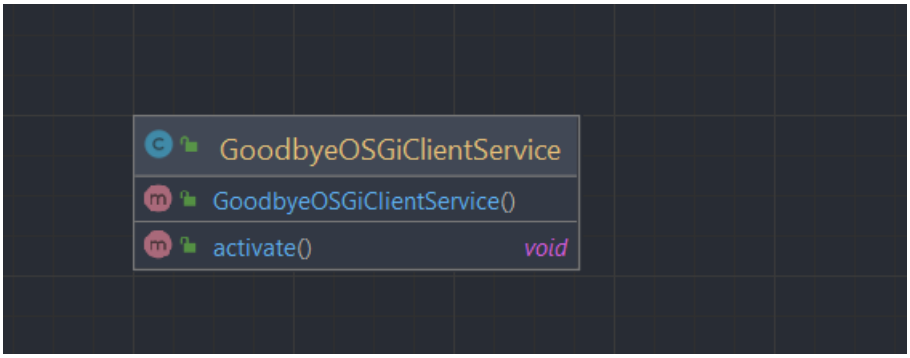


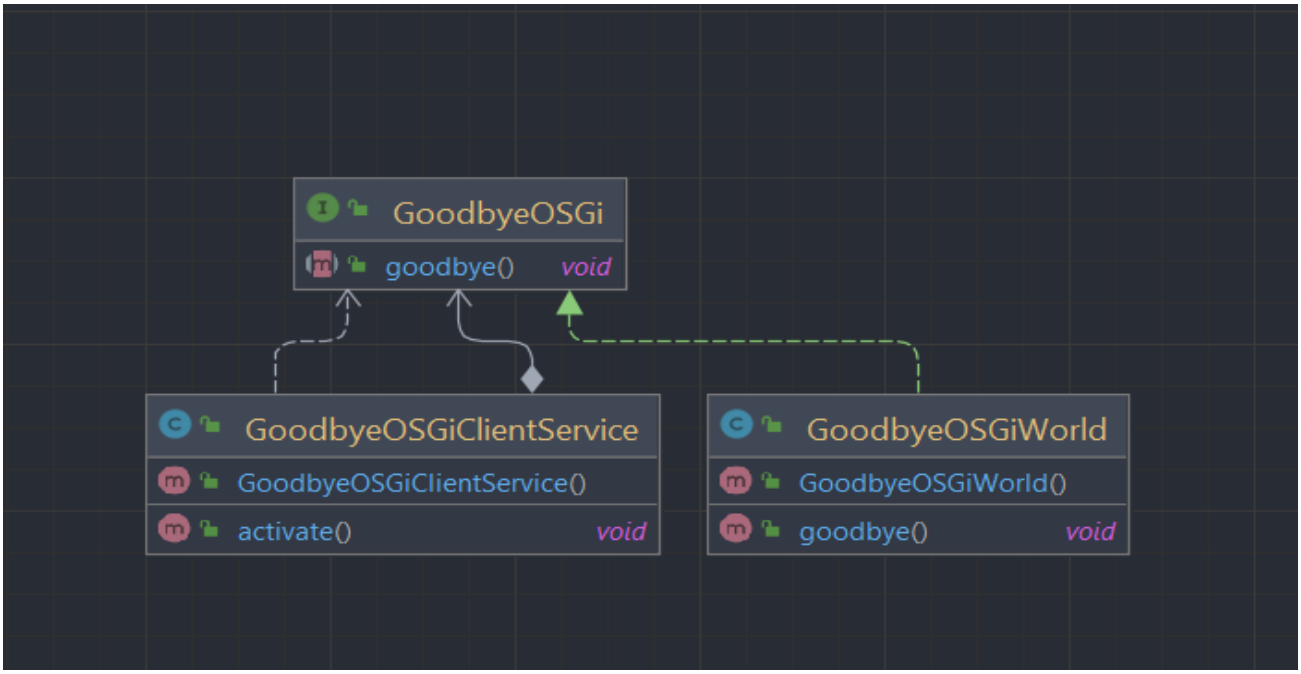
Схема модуля клиента

В классе клиента мы с помощью простой аннотации говорим, что этот класс является сервисом и указываем какой метод необходимо вызвать при активации этого сервиса, а также указываем ссылку на сервис предоставляющий интерфейс и реализацию.

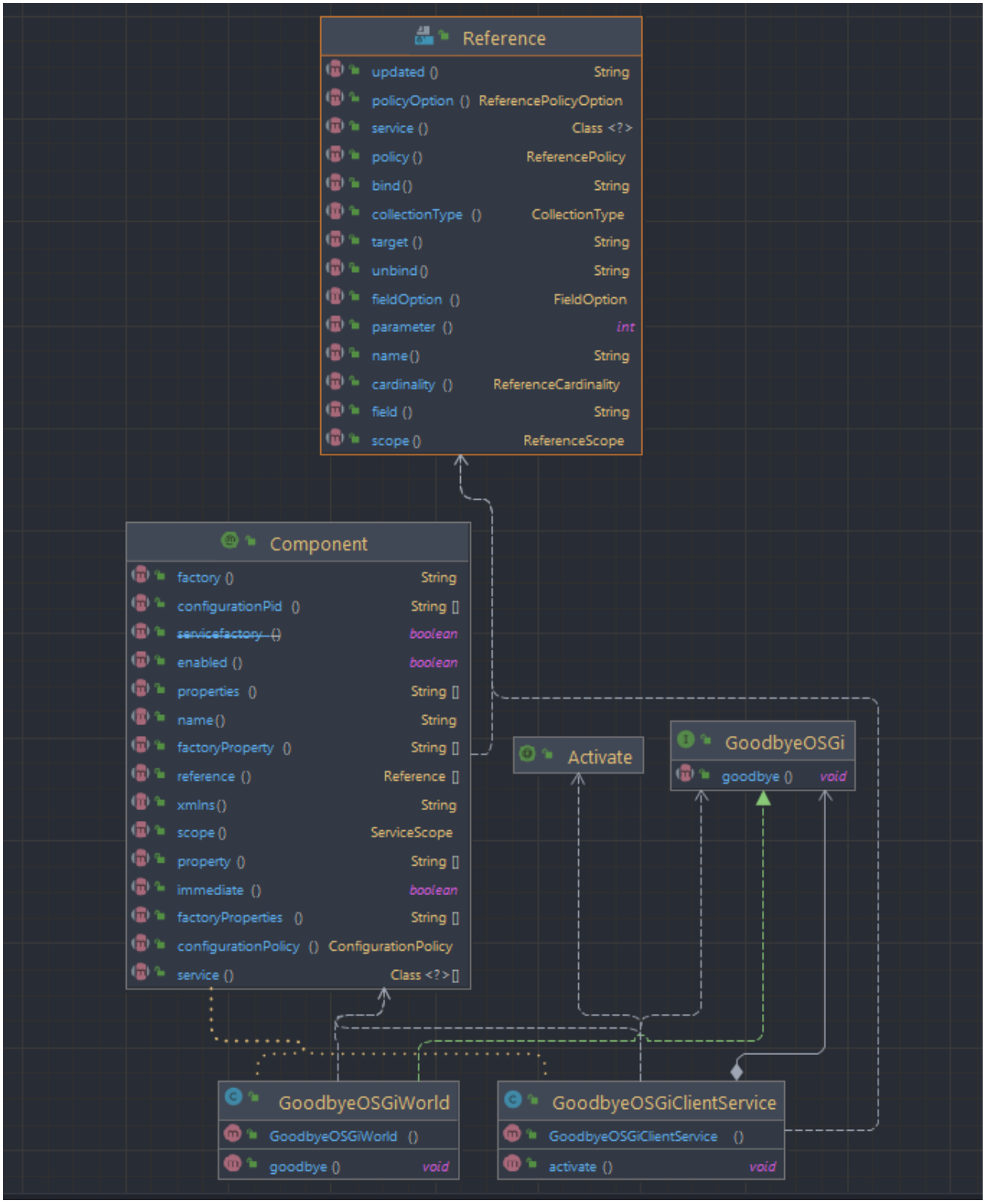
```java
@Component
@ServiceVendor("Pushkarev Gleb")
@ServiceDescription("Provides a friendly farewell on activation.")
public class GoodbyeOSGiClientService {

    @Reference(service = GoodbyeOSGi.class)
    private GoodbyeOSGi goodbyeOSGi;

    @Activate
    public void activate() { goodbyeOSGi.goodbye(); }
}
```

Зависимости устроены так же, как и во 2 этапе, но к ним добавляются:

1. *org.osgi.service.component.annotations* – предоставляет основные аннотации
2. *org.osgi.service.component* – предоставляет вспомогательны аннотации (vendor и description)
3. *maven-scr-plugin* – генерирует xml файлы с необходимой информацией для фреймворка
4. *org.apache.felix.scr.annotations* – необходим для *maven-scr-plugin*



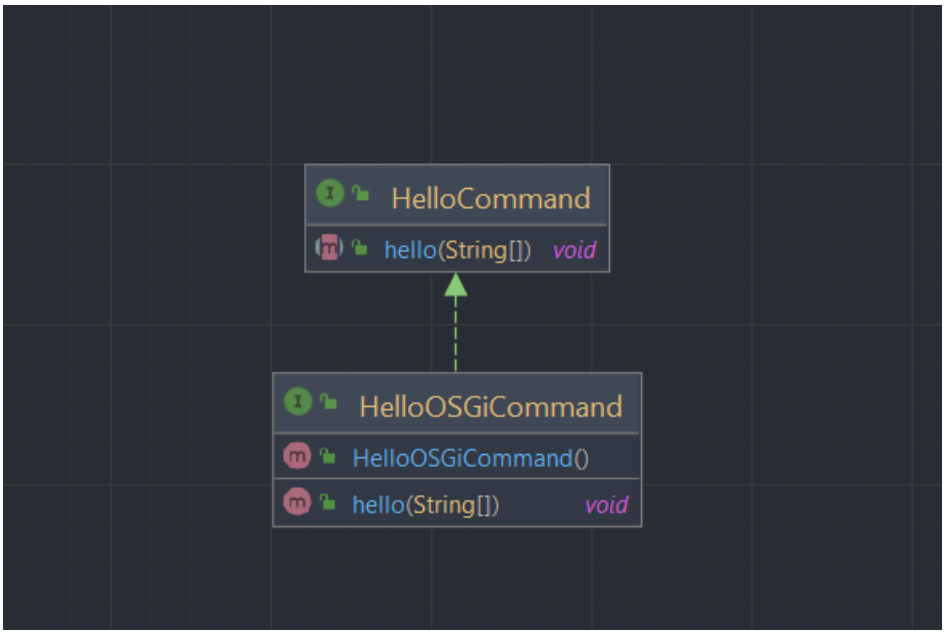
Общая схема Этапа 2



Общая схема с аннотациями

- **Этап 4. Создание собственной команды для Apache Felix Gogo**

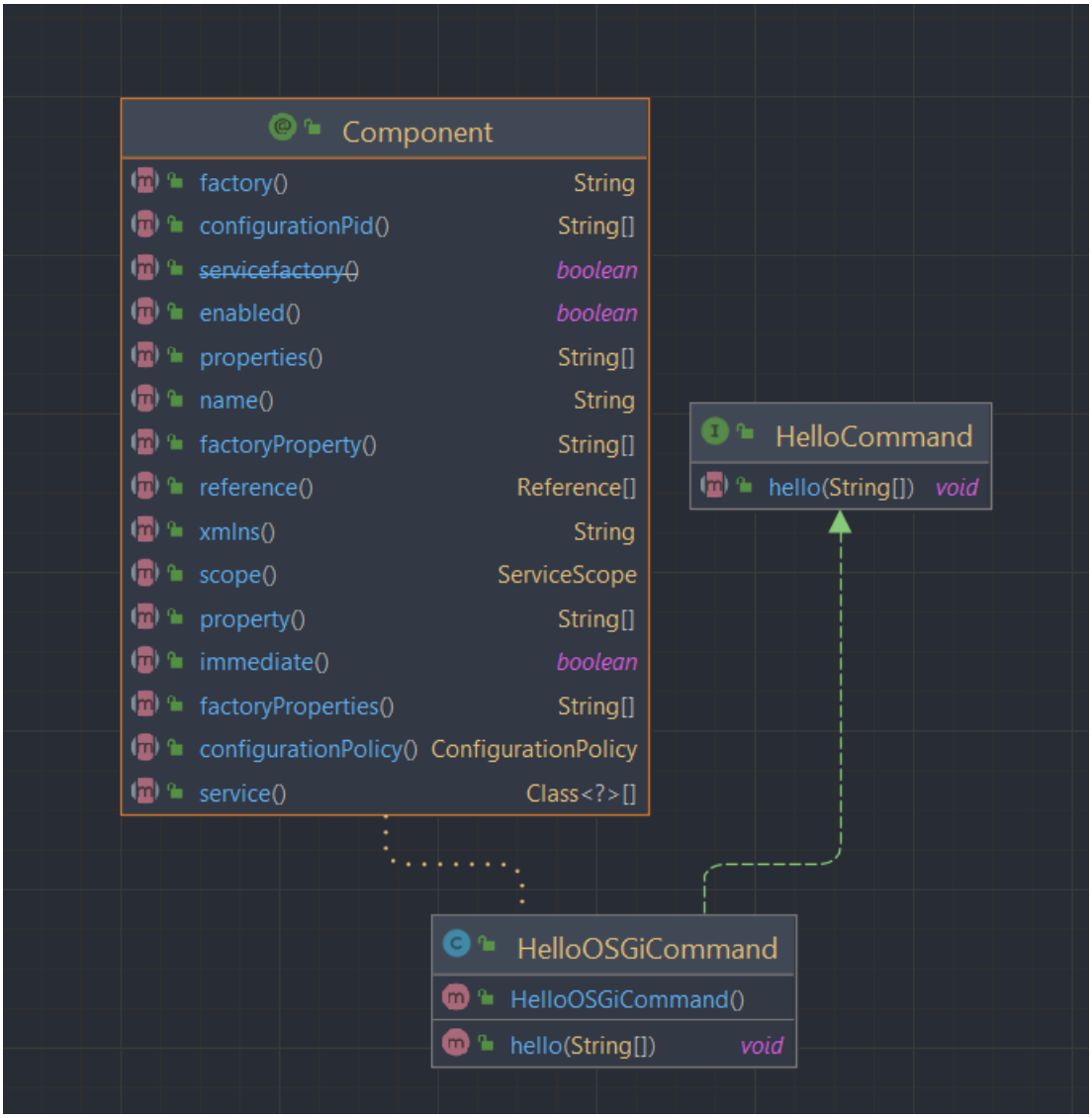
  Была создана собственную команда «*practice:hello*» с одним параметром, которая при вызовепечатает на консоль «Hello, <param>», где <param> - введенный пользователем параметр.
  Был создан один бандл, включающий в себя один интерфейс (API команды) и реализация команды с использованием аннотаций.



модуль ru.ifmo.pga.hello.command
бандл pga_hello_osgi_command

Модуль *ru.ifmo.pga.hello.command* содержит реализацию команды. Здесь так же использованы аннотации.

```
@Component(
        service = HelloCommand.class,
        property = {
                "osgi.command.scope=practice",
                "osgi.command.function=hello"
        }
)
@ServiceVendor("Pushkarev Gleb")
@ServiceDescription("Provides a friendly greeting.")
public class HelloOSGiCommand implements HelloCommand {

    @Override
    public void hello(String... nameOfUsers) {
        if (nameOfUsers.length != 1) {
            System.out.println("Please specify one username.");
            return;
        }
        System.out.println("Hello, " + nameOfUsers[0]);
    }
}
```



Общая схема с аннотациями

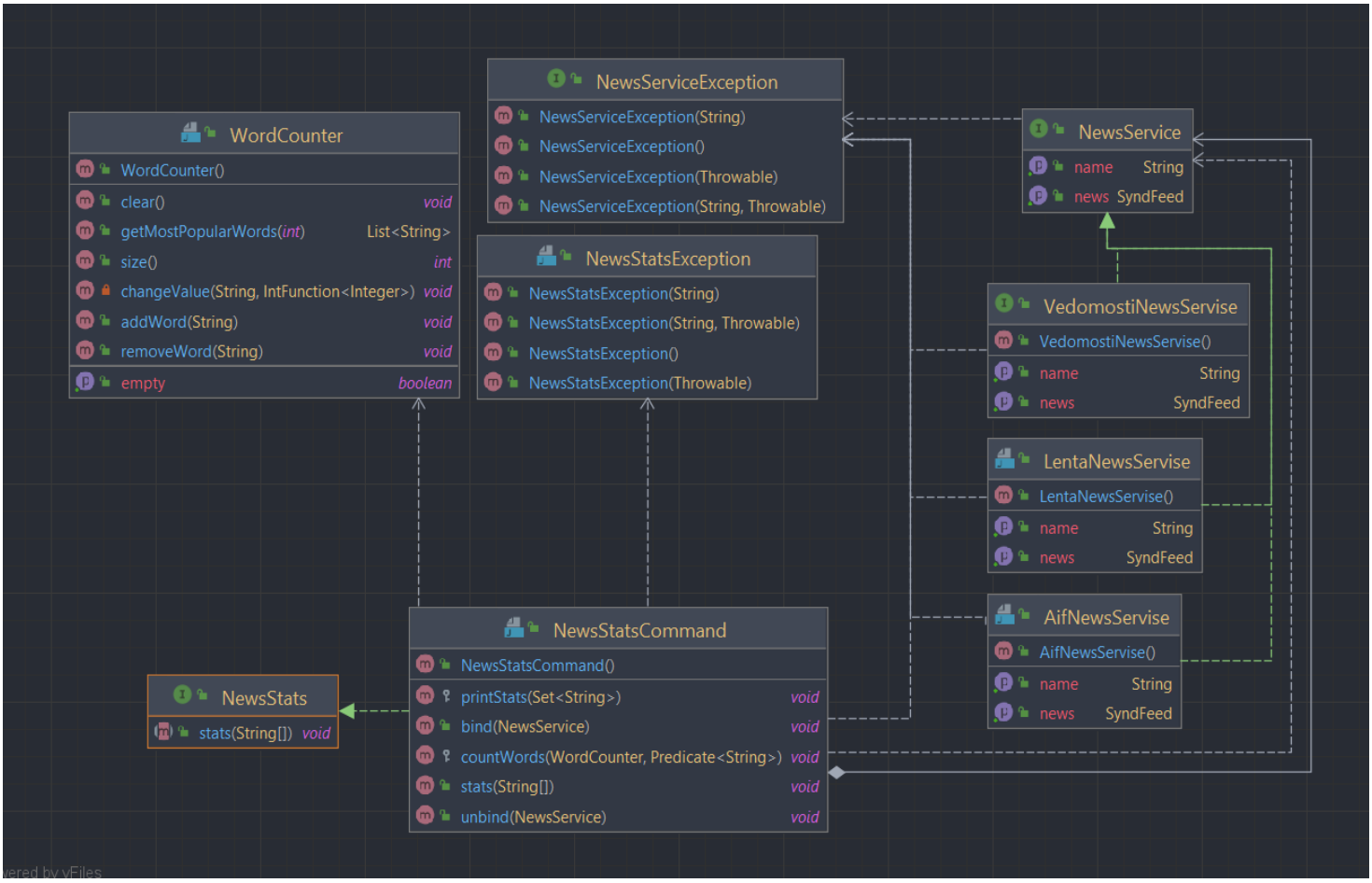
Зависимости такие же, как и в этапе 3.

- **Этап 5. Создание приложения**

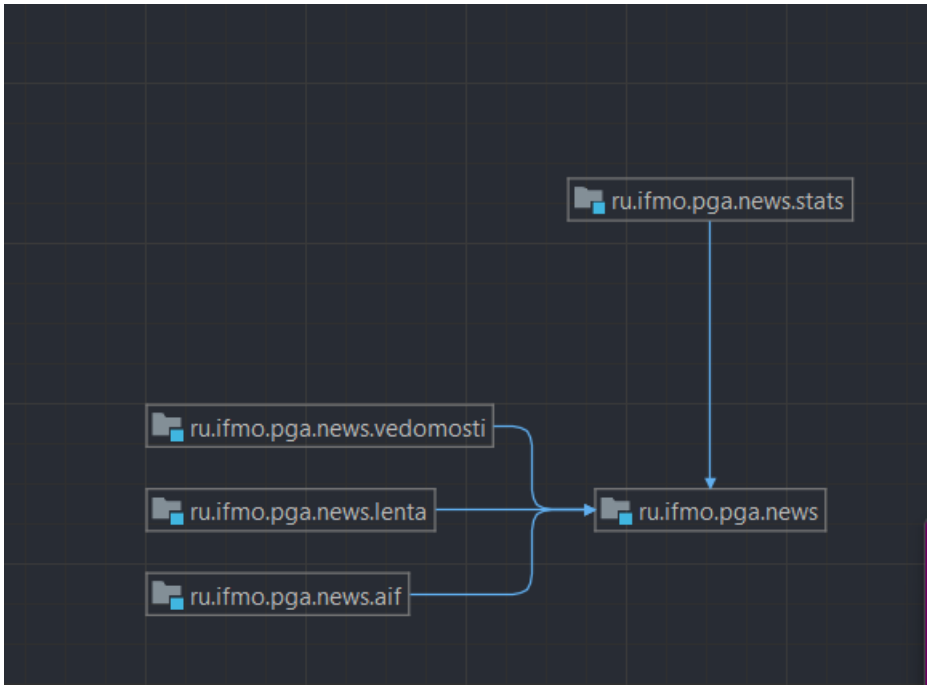
  Было создано приложение, которое получает через API новостных порталов список актуальных новостей и выводит на консоль 10 самых часто встречающихся слов из заголовков новостей.

  Процесс подсчета инициируется пользователем приложения с помощью консольной команды «*news:stats*». Пользователь может передать источник в качестве параметра. Если команда вводится без параметров, пользователю предлагается выбрать источник данных (один из доступных в системе, или все сразу). Список источников выводится на консоль.

  Если пользователь выбирает незарегистрированный (недоступный) источник данных, в системе не зарегистрировано ни одного источника или сеть недоступна, пользователю выводится соответствующее сообщение.



*модули ru.ifmo.pga.news.**



Зависимости между модулями

Зависимости как в 3 и 4 этапах, а также:

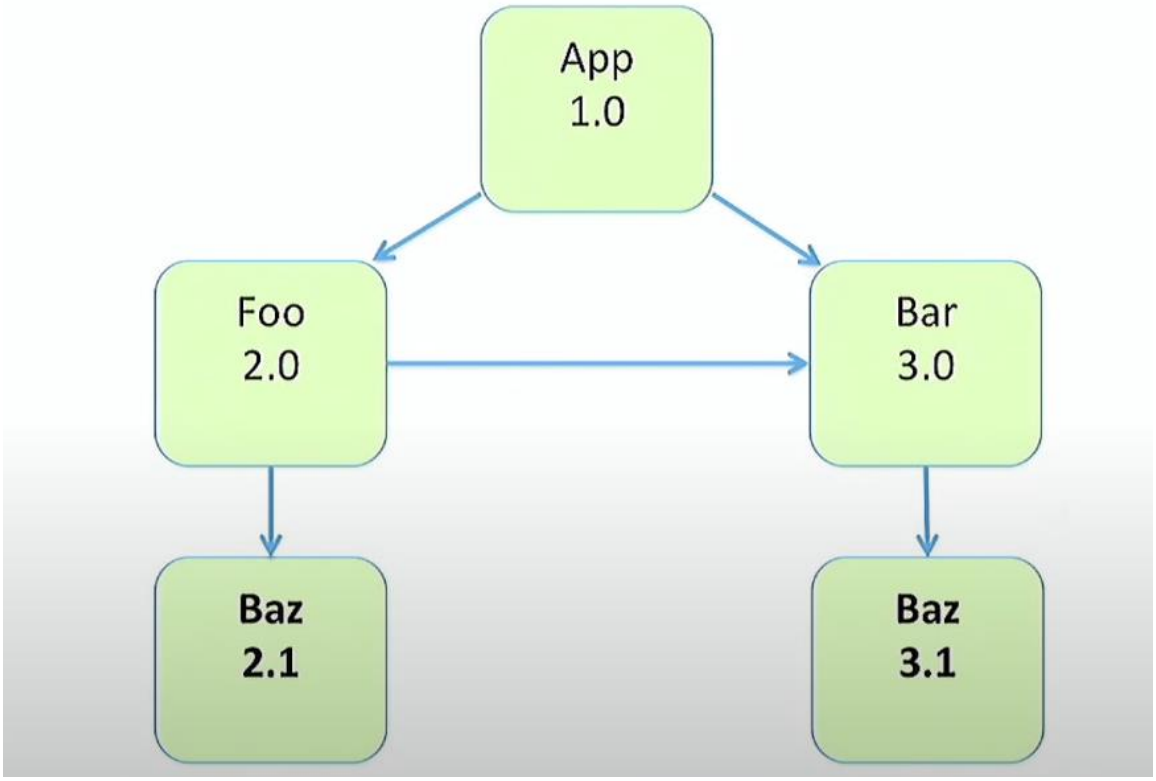
1. *rome* - парсит RSS новости
2. *slf4j-simple* - необходима для *rome*
3. *jaxen* - необходима для *rome*

Выводы

- **Преимущества**
  1. Возможность на ходу обновлять/перезагружать бандлы, но только в некоторых случаях, так как при перезагрузке бандла, перезагрузятся и все бандлы, имеющие на него зависимость, что может все сломать. Например:
     Бандл содержащий *SCR* корректно запускается, но не работает если он был запущен позже бандла содержащего *LogService*,
     Тогда при обновлении бандла на который у *SCR* есть зависимость, он так же перезагрузиться и не сможет запуститься, потому что *LogService* уже запущен и не перезапускался, фреймворк не может корректно разбирать такие случаи.
     Так же циклические зависимости все ломают (о них дальше).

  2. Возможность скрыть реализацию.
     На самом деле не работает ведь из контекста возвращается конкретный класс – класс реализации, а reflection никто не отменял.

  3. Возможность загружать несколько библиотек разных версий в одно приложение:
     Но так делать все равно нельзя, так что такой себе плюс.
     Тут проблему **Jar Hell** никто не решил.
     Вот пример, когда все сломается.



- **Недостатки**
    1. Пример с обновлением бандла, в любом случае не стоит ничего обновлять на продакшене, а следовательно, вначале поднимать полностью идентичный контейнер OSGi, для тестирования обновления, что может быть проблематично.
       Нормально работает только с “плагинами”, но их можно реализовать в разы проще.

    2. Сокрытие: Jigsaw намного лучше все это скрывает и никакой reflection не поможет.

    3. Возможность загружать несколько библиотек разных версий, уже указано почему.

    4. Разрешены циклические зависимости, что само по себе плохо, дак еще и придётся полностью выгружать приложение (либо все упадет).

    5. Все реализовано на куче загрузчиков классов, это, во-первых, не слишком быстро, а во-вторых, может произойти утечка памяти (*Classloader memory leak)*

    6. Еще немного про циклические зависимости.
       Порядок загрузки классов в JVM не определен, а порядок загрузки бандлов зависит от порядка загрузки классов JVM, если она будет энергично разрешать ссылки между классами OSGi упадет, а на нахождение подобного рода ошибки потребуется очень много времени и ресурсов

    7. Все выполняется в runtime, OSGi не интегрирован в Java, а следовательно проверки не могут проводиться в compile time, а это бы помогло избежать многих вышеизложенных проблем.
       Опять же Jigsaw позволяет все это делать в compile time.

    8. При использовании библиотек, в манифесте которых не прописаны конфигурация для использования в OSGi, придётся собственноручно переупаковывать их в бандлы, просто завернуть их внутрь своего бандла не выйдет.
       К тому же иногда такие библиотеки требуют compile time зависимости, которых может просто не быть в открытом доступе, в таком случае возможно не получиться использовать библиотеку.

    9. Jigsaw не имеет ни 1 из выше перечисленных проблем.

- **Какие системы целесообразно реализовывать на OSGi**
  Кажется, что никакие, кроме тех случаем, когда вам необходимо использовать модули в версии Java ниже 9.

  Кажется, что можно использовать как песочницу для тестирования архитектуры приложения, использующем архитектуру микрсервисов.

- **Выводы**
  Ни 1 из фич предложенных разработчиками OSGi нормально не работает, используйте Jigsaw.