Содержание

1 Введение

При разработке достаточно больших по объему кода сервисов на объектно-ориентированных языках программирования желательно придерживаться пяти основных принципов дизайна классов — SOLID:

1. Single responsibility principle — на класс должна быть возложена лишь одна ответственность
2. Open-closed principle — класс должен иметь возможность расширения функциональности без изменения имеющейся
3. Liskov substitution principle — наследник должен уметь заменять свой базовый класс без изменения поведения
4. Interface segregation principle — универсальные интерфейсы следует разделять на более специализированные
5. Dependency inversion principle — зависимости следует выстраивать между абстракциями и не следует между реализациями

Соблюдение последнего принципа подразумевает, что использование оператора создания объекта недопустимо в коде реализации модуля (если это не фабрика). Из этого следует, что все зависимости нужно принимать в конструктор. Получается, что в коде приложения должна быть некоторая точка старта, в которой создаются все используемые объекты. В достаточно больших проектах точка старта будет иметь внушительный объем, из-за чего она будет трудноподдреживаема.

Одним из решений данной проблемы является использование Dependency Injection (или Inversion of Control) контейнеров. Контейнер берет на себя ответственность за создание объектов и разрешение зависимостей. Существует достаточно много реализаций DI-контейнеров для языка Java. В параграфе 1 рассмотрены два из числа наиболее используемых контейнера, выделен их главный недостаток — большой объем конфигурационного кода:

контейнеры требуют от разработчика тем или иным способом перечислить все классы, экземпляры которых потребуется получать или создавать.

Была поставлена цель — реализовать DI-контейнер, который не потребует большого количества конфигурационного кода.

Цель достигнута: в параграфе 2 описан контейнер, отличительной особенностью которого является отсутствие необходимости перечислять все классы, используемые приложением при его конфигурировании. Имеется опыт внедрения контейнера в веб-сервис. Исходный код и собранная библиотека доступны со страницы проекта https://code.google.com/p/jrobo-container .

2 Анализ существующих DI-контейнеров

2.1 Применение принципа инверсии зависимостей без использования DI-контейнеров

Рассмотрим следующий пример. Есть сервис, поддерживающий несколько способов аутентификации:

- аутентификация с использованием сторонних сервисов
- аутентификация с использованием собственного механизма

Аутентификация будет производиться через интерфейс IAuthenticator, который описан в листинге 1:

Листинг 1: IAuthenticator

```
public interface IAuthenticator
{
    String authenticate(String login, String password);
    String getName();
}
```

Метод authenticate принимает на вход логин и пароль пользователя и возвращает идентификатор сессии или null, если аутентификация не удалась, метод getName возвращает имя способа аутентификации, который реализован данным классом. Необходимо реализовать интерфейс, описанный в листинге 2:

Листинг 2: IAuthenticatorsProvider

```
public interface IAuthenticatorsProvider
{
    IAuthenticator getAuthenticator(String name);
}
```

Метод getAuthenticator должен принять на вход одну строку name и вернуть реализацию способа аутентификации с таким именем или null, если такого нет.

Рассмотрим реализацию данной задачи без использования DI контейнеров. В листинге 3 приведен код класса AuthenticatorsProvider, реализующе-

го интерфейс IAuthenticatorsProvider. Листинг 4 демонстрирует, как нужно создавать экземпляр этого класса без использования контейнера. Здесь LocalAuthenticator — класс, отвечающий за аутентификацию с использованием собственного механизма, а GmailAuthenticator и FacebookAuthenticator — классы, отвечающие за аутентификацию с помощью сторонних сервисов.

Листинг 3: Реализация интерфейса IAuthenticatorsProvider

```
public class AuthenticatorsProvider implements IAuthenticatorsProvider
{
    IAuthenticator[] authenticators;

    public AuthenticatorsProvider(IAuthenticator[] authenticators)
    {
        this.authenticators = authenticators;
    }

    public IAuthenticator getAuthenticator(String authenticatorName)
    {
        for(IAuthenticator authenticator : authenticators)
            if(authenticator.getName().equals(authenticatorName))
                return authenticator;
        return null;
    }
}
```

Листинг 4: Место получения (в данном случае создания) экземпляра класса AuthenticatorsProvider

```
IAuthenticatorsProvider provider = new AuthenticatorsProvider(new IAuthenticator[]{new
    LocalAuthenticator(), new FacebookAuthenticator(), new GmailAuthenticator()});
```

В листинге 4 показан кусок достаточно рутинного кода. В действительности, когда классов на порядок больше, количество рутины соответственно возрастет. Dependency Injection контейнеры берут на себя ответственность за создание объектов, тем самым избавляя разработчика от ручного труда.

2.2 Spring IoC container

Spring Framework — один из самых широко распространенных фреймворков среди Java-разработчиков. Его можно рассматривать как коллекцию меньших фреймворков, каждый из которых может работать по отдельности. Одна из компонент Spring — это IoC контейнер.

2.2.1 Способы конфигурирования

Spring IoC Container — контейнер, конфигурирование которого производится в основном с использованием XML-файлов. Создать контейнер можно следующим образом:

Листинг 5: Создание контейнера

```
ApplicationContext applicationContext = new ClassPathXmlApplicationContext(new String[]{"
    beans.xml"});
```

Здесь «beans.xml» — XML-файл, в котором описана конфигурация контейнера.

В файле с конфигурацией необходимо описать все объекты, которые нужно будет впоследствии получать из контейнера. Для описания объекта создается XML-элемент <bean>, в атрибутах которого надо указать id — идентификатор объекта и class — полное имя класса, экземпляром которого этот объект будет являться. Внутри элемента можно указать, какие значения в какие поля подставить при построении данного объекта. Например, если есть класс Sample с конструктором, принимающим на вход два аргумента String и int, и из контейнера хочется получить экземпляр этого класса с подстановкой в качестве аргумента строки «Just sample» и числа 146, нужно создать следующий элемент:

Листинг 6: Пример конфигурации объекта

```
<bean id="SampleClassImpl" class="Sample">
        <constructor-arg type="int" value="146" />
        <constructor-arg type="java.lang.String" value="Just␣sample" />
</bean>
```

В элементе <constructor-arg> необходимо указать, в какой именно аргумент будет производиться подстановка и значение, которое нужно подставить. Аргумент можно указать с помощью следующих атрибутов:

- type – тип аргумента конструктора
- index – номер аргумента конструктора
- name – имя аргумента конструктора

Подставляемое значение указывается одним из двух способов:

- value – значение, которое хотим передать в качестве аргумента конструктору
- ref – идентификатор объекта, который хотим подставить в качестве аргумента (объект с этим идентификатором также должен быть описан в конфигурационном файле)

Если в файле с конфигурацией есть описание лишь одного объекта, имеющего заданный тип, и нужно подставить этот объект в некоторое поле в другом объекте, то можно воспользоваться аннотацией @Autowired. Если поле помечено этой аннотацией, то контейнер сам найдет объект, который нужно подставить. Также этой аннотацией можно пометить конструктор, тогда в качестве аргументов в конструктор будут передаваться те элементы, которые найдет контейнер.

Если поле помечено @Autowired, но при этом есть несколько описаний объектов одного и того же типа — будет вызвано исключение. Но если при этом @Autowired помечен массив объектов, то контейнер подставит в это поле все объекты, которые указаны в конфигурации и подходят под данный тип. Для того, чтобы возспользоваться этой аннотацией, необходимо в XML-файле добавить тэг <context:annotation-config />

Основные моменты реализации описанного в начале примера выглядит следующим образом:

Листинг 7: Файл конфигурации (без стандартных заголовков)

```
<context:annotation-config/>
<bean id="gmail" class="GmailAuthenticator"/>
<bean id="facebook" class="FacebookAuthenticator"/>
<bean id="local" class="LocalAuthenticator"/>
<bean id="authenticatorsProvider" class="AuthenticatorsProvider"/>
```

Листинг 8: Конструктор класса AuthenticatorsProvider

```
@Autowired
public AuthenticatorsProvider(IAuthenticator[] authenticators)
{
        this.authenticators = authenticators;
}
```

Листинг 9: Место получения экземпляра

```
IAuthenticatorsProvider provider = (IAuthenticatorsProvider)applicationContext.getBean("
    authenticatorsProvider");
```

2.2.2 Плюсы и минусы

Основной плюс данного контейнера — это его гибкость. Богатый набор способов конфигурации объектов позволяет собрать практически любой экземпляр, который может понадобиться. Однако, есть ряд недостатков.

Метод getBean у ApplicationContext всегда возвращает Object, то есть контейнер не является «type safe», и каждый раз, когда мы делаем запрос на получение объекта, нам нужно явно приводить полученный объект к нужному типу.

Если появится новый механизм аутентификации — придется править конфигурационный файл: добавлять новый элемент конфигурации.

Как видно, почти вся информация о конфигурации должна содержаться в конфигурационном XML-файле (как минимум, там нужно описать все объекты, которые могут понадобиться). Понятно, что его размер прямо пропорционален размеру проекта, в который происходит внедрение контейнера, соответственно «с нуля» внедрить его в огромный проект достаточно трудно.

2.3 Google Guice

2.3.1 Способы конфигурирования

Google Guice — контейнер, конфигурирование которого производится с помощью классов-конфигураторов и аннотаций. Создается контейнер с помощью следующего кода:

Листинг 10: Создание контейнера

```
Injector injector = Guice.createInjector(new Module());
```

Класс Module должен наследоваться от класса AbstractModule, который требует реализации одного метода — configure. В этом методе и будет

описана конфигурация. В этом месте нужно явно прописать, какой класс будет реализовывать конкретную абстракцию, или какой конкретно объект вернуть при запросе, например:

Листинг 11: Примеры конфигурирования в наследнике AbstractModule

```
@Override
protected void configure()
{
        bind(IAuthenticatorsProvider.class).to(AuthenticatorsProvider.class);
        bind(IGmailAuthenticator.class).toInstance(new GmailAuthenticator());
}
```

Для инжектирования зависимости используется аннотация @Inject.

В отличие от Spring IoC Container, Google Guice не позволяет столь же просто сделать инъекцию в конструктор с аргументом типа массив, но позволяет сделать это с аргументом типа Set. Для этого используется класс Multibinder.

Реализация нашего примера с использованием данного контейнера имеет следующие ключевые моменты:

Листинг 12: Конфигурирование

```
import com.google.inject.AbstractModule;
import com.google.inject.multibindings.Multibinder;

public class Module extends AbstractModule
{
    @Override
    protected void configure()
    {
        bind(IAuthenticatorsProvider.class).to(AuthenticatorsProvider.class);
        Multibinder<IAuthenticator> multibinder = Multibinder.newSetBinder(binder(),
            IAuthenticator.class);
        multibinder.addBinding().to(FacebookAuthenticator.class);
        multibinder.addBinding().to(GmailAuthenticator.class);
        multibinder.addBinding().to(LocalAuthenticator.class);
    }
}
```

Листинг 13: Конструктор класса AuthenticatorsProvider

```
@Inject
public AuthenticatorsProvider(Set<IAuthenticator> authenticators)
{
    this.authenticators = authenticators.toArray(new IAuthenticator[authenticators.size()]);
}
```

Листинг 14: Место получения экземпляра

```
IAuthenticatorsProvider provider = injector.getInstance(IAuthenticatorsProvider.class);
```

2.3.2 Плюсы и минусы

Google Guice, в отличие от Spring IoC Container, сохраняет тип, соответственно не придется каждый раз явно использовать приведение типа при запросе getInstance к контейнеру.

Однако проблема с необходимостью описания всех классов в конфигурации никуда не делась, просто вся рутина переместилась из XML-файла в класс-конфигуратор. Более того, при внедрении контейнера придется прописывать @Inject во всех местах, где потребуется вмешательство контейнера, то есть придется вносить достаточно существенные изменения в изначальный код.

3 Описание реализованного контейнера

Опишем реализованный контейнер в общих чертах.

В момент создания контейнер прочитает скомпилированные классы из указанных папок и построит на этих классах граф реализации; эти процессы описаны в параграфе 4.

Возможности точечной конфигурации описаны в параграфе 5.

При запросах на получение или создание вызываются методы, отвечающие за конструирование объекта. Алгоритмы конструирования описаны в параграфе 7.

Приложение, написанное на Java, обычно состоит из множества модулей, каждый из которых должен быть загружен определенным загрузчиком классов, более подробно этот вопрос освещен в параграфе 6.

Дополнительные возможности контейнера — протоколирование процесса построения объекта и автоматическая генерация классов-фабрик — описаны в параграфах 8 и 9.

3.1 Инициализация контейнера

Инициализация контейнера заключается в просмотре некоторого множества байткодов классов и составлении графа наследования между этими классами.

3.1.1 Просмотр байткодов

Для начала требуется определиться с тем, экземпляры каких именно классов будем получать из контейнера и где лежат соответствующие им байткоды. Иногда будет достаточно просто просмотреть содержимое папок, перечисленных в свойстве «java.class.path», но бывают случаи, когда этого недостаточно. Например, если речь идет о веб-сервисе, то скорее всего это свойство просто не определено, и в таких случаях очень часто невозможно понять по переменным среды откуда берутся классы, из-за чего придется

вручную сообщить контейнеру о всех папках, в которых следует искать требуемые файлы. Также следует указать контейнеру, какие jar-файлы следует обрабатывать, а какие нет. Всю эту информацию контейнер может принять в конструктор в виде реализации интерфейса IClassPathScannerConfiguration. Он имеет два метода:

- getClassPaths — метод должен возвращать список всех папок, в которых следует искать байткоды
- acceptsJar — метод принимает в качестве единственного аргумента имя jar-файла, и должен вернуть true, если требуется просмотреть его содержимое и найти там байткоды, или false в противном случае

Если создавать контейнер с использованием пустого конструктора, то будет использоваться реализация интерфейса, которая возвращает значение свойства java.class.path в качестве списка папок для просмотра, и не просматривает никакие jar-файлы.

Просмотр папок производится с помощью поиска в глубину по дереву директорий. При анализе очередного элемента папки возможны 4 варианта типа элемента и соответствующих действий:

- папка — рекурсивно запустить поиск в глубину из этой папки
- class-файл — прочитать байткод из файла, преобразовать его в объект, содержащий информацию о внутреннем устройстве класса, и сохранить его в память
- jar-файл — с помощью метода acceptsJar принять решение, читать этот файл или нет, и в случае положительного решения запустить поиск байткодов по элементам jar-файла
- любой другой файл — ничего не делать

Для преобразования байткода используется библиотека BCEL (Byte Code Engineering Library) [5], которая позволяет прочитать байткод из файла и

преобразовать его в объект типа JavaClass, из которого в свою очередь достаточно удобно брать информацию об устройстве класса.

3.1.2 Построение графа реализации

Теперь стоит задача — построить структуру данных, которая позволит эффективно узнавать список всех реализаций конкретной абстракции. Здесь под абстракцией понимается интерфейс, абстрактный класс или неабстрактный класс, под классом же понимается исключительно неабстрактный класс.

При сохранении в память строится ориентированный граф прямой реализации между абстракциями по следующему правилу: из абстракции A в абстракцию B идет ребро тогда и только тогда, когда абстракция A является либо непосредственным предком абстракции B, либо интерфейсом, который абстракция B непосредственно реализует.

После того, как все папки просмотрены, все классы перечислены и на них построен граф прямой реализации, строится расширенный граф реализации, с помощью которого впоследствии контейнер будет принимать решение о том, экземпляр какого именно класса следует вернуть в качестве результата запроса. Его можно определить следующим образом: из абстракции A в абстракцию B идет ребро тогда и только тогда, когда B является неабстрактным классом, и при этом существует некоторый путь из абстракции A в абстракцию B в графе прямой реализации. Расширенный граф реализаций строится для того, чтобы можно было достаточно быстро получать список реализаций абстракции. Это делается с использованием одного запроса get к встроенному в язык Java HashMap'у, который в среднем работает за O(1).

Покажем, как должны выглядеть вышеописанные графы на примере, описанном в главе 2:

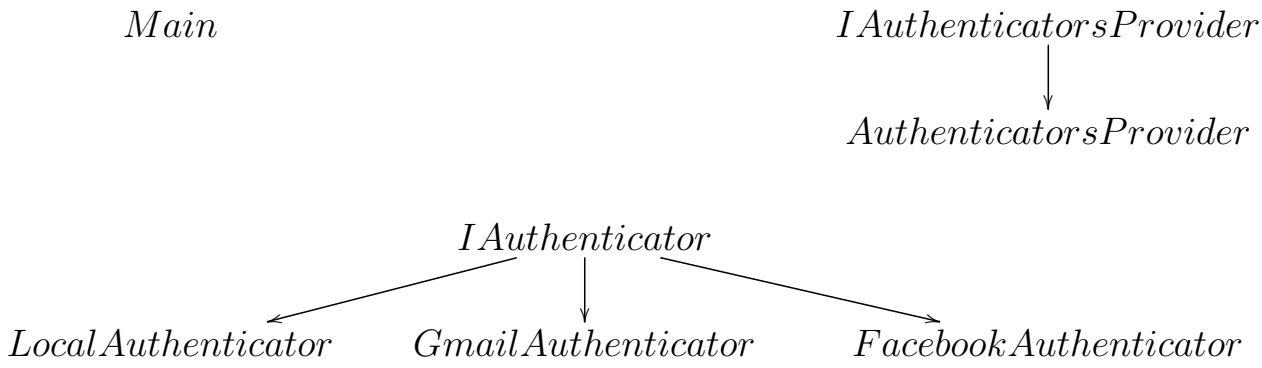

Рисунок 1. Граф прямой реализации

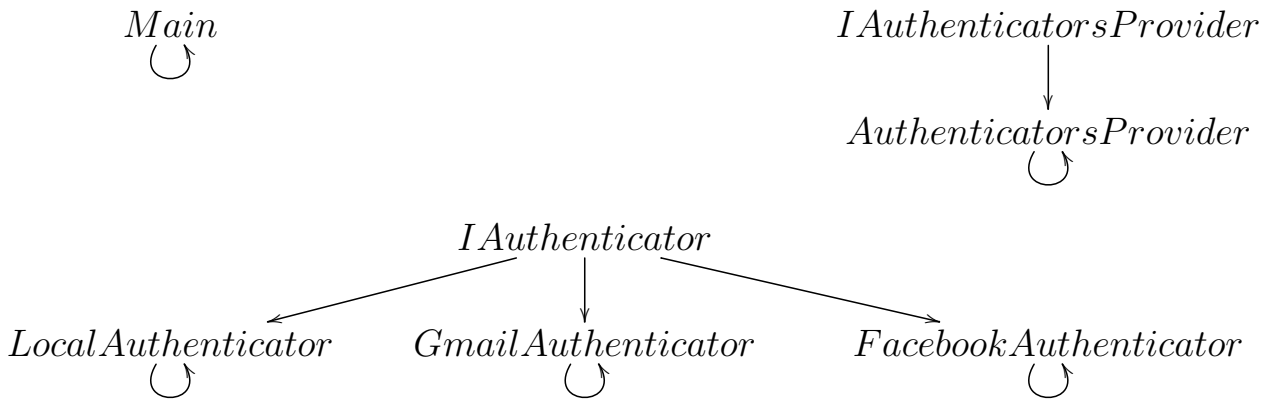

Рисунок 2. Расширенный граф реализации

3.2 Возможности конфигурации контейнера

3.2.1 Внутреннее устройство конфигурирования

Для достижения гибкости контейнера необходимо тщательно продумать внутренний механизм конфигурирования и его реализацию. При этом механизм должен быть достаточно гибким: он не должен быть жестко завязан

на имеющихся типах конфигурирования, наоборот, он должен быть открытым для новых. Отсюда появилась идея интерфейса IConfiguration. Каждая абстракция, которая рассматривается контейнером, имеет свою конфигурацию — информацию о том, как должен собираться соответствующий объект, если контейнеру требуется его создать. Вся эта информация содержится в классе, реализующем интерфейс IConfiguration. Он имеет три метода, таких же, как и методы-запросы у самого контейнера: get, create и getAll. В данный момент метод getAll ведет себя одинаково у всех реализаций: он получит все реализации данной абстракции из расширенного графа наследования, сделает для каждой из реализаций запрос на получение экземпляров и вернет все эти объекты в одном массиве в качестве результата. Существенные отличия будут в алгоритмах работы методов get и create. Манипулирование объектами-конфигурациями производится с помощью интерфейса IConfigurationsManager. Запрос к контейнеру делегируется соответствующему объекту IConfiguration, который обеспечивает поведение, предписанное пользователем.

Для описания работы реализаций этого интерфейса нам понадобятся несколько определений.

Разрешение абстракции в класс — это выбор единственного ребенка данной абстракции в расширенном графе наследования. Если он не единственный — считается, что разрешение невозможно и вызывается исключение.

Конструирование объекта (экземпляра) — это создание экземпляра определенного класса. Более подробно данный процесс будет рассмотрен в следующей части.

По умолчанию в качестве конфигурации у каждой абстракции выставлена AutoConfiguration.

Ответ на запрос create при данной конфигурации будет получен с помощью следующего алгоритма:

1. Разрешение абстракции в класс

2. Сборка экземпляра полученного класса, который и будет возвращен в

качестве результата

Результат запроса get при конфигурации по умолчанию будет получен с помощью следующего алгоритма:

1. Если запрос на получение экземпляра данной абстракции уже был когда-либо выполнен, то вернуть в качестве результата ответ на тот запрос
2. Разрешение абстракции в класс
3. Если абстракция была разрешена сама в себя (это значит, что абстракция сама является классом), то произойдет сборка экземпляра и он вернется в качестве результата
4. В противном случае будет сделан перевызов метода get у конфигурации, соответствующей классу, в который была разрешена эта абстракция

3.2.2 Привязка конкретной реализации к абстракции

Если абстракция имеет множество реализаций, то без дополнительной конфигурации контейнер не сможет принять решение, экземпляр какого класса возвращать.

Решение 3.1. *Расширить интерфейс контейнера, добавив в него метод bindImplementation, позволяющий привязывать к конкретной абстракции конкретную реализацию, экземпляр которой требуется впоследствии получать из контейнера.*

Метод имеет два аргумента. Вызов метода bindImplementation с аргументами abstraction и implementation привяжет к абстракции abstracion ее реализацию implementation, подменив текущую конфигурацию этой абстракции на конфигурацию с привязанной реализацией. Результаты запросов get и create при данной конфигурации будут получены с помощью делегирования этих запросов конфигурации, привязанной к указанной реализации.

3.2.3 Привязка конкретного экземпляра к абстракции

Бывают ситуации, в которых требуется получать из контейнера в качестве ответа на запрос на получение один и тот же объект, созданный вне контейнера.

Решение 3.2. *Расширить интерфейс контейнера, добавив в него метод bindInstance, позволяющий привязывать к конкретной абстракции конкретный экземпляр одного из ее наследников.*

Метод имеет два аргумента, первый имеет тип Class, второй — Object. Вызов bindInstance(abstraction, instance) привяжет к абстракции abstraction объект instance, подменив текущую конфигурацию этой абстракции на конфигурацию с привязанным экземпляром. При этой конфигурации запрос на создание будет обрабатываться тем же образом, что и при конфигурации по умолчанию, а при запросе на получение будет возвращаться всегда объект instance.

3.3 Загрузка классов

В, Java перед тем как использовать класс, необходимо загрузить его в JVM. В этой части будет рассмотрен вопрос получения класса по его байткоду и его загрузки.

Байткод — последовательность байт, получаемая в результате компилирования исходного кода класса.

Задача 3.1. *Реализовать механизм, позволяющий прочитать файл с байткодом и получить объект типа Class, чтобы потом с помощью этого объекта создавать экземпляры соответствующего класса.*

Поверхностное исследование дает следующий результат: в языке Java существует механизм загрузки классов, реализованный как абстрактный класс ClassLoader, наследники которого могут загружать классы. Среди методов данного класса есть protected-метод defineClass, который позволяет получить объект типа Class по массиву байтов, а также метод loadClass,

который загружает класс по его имени. Используя эти знания, можно сформулировать следующее

Решение 3.3. *Реализовать собственного наследника класса ClassLoader, в котором можно получить Class по байткоду и совершить его загрузку с помощью системного загрузчика (системный загрузчик классов может быть получен с помощью вызова ClassLoader.getSystemClassLoader()).*

Это решение действительно работает в некоторых случаях. Например, модульное тестирование кода не выявило никаких проблем. Некорректность данного подхода выявилась при попытке внедрить контейнер в веб-сервис: были обнаружены классы, которые требовали загрузку несистемым загрузчиком классов.

Более глубокое исследование механизма загрузки классов показало, что загрузка классов производится далеко не всегда с помощью системного загрузчика классов. Например, если нам требуется загрузить класс с некоторого удаленного источника, потребуется реализация загрузчика, который будет делать запрос по сети на некий сторонний сервис, с которого в качестве результата на этот запрос вернется нужный объект типа Class. Такие загрузчики могут ипользоваться, в частности, в клиент-серверных архитектурах.

Получается, что нельзя загружать все классы системным загрузчиком, потому что среди них может оказаться класс, который не сможет быть им загружен. И возникает новая

Задача 3.2. *Научиться определять, каким именно загрузчиком должен быть загружен данный класс.*

В язык Java встроен достаточно удобный механизм для профилирования и прочих смежных задач — Instrumentation. Данный интерфейс помимо прочих имеет метод getAllLoadedClasses, возвращающий список всех классов, которые подгружены JVM на момент вызова метода. У объекта типа Class есть метод getClassLoader, который вернет загрузчика данного класса. Получаем

Решение 3.4. *В момент инициализации контейнера получить список всех загруженных классов, после чего получить список всех ClassLoader'ов, использовавшихся для загрузки, и после этого, при необходимости загрузить класс, сначала проверить, не загружен ли он, и если он не загружен, то попытаться произвести загрузку одним из уже известных загрузчиков.*

Однако и это решение нам не подходит: если экземпляр контейнера является статическим полем и инициализируется во время загрузки класса, то возможна следующая ситуация: класс, в котором лежит контейнер, загружается не последним, метод getAllLoadedClasses вернет только уже загруженные классы, среди которых может не оказаться ни одного класса, который должен быть загружен некоторым ClassLoader'ом, и мы о нем просто ничего не узнаем.

Данные выводы подводят к следующему утверждению: *в момент инициализации контейнера невозможно узнать полный список всех загрузчиков классов, которые будут использованы во время работы приложения.* Отсюда вытекает

Задача 3.3. *Реализовать механизм, позволяющий сообщать контейнеру о загрузчиках конкретных классов.*

Понятно, что конфигурирование загрузчиков для каждого класса породит огромное количество кода, что противоречит нашему стремлению минимизировать код конфигурирования контейнера. Однако можно воспользоваться следующим соображением: *большинство классов загружаются тем же загрузчиком, что и интерфейс, который он реализует.* Мы всегда можем узнать загрузчик любого аргумента конструктора любого класса и загрузчик аргумента запроса на получение объекта к контейнеру с помощью метода getClassLoader(). Таким образом, получаем

Решение 3.5. *Расширить интерфейс контейнера, добавив в него метод, позволяющий привязывать к конкретному классу конкретный загрузчик, которым его необходимо грузить.*

Это решение сохраняет минималистичность конфигурации при выполнении вышеуказанного соображения, так как в этом случае классов, для которых нужно явно указать загрузчика, будет достаточно мало.

3.4 Конструирование объекта

Конструирование объекта происходит в тот момент, когда уже определено, экземпляр какого конкретно класса должен быть создан. Помимо конкретного класса также известен список требуемых подстановок в конструктор в случае параметризованного запроса на создание. Список подстановок представляет из себя массив пар «тип-экземпляр». В конструировании можно выделить три основных этапа:

1. выбор подходящего конструктора
2. получение аргументов для выбранного конструктора
3. вызов конструктора для создания объекта

3.4.1 Выбор подходящего конструктора

Получить список всех конструкторов класса можно с помощью вызова метода getConstructors(). Подходящий конструктор будем искать среди всех следующим алгоритмом:

1. Если конструкторов вообще не найдено — будет вызвано исключение
2. Если конструктор найден и он ровно один, то он будет считаться подходящим
3. Будут найдены все конструкторы, помеченные аннтоацией @ContainerConstructor
4. Если этих конструкторов несколько — будет вызвано исключение

5. Если этот конструктор ровно один — он будет считаться подходящим. Во всех остальных случаях ни один конструктор не проаннотирован как @ContainerConsturctor

6. Если список требуемых подстановок не null и существует конструктор, подходящий под этот список, то он будет считаться подходящим

7. Если конструктор с непустым списком аргументов ровно один, то он будет считаться подходящим

8. Иначе будет вызвано исключение — невозможно однозначно определить, какой именно конструктор использовать для сборки объекта

Определять, подходит ли конструктор под список требуемых подстановок будем с помощью следующего алгоритма:

1. Помечаем все подстановки в списке неиспользованными

2. Для каждого аргумента конструктора проверяем, является ли его тип предком типа элемента на соответствующей позиции в списке подстановок

3. Если является, то помечаем подстановку как использованную

4. Иначе проверяем, сможем ли мы получить экземпляр данного типа из контейнера, если не сможем — конструктор не подходит

5. После всех этих действий для всех аргументов проверяем, что все подстановки использованы. Если все — конструктор подходит, иначе нет

3.4.2 Получение аргументов для выбранного конструктора

Конструктор выбран, теперь необходимо построить список объектов для передачи их в качестве аргументов в этот конструктор для последующего создания объекта. На этом этапе будет активно использоваться список требуемых подстановок, если он не пуст. Используется следующий алгоритм:

1. Помечаем все подстановки в списке неиспользованными
2. Просматриваем список типов аргументов выбранного конструктора
3. Если существует неиспользованная подстановка, тип которой является наследником типа текущего аргумента, то
   (a) Помечаем данную подстановку как использованную
   (b) Подставляем в качестве очередного аргумента объект из данной подстановки
4. В противном случае подставляем значение, которое вернет get-запрос по данному типу к контейнеру

3.4.3 Вызов конструктора для создания объекта

Мы определились с выбором конструктора и построили массив аргументов для передачи в этот конструктор. Осталось вызвать у выбранного конструктора метод newInstance(parameters), и мы получим необходимый объект.

3.5 Протоколирование в контейнере

В контейнер заложена возможность получения протокола (лога) сборки последнего объекта. Метод getLastLog вернет строковое представление лога в достаточно удобном формате: будут отмечены все этапы построения объекта, все вызовы методов получения или создания объектов, отмечены переиспользования уже собранных объектов, для удобства добавлена табуляция. Например, для нашего примера лог построения требуемого объекта будет выглядеть как в листинге 15:

Листинг 15: Пример лога получения экземпляра реализации интерфейса IAuthenticatorsProvider

```
Getting IAuthenticatorsProvider
  Creating IAuthenticatorsProvider
    Getting AuthenticatorsProvider
```

```
      Creating AuthenticatorsProvider
        Getting all IAuthenticator
          Getting FacebookAuthenticator
            Creating FacebookAuthenticator
            End creating FacebookAuthenticator
          End getting FacebookAuthenticator
          Getting GmailAuthenticator
            Creating GmailAuthenticator
            End creating GmailAuthenticator
          End getting GmailAuthenticator
          Getting LocalAuthenticator
            Creating LocalAuthenticator
            End creating LocalAuthenticator
          End getting LocalAuthenticator
        End getting all IAuthenticator
      End creating AuthenticatorsProvider
    End getting AuthenticatorsProvider
  End creating IAuthenticatorsProvider
End getting IAuthenticatorsProvider
```

3.6 Фабрики

Бывают ситуации, когда необходимо создать несколько объектов одного типа, которые отличаются лишь аргументами, которые следует передать в конструктор. С помощью контейнера это можно сделать вручную с помощью множества вызовов метода create с нужными подстановками. Например, если у нас есть класс Sample с единственным конструктором, принимающим на вход два аргумента типа int, то создать несколько экземпляров с разными аргументами можно следующим кодом:

Листинг 16: Пример создания множества экземпляров с разными параметрами

```
container.create(Sample.class, new AbstractionInstancePair(int.class, 0), new
    AbstractionInstancePair(int.class, 1));
container.create(Sample.class, new AbstractionInstancePair(int.class, 2), new
    AbstractionInstancePair(int.class, 3));
container.create(Sample.class, new AbstractionInstancePair(int.class, 4), new
    AbstractionInstancePair(int.class, 5));
```

Был реализован механизм, позволяющий избежать использования контейнера напрямую (то есть непосредственно вызова метода create). Суть его следующая: вместо того, чтобы создавать объекты прямо конструктором, переложить эту обязанность на контейнерные фабрики. Контейнерная фаб-

рика — интерфейс, который имеет методы create с нужными аргументами, возвращающие значения нужного типа, реализация которого генерируется самим контейнером. Например, для указанного выше примера должен быть создан следующий интерфейс:

Листинг 17: Пример фабрики

```
@ContainerFactory
public interface ISampleFactory
{
        Sample create(int a, int b);
}
```

Аннотация @ContainerFactory указывает контейнеру, что данная абстракция является контейнерной фабрикой, и необходимо сгенерировать ее реализацию. Например, для нашего примера будет сгенерирован следующий класс:

Листинг 18: Реализация интерфейса ISampleFactory, которая будет сгенерирована контейнером

```
public class ISampleFactoryGeneratedImplementation
{
        private IContainer container;

        public ISampleFactoryGeneratedImplementation(IContainer container)
        {
                this.container = container;
        }

        public Sample create(int a, int b)
        {
                return container.create(Sample.class, new AbstractionInstancePair[]{new
                    AbstractionInstancePair(int.class, a), new AbstractionInstancePair(int.
                    class, b)});
        }
}
```

И теперь код, приведенный в начале главы, преобразится в следующий:

Листинг 19: Код с использованием контейнерных фабрик

```
factory.create(0, 1);
factory.create(2, 3);
factory.create(4, 5);
```

Видно, что теперь создавать объекты стало гораздо удобнее, и для этого надо всего лишь создать интерфейс и в нужном месте принять этот интер-

фейс в конструктор, контейнер по аннотанции @ContainerFactory поймет, что этот интерфейс является фабрикой и сам сгенерирует его реализацию.

Более формально процесс генерации реализации можно описать следующим образом:

1. Создать поле, в котором будет лежать контейнер
2. Создать конструктор, принимающий в качестве единственного аргумента контейнер
3. Просмотреть все методы интерфейса, имеющие имя create, и создать реализацию каждого такого метода

Для метода T create(T1 t1, T2 t2, ..., Tn tn) будет сгенерирован следующий код:

Листинг 20: Общий вид кода генерируемого метода

```
public T create(T1 t1, T2 t2, ..., Tn tn)
{
        return container.create(T.class, new AbstractionInstancePair[]{
                        new AbstractionInstancePair(T1.class, t1),
                        new AbstractionInstancePair(T2.class, t2),
                        ...,
                        new AbstractionInstancePair(Tn.class, tn)});
}
```

Генерация кода реализована с помощью BCEL, которая позволяет генерировать объекты типа JavaClass и сериализовывать их в массив байт (непосредственно получать байткод). Полученный байткод загружается простейшим загрузчиком классов, после чего мы можем использовать полученную реализацию в коде.

Для создания класса используется ClassGen — генератор классов. Его основные используемые методы, использованные в работе, следующие:

- addField — добавляет поле, переданное в качестве аргумента
- addMethod — добавляет метод, переданный в качестве аргумента

- getJavaClass — возвращает объект типа JavaClass, соответствующий сгенерированному классу

Поле представляет из себя класс Field, которое получается с помощью класса-генератора полей FieldGen. Ему указывается имя поля, модификаторы доступа, тип, а также другие аргументы.

Метод представляет из себя класс Method, который получается с помощью класса-генератора методов MethodGen. Генератору методов указываются модификаторы доступа, тип возвращаемого значения, типы и имена аргументов, имя метода, имя класса, которому будет принадлежать метод, список инструкций и другие аргументы.

Список инструкций — список кодов команд, которые являются внутренним представлением исходного кода в JVM. По сути это некоторый аналог ассемблера для языка Java. Для построения списков инструкций есть достаточно удобные встроенные в BCEL инструменты. Детали реализации можно посмотреть в исходном коде проекта.

4 Заключение

В данной работе представлено описание реализованного DI-контейнера для языка Java, обладающего следующими отличительными особенностями:

- минимальность конфигурации — пользователю не требуется явно перечислять все классы, используемые в приложении, и настраивать способ их получения: в большинстве случаев контейнер сам принимает соответствующее решение
- возможность автоматической генерации классов-фабрик

Полученный контейнер является законченной библиотекой и может быть внедрен как в новые проекты, так и в уже существующие.

При текущей реализации граф наследования не изменяется в ходе работы приложения, в частности, туда невозможно добавить новые классы. Реализация возможности динамичности графа зависимостей добавит гибкости контейнеру и позволит расширить круг решаемых им задач.

Автор выражает благодарность научному руководителю Бурмистрову Ивану Сергеевичу за ценные советы по качеству кода и оформлению работы, Хворосту Алексею Александровичу за возможность внедрения контейнера в реальный сервис и ценные замечания по поводу работы, а также Клепинину Александру Владимировичу за ценные консультации по тонкостям языка Java и ценные советы по оформлению работы.

Список литературы

[1] *R. Martin* Design Principles and Design Patterns [Электронный ресурс]. Режим доступа:
http://www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf

[2] Документация по Spring IoC container [Электронный ресурс]. Режим доступа:
http://static.springsource.org/spring/docs/3.0.x/reference/beans.html

[3] Документация по Google Guice[Электронный ресурс]. Режим доступа:
https://code.google.com/p/google-guice/wiki/TableOfContents

[4] *Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra* Head First Design Patterns, O'Reilly Media, 2004

[5] Byte Code Engineering Library [Электронный ресурс]. Режим доступа:
http://commons.apache.org/proper/commons-bcel/