Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Увеличение модульности программного обеспечения на языке Java.

Курсовая работа студента 345 группы
Абишева Тимура Маратовича

Научный руководитель
профессор

В. О. Сафонов

Санкт-Петербург
2010

Оглавление

1. Введение

1.1. Язык Java.

Java — объектно-ориентированный язык программирования, разработанный компанией Sun Microsystems. Датой официального выпуска называют 23 мая 1995 года [1]. С момента появления язык пережил множество обновлений и текущей его версией является версия 1.6 [1]. Программы на языке Java транслируются в так называемый байт-код, который является платформо-независимым. Далее данный байт-код запускается на виртуальной машине Java также известной как jvm. Благодаря такой технологии запуска, приложения на языке являются кросс-платформенными, и программа написанная, к примеру, на Windows будет также работать на Linux и Mac OS X.

Язык Java имеет множество возможностей, вот основные из них:

1. автоматическое управление памятью (благодаря сборщику мусора)
2. большая встроенная библиотека, содержащая, к примеру, реализации множества коллекций, алгоритмов и средств работы с сетью и многопоточностью.
3. Поддержка параметризованный типов (generics) делающая программы более безопасными и простыми

Основной парадигмой программирования на Java является объектно-ориентированное программирование.

На данный момент Java является одним из самых распространенных языков программирования, её использование простирается от огромных бизнес приложений до простых утилит.

1.2. Постановка и актуальность задачи.

Увлечение модульности программного обеспечения и уменьшения его связанности стояло в программном обеспечении испокон веков. С появлением test-driven development [2] (разработкой через тестирование) важность модульности и малой связанности программ лишь возросла. Давайте разберемся с текущими уровнями модульности в мире Java:

1. первый уровень — уровень класса. Так как класс инкапсулирует в себя состояние он и является наименьшей замкнутой частью программного обеспечения — его наименьшим модулей.
2. второй уровень — уровень пакета. Пакет инкапсулирует в себя набор классов, и своими публичными классами он как бы предоставляет свой интерфейс. Это тоже модуль программного обеспечения, но уже чуть большего размера.

3. ну и наконец — вся программа.

Решая задачи промышленного программирования я обратил внимание на нехватку модулей размером больше чем уровень пакета. Рассмотрим простейший пример: для управления отложенными заданиями в программном обеспечении использует какая-либо tms (система управления временем). Пусть например она хранит свои данные в базе данных mysql и ее основной модуль называется tms-core. Его зависимостью будет настройка базы данных. Также иногда хочется управлять текущими задачами с помощью веб-интерфейса, назовем этот модуль для этого tms-webui и его зависимостью будет tms-core.

Целью курсовой было создания удобного программного фреймворка для обеспечения такого рода зависимостей с помощью введенного мною понятия модуля, а также с помощью принципа обращения контроля.

Данная задача безусловно является актуальной в современном мире, в следствии разработки огромных программных проектов, которые требуют новый уровень модульности для облегчения повторного использования и тестирования программного обеспечения.

1.3. Средства разработки.

В качестве средства разработки использовалась бесплатная и свободно распространяемая интегрированная среда разработки IntelliJ IDEA Community Edition [3]. Выбор в её пользу был сделан благодаря мощной поддержки разработки на языке Java данной средой программирования. IDEA предоставляет множество средств для написания и рефакторинга Java кода, прекрасные средства для отладки и тестирования, а также поддержку систем контроля версий. В качестве операционной системы использованной во время разработки был выбран Linux, в следствие свой бесплатности и удобной разработки под ним. Для разработки не использовались никакие уже существующие каркасы приложений для Java, так как по сути создавался этот самый каркас, однако в будущем предполагается использовать log4j для записи логов в приложении.

2. Решение поставленных задач

2.1. Понятие принципа обращения контроля.

Принцип обращения контроля является важным принципом объектно-ориентированного программирования и используется для уменьшения связанности в программном обеспечении. Очевидно что задача уменьшения связанности тесно связанна с задачей увеличения модульности, а именно уменьшая связанность мы облегчаем для себя задачу выделения отдельных модулей.

Поясним принцип обращения контроля на примере:

Пусть у нас есть интерфейсы IReader, IWriter и ICopier. Если раньше в реализации ICopier мы явно указывали какие реализации IReader и IWriter надо использовать, то теперь мы эти зависимости будут описываться в отдельном месте и проставляться контейнером зависимостей. В случае прямого обращения к контейнеру и реализующих классов реализацию называют Service Locator, а в случае пассивного Dependency Injection (далее DI, внедрение зависимостей). Приведем пример кода с DI:

Интерфейсы:

```
public interface ICopier {
    void copy();
}
public interface IReader {
    String read();
}
public interface IWriter {
    void write(String data);
}
```

И реализации:

```
public class Copier implements ICopier {
    private IReader reader;
    private IWriter writer;

    private String prefix;
    private int repeatCount;
```

```
    public Copier() {
        System.out.println("copier constructor");
    }

    public void setReader(IReader reader) {
        this.reader = reader;
    }

    public void setWriter(IWriter writer) {
        this.writer = writer;
    }

    public void setPrefix(String prefix) {
        this.prefix = prefix;
    }

    public void setRepeatCount(int repeatCount) {
        this.repeatCount = repeatCount;
    }

    public void copy() {
        writer.write(prefix);
        for (int i = 0; i < repeatCount; i++) {
            writer.write(reader.read());
        }
    }
}
public class Reader implements IReader {
    public String read() {
        return "Test";
    }
}
public class Writer implements IWriter {
    public void write(String data) {
```

```
        System.out.println(data);
    }
}
```

Благодаря использованию принципа обращения контроля у нас исчезла привязка реализации ICopier к реализациям IWriter и IReader, они теперь проставляются в конфигурации контейнера. К примеру так:

```
public class Example extends JavaModuleConfig {
    protected void config() {
        Bean repeatCount = importBean("repeatCount", Integer.class);
        Bean prefix = importBean("prefix", String.class);

        bean("writer").withClass(Writer.class).finish();
        bean("reader").withClass(Reader.class).finish();

        Bean copier = notLazySingletonBean("copier").withClass(Copier.class)
                .set("Writer").toBean("writer")
                .set("Reader").toBean("reader")
                .set("RepeatCount").to(repeatCount)
                .set("Prefix").to(prefix)
                .finish();

        exportBeans(copier);
    }
}
```

2.2. Введение понятия модуль и бин в программное обеспечение.

В своей работе я ввожу два новых понятия и на их основе получаю более модульное программное обеспечение.

Начнем с понятия бина. Бин является наименьшей структурной единицей моего фреймворка, и по сути является просто экземпляром объекта. Бин описывается именем своего класса и переменных необходимых для его создания, это могут быть строки, числа, либо другие бины.

Пример описания бина:

```
        bean("writer").withClass(Writer.class).finish();
```

Модулем мы будем называть набор отображение с одного наборов бинов (входных) в другой (выходной). Модуль и является собственно модулем в программном обеспечении. К примеру, в ранее описаном tms модуль tms-core будет иметь входным бином подключение к базе данных, а выходным — бин для управления tms. Tms-webui же будет принимать во вход бин управления и предоставлять Servletы или что-либо подобное. Пример описание модуля можно увидеть выше.

2.3. Абстрагирование понятий модуля и его конфигурации.

В своей работе мне хотелось достичь независимость контейнера от формата его описания, при достижении такой независимости можно было описывать контейнеры как с помощью xml так и прямо в коде java. Для этого был определен интерфейс модуля:

```
public interface Module {
   BeanContainer getExportedBeans(BeanContainer importedBeans) throws
NotAllImportedBeansException;
}
```

То есть модуль есть отображение одного набора бинов в другой.

Также был определен настраиваемый модуль и его конфигурация:

```
public interface ModuleConfig {
   void config(ConfigurableModuleContext context);
}
```

С помощью контекста ModuleConfig способен конфигурировать бины.

Я реализовал два варианта ModuleConfig - JavaModuleConfig и XMLModuleConfig, соответственно для конфигурации посредством Java кода и кода XML.

2.4. Создание исполняемого и проверочного контекста.

Для возможности тестирования создаваемых модулей были созданы два различных контекста для конфигураций — контекст исполняемый и контекст тестирующий. Если мы при конфигурации модуля передадим в качестве контекста контекст тестирующий, бины создаваться не будут, однако будет проверяться совпадение типов и корректность объявлений. Если же передать исполняемый контекст, то бины создадутся и будут готовы к использованию. Чтобы увидеть как это используется можно взглянуть на реализацию ConfigurableModule:

```
public class ConfigurableModule implements Module {
```

```
private final ModuleConfig config;

    public ConfigurableModule(ModuleConfig config) {
        this.config = config;

        CheckingContext context = new CheckingContext();
        config.config(context); // checking config
    }

public BeanContainer getExportedBeans(BeanContainer importedBeans) throws
NotAllImportedBeansException {
        RunnableContext context = new RunnableContext(importedBeans);
        config.config(context);
        return context.getExportedBeans();
    }
}
```

Как мы видим тут, корректность модуля будет проверятся на момент его создания, а не использования. Таким образом мы можем проверять модули без поддержки данного действия со стороны IDE, даже если пишем программное обеспечение в простом редакторе кода, например, в gedit.

3. Заключение

3.1. Результат.

В результате работы был разработан фреймворка для управления зависимостями в программном обеспечении на языке Java.

Его основными возможности:

1. реализация внедрения зависимостей
2. возможность написание конфигураций как на Java так и на XML.
3. возможность проверки конфигураций, без поддержки фреймворка средой программирования, а также независимость этой возможности от формата конфигурации.
4. реализация понятия модуля, способного зависеть от других модулей.

Также мною был получен огромный опыт реализации и проектирования большого программного обеспечения на языке Java. Я получил опыт использования технологии Java Reflection.

Скачать фреймворк можно по адресу http://github.com/ttim/tcontainer.

3.2. Сравнение с существующими решениями.

Было просмотрены три крупных IoC фреймворка на языке Java: Google Guice [4], SpringSource Spring [5] и PicoContainer [6]. Понятие модуля с зависимостями в виде похожим на мой было найдено только в PicoContainer, однако PicoContainer предлагает лишь строгую иерархическую структуру модулей, в которых родительский модуль (в терминологии PicoContainer — контейнер) имеет доступ ко всем бинам зависимых от него модулей.

3.3. Развитие.

На данный момент есть только реализация внедрения зависимостей с помощью setter-getter инъекций, поэтому хотелось бы добавить еще инъекции посредством конструктора. Также с помощью аннотаций можно значительно увеличить читабельность кода для внедрения зависимостей.

Эти два направления я считаю основными для дальнейшего развития.

Список литературы

[1] Java//Официальный сайт Java URL: http://java.sun.com/ (дата обращения: 29.05.2010)

[2] Test-driven development//Статья на wikipedia.org URL: http://ru.wikipedia.org/wiki/Разработка_через_тестирование (дата обращения: 29.05.2010)

[3] IntelliJ IDEA Community Edition//Официальный сайт URL: http://www.jetbrains.com/idea/ (дата обращения: 29.05.2010)

[4] Google Guice//Проект на Google Code URL: http://code.google.com/p/google-guice/ (дата обращения: 29.05.2010)

[5] SpringSource Spring//Официальный сайт URL: http://www.springsource.com/ (дата обращения: 29.05.2010)

[6] PicoContainer//Официальный сайт URL: http://www.picocontainer.org/ (дата обращения: 29.05.2010)