

# Инвертирование зависимостей

JSP & Servlets  
14 уровень, 8 лекция

ОТКРЫТА

## 9.1 Dependency Inversion

Помнишь, мы когда-то говорили, что в серверном приложении нельзя просто так создавать потоки через `new Thread().start()`? Потоки должен создавать только контейнер. Теперь мы разовьем эту мысль еще сильнее.

Все объекты тоже должен создавать только контейнер. Конечно, речь не идет обо всех объектах, а скорее о так называемых бизнес-объектах. Их еще часто называют бинами. Ноги этого подхода растут из пятого принципа SOLID, который требует избавляться от классов и переходить на интерфейсы:

- Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Реализация должна зависеть от абстракции.

Модули не должны содержать ссылки на конкретные реализации, а все зависимости и взаимодействие между ними должно строиться исключительно на основе абстракций (то есть интерфейсов). Саму суть этого правила можно записать одной фразой: **все зависимости должны быть в виде интерфейсов**.

Несмотря на свою фундаментальность и кажущуюся простоту, это правило нарушается чаще всего. А именно, каждый раз, когда в коде программы/модуля мы используем оператор `new` и создаем новый объект конкретного типа, тем самым вместо зависимости от интерфейса образуется зависимость от реализации.

Понятно, что этого нельзя избежать и объекты где-то должны создаваться. Но, по крайней мере, нужно свести к минимуму количество мест, где это делается и в которых явно указываются классы, а также локализовать и изолировать такие места, чтобы они не были разбросаны по всему коду программы.

Очень хорошим решением является безумная идея о том, чтобы сконцентрировать создание новых объектов в рамках специализированных объектов и модулей — фабрик, сервис локаторов, IoC-контейнеров.

В каком-то смысле такое решение следует Принципу единственного выбора (Single Choice Principle), который говорит: **"Всякий раз, когда система программного обеспечения должна поддерживать множество альтернатив, их полный список должен быть известен только одному модулю системы"**.

Поэтому, если в будущем придется добавить новые варианты (или новые реализации, как в рассматриваемом нами случае создания новых объектов), то достаточно будет произвести обновление только того модуля, в котором содержится эта информация, а все остальные модули останутся незатронутыми и смогут продолжать свою работу как обычно.

### Пример 1

Было бы разумно вместо `new ArrayList` писать что-то типа `List.new()`, JDK подставила бы вам правильную реализацию листа: `ArrayList`, `LinkedList` или даже `ConcurrentList`.

Например, компилятор смотрит, что к объекту есть обращения из различных потоков и ставит туда потоко-безопасную реализацию. Или слишком много вставок в середину листа, тогда реализация будет основана на `LinkedList`.

### Пример 2

Это уже произошло с сортировками, например. Когда последний раз ты писал алгоритм сортировки для сортировки коллекции? Вместо этого теперь все пользуются метод `Collections.sort()`, а элементы коллекции должны поддерживать интерфейс `Comparable` (сравниваемый).

Если в метод `sort()` передать коллекцию из меньше чем 10 элементов, ее вполне можно отсортировать сортировкой пузырьком (Bubble sort), а не Quicksort.

### Пример 3

Компилятор уже следит за тем, как ты конкатенируешь строки и заменят ваш код на `StringBuilder.append()`.

## 9.2 Инвертирование зависимостей на практике

Теперь самое интересное: давай подумаем, как нам совместить теорию и практику. Каким образом модули могут корректно создавать и получать свои “зависимости” и не нарушать Dependency Inversion?

Для этого при проектировании модуля ты должен решить для себя:

- что модуль делает, какую функцию выполняет;
- то модулю нужно от его окружения, то есть с какими объектами/модулями ему придется иметь дело;
- и как он это будет получать.

Чтобы соблюсти принципы Dependency Inversion тебе обязательно нужно определиться с тем, какие внешние объекты использует ваш модуль и как он будет получить на них ссылки.

И тут возможны следующие варианты:

- модуль сам создает объекты;
- модуль берет объекты из контейнера;
- модуль понятия не имеет откуда берутся объекты.

Проблема в том, что для создания объекта необходимо вызвать конструктор конкретного типа, и в результате модуль будет зависеть не от интерфейса, а от конкретной реализации. Но если мы не хотим, чтобы в коде модуля объекты создавались явно, то можно использовать паттерн Фабричный Метод (**Factory Method**).

"Суть заключается в том, что вместо непосредственного инстанцирования объекта через `new`, мы предоставляем классу-клиенту некоторый интерфейс для создания объектов. Поскольку такой интерфейс при правильном дизайне всегда может быть переопределен, мы получаем определенную гибкость при использовании низкоуровневых модулей в модулях высокого уровня".

В случаях, когда нужно создавать группы или семейства взаимосвязанных объектов, вместо Фабричного Метода используется Абстрактная Фабрика (**Abstract factory**).

## 9.3 Использование Service Locator

Модуль берет необходимые объекты у того, у кого они уже есть. Предполагается, что в системе есть некоторый репозиторий объектов, в который модули могут “класть” свои объекты и “брать” объекты из репозитория.

Этот подход реализуется шаблоном Локатор Сервисов (**Service Locator**), основная идея которого заключается в том, что в программе имеется объект, знающий, как получить все зависимости (сервисы), которые могут потребоваться.

Главное отличие от фабрик в том, что Service Locator не создает объекты, а фактически уже содержит в себе инстанцированные объекты (или знает где/как их получить, а если и создает, то только один раз при первом обращении). Фабрика при каждом обращении создает новый объект, который ты получаешь в полную собственность и можешь делать с ним что хочешь.

**Важно!** Локатор сервисов выдает ссылки на одни и те же уже существующие объекты. Поэтому с объектами, выданными Service Locator, нужно быть очень осторожным, так как одновременно с тобой ими может пользоваться кто-то еще.

Объекты в Service Locator могут быть добавлены напрямую через конфигурационный файл да и вообще любым удобным программисту способом. Сам Service Locator может быть статическим классом с набором статических методов, синглтоном или интерфейсом и передаваться требуемым классам через конструктор или метод.

Service Locator иногда называют антипаттерном и не рекомендуют использовать (потому что он создает неявные связности и дает лишь видимость хорошего дизайна). Подробно можно почитать у Марка Симана:

- [Service Locator is an Anti-Pattern](#)
- [Abstract Factory or Service Locator?](#)

## 9.4 Dependency Injection

Модуль вообще не заботится о “добывании” зависимостей. Он лишь определяет, что ему нужно для работы, а все необходимые зависимости ему поставляются (внедряются) извне кем-то другим.

Это так и называется — **Внедрение Зависимостей** (Dependency Injection). Обычно требуемые зависимости передаются либо в качестве параметров конструктора (Constructor Injection), либо через методы класса (Setter injection).

Такой подход инвертирует процесс создания зависимости — вместо самого модуля создание зависимостей контролирует кто-то извне. Модуль из активного эмитента объектов становится пассивным — не он создает, а для него создают другие.

Такое изменение направления действия называется **Инверсия Контроля (Inversion of Control)**, или Принцип Голливуда — “Не звоните нам, мы сами вам позвоним”.

Это **самое гибкое решение, дающее модулям наибольшую автономность**. Можно сказать, что только оно в полной мере реализует “Принцип единственной ответственности” — модуль должен быть полностью сфокусирован на том, чтобы хорошо выполнять свою функцию и не заботиться ни о чем другом.

Обеспечение модуля всем необходимым для работы — это отдельная задача, которой должен заниматься соответствующий “специалист” (обычно управлением зависимостями и их внедрениями занимается некий контейнер — IoC-контейнер).

По сути, здесь все как в жизни: в хорошо организованной компании программисты программируют, а столы, компьютеры и все необходимое им для работы покупает и обеспечивает офис-менеджер. Или, если использовать метафору программы как конструктора — модуль не должен думать о проводах, сборкой конструктора занимается кто-то другой, а не сами детали.

Не будет преувеличением сказать, что использование интерфейсов для описания зависимостей между модулями (Dependency Inversion) + корректное создание и внедрение этих зависимостей (прежде всего Dependency Injection) являются **ключевыми техниками для снижения связанности**.

Они служат тем фундаментом, на котором вообще держится слабая связанность кода, его гибкость, устойчивость к изменениям, переиспользование, и без которого все остальные техники имеют мало смысла. Это основа основ слабой связности и хорошей архитектуры.

Принцип Inversion of Control (вместе с Dependency Injection и Service Locator) детально разбирается Мартином Фаулером. Есть переводы обеих его статей: ["Inversion of Control Containers and the Dependency Injection pattern"](#) и ["Inversion of Control"](#).



Комментарии (3)

популярные    новые    старые

JavaCoder

Введите текст комментария

Andrey Panchenko    Моет полы в Яндекс

19 сентября 2022, 11:17    ⋮

После самостоятельного написания нескольких приложений на Spring Boot, я подумал, что понимаю, как оно работает, но начал конкретно буксовать на собеседовании на простых вопросах: "что такое бины? что такое DI, IoC? как ты у интерфейсов вызывал методы? зачем ты пишешь эти аннотации @service и другие?". Эта статья начинает открывать мне глаза. Но я вообще не понимаю, как это читают люди без опыта написания приложений с использованием эти принципов, потому что он у меня есть, и я с трудом понимаю!

Ответить

15 августа 2022, 17:48

...

Вообще все статьи этого уровня прям классные, спасибо за них!

Ответить

+5

Ответить

15 августа 2022, 17:45

...

Докину сюда еще одну шикарную статью. По истине может быть первой для "понимания" что такое DI (Dependency Injection), IoC (Inversion of Control) и DI (Dependency Inversion) [Инверсии зависимостей управления впрыском](#)

Точнее не так, я вас обманул .

В первый раз вы ничего не поймете, но в вашем пространстве появятся такие акронимы как DI, IoC и DI, в которые будет необходимо копать. Приятное ощущение на самом деле, чем-то мне напоминает Java.IO (будь он проклят), concurrency и любой другой ваш "любимый" score из джавы.

Ответить

+4

ОБУЧЕНИЕ

- Курсы программирования
- Курс Java
- Помощь по задачам
- Подписки
- Задачи-игры

СООБЩЕСТВО

- Пользователи
- Статьи
- Форум
- Чат
- Истории успеха
- Активности

КОМПАНИЯ

- О нас
- Контакты
- Отзывы
- FAQ
- Поддержка



JavaRush — это интерактивный онлайн-курс по изучению Java-программирования с нуля. Он содержит 1200 практических задач с проверкой решения в один клик, необходимый минимум теории по основам Java и мотивирующие фишки, которые помогут пройти курс до конца: игры, опросы, интересные проекты и статьи об эффективном обучении и карьере Java-девелопера.

ПОДПИСЫВАЙТЕСЬ

ЯЗЫК ИНТЕРФЕЙСА

Русский

▼

СКАЧИВАЙТЕ НАШИ ПРИЛОЖЕНИЯ

