Качество архитектуры

Выделяют, по меньшей мере, три характерных признака плохого дизайна архитектуры:

- Жесткость;
- Хрупкость;
- Монолитность.

Жесткость дизайна – это его сопротивление к изменениям. Изменение в одном модуле системы, влечет за собой необходимость внесения каскадных изменений в другие модули. Как правило, в такой ситуации, разработчик постоянно ошибается при оценке времени необходимого для выполнения той или иной задачи по сопровождению системы.

Дизайн считается **хрупким**, если внесение изменений в один модуль приводит к поломке другого модуля, который, на первый взгляд, не связан с тем, который мы меняли. Такие «сюрпризы» так же нередко затягивают сроки сдачи проекта, и не дают расширять функциональность системы так, как того требуют обстоятельства.

Монолитность дизайна характеризуется сопротивлением к повторному использованию кода. Разработчики всегда стараются, как можно более эффективно использовать свой код. Но в случае если компоненты системы сильно связаны между собой, вам не удастся использовать отдельный компонент в контексте отличном от уже существующего в вашей системе.

В противопоставление плохому дизайну, хороший дизайн архитектуры должен быть **гибким**, **устойчивым**, и приспособленным к **повторному использованию**. Чем ниже взаимосвязь компонентов приложения друг с другом, тем выше гибкость и мобильность всего приложения в целом. Приложения, характеризующиеся высоким коэффициентом мобильности, позволяют применять свои компоненты вновь и вновь для решения однотипных задач. Это ведет к снижению дублирования кода. Такие приложения состоят из большого набора довольно мелких компонентов, каждый из которых выполняет малую часть работы, но выполняет ее качественно. Мелкие компоненты гораздо проще тестировать, реализовывать и сопровождать.

Соблюдение принципа инверсии зависимостей приводит к лучшей приспособленности кода к изменениям и меньшей зависимости от контекста выполнения.

Инверсия зависимостей

Инверсия зависимости – это особый вид IoC (инверсии управления), который применяется в объектно-ориентированном подходе для удаления зависимостей между классами. Зависимости между классами превращаются в ассоциации между объектами. Ассоциации между объектами могут устанавливаться и меняться во время выполнения приложения, что позволяет сделать модули менее связанными между собой.

Принцип инверсии зависимостей

1. **Модули верхнего уровня не должны зависеть от модулей нижнего уровня. Оба типа модулей должны зависеть от абстракций;**
2. **Абстракция не должна зависеть от реализации. Реализация должна зависеть от абстракции.**

Традиционные методы разработки (например, процедурное программирование) имеют тенденцию к созданию кода, в котором высокоуровневые модули, как раз, зависят от низкоуровневых. Это происходит из-за того, что одна из целей этих методов разработки – определение иерархии подпрограмм, а следовательно и иерархии вызовов внутри модулей (высокоуровневые модули вызывают низкоуровневые). Именно это является причиной низкой гибкости и закостенелости дизайна. При верном использовании, ОО методики позволяют обойти это ограничение.

Рассмотрим пример программы, которая копирует в файл данные, введенные с клавиатуры.

У нас есть три модуля (в данном случае это функции). Один модуль (иногда его называют сервис) отвечает за чтение с клавиатуры. Второй – за вывод в файл. А третий высокоуровневый модуль объединяет два низкоуровневых модуля с целью организации из работы.

Наш модуль copy может выглядеть примерно следующим образом.

```
while ( (data = readKeyboard()) != -1 )
{
  writeFile("./filename", data);
}
```

Низкоуровневые модули *readKeyboard* и *writeFile* обладают высокой гибкостью. Мы легко можем использовать их в контексте отличном от функции copy. Однако сама функция «copy» не может быть повторно использована в другом контексте. Например, для отправки данных из файла системному обработчику логов.

Используя принцип инверсии зависимостей, можно сделать модуль copy независимым от объектов источника и назначения данных. Для этого необходимо выработать абстракции для этих объектов, и сделать модули зависимыми от этих абстракций, а не друг от друга.

```
interface Reader
{
  public int read();
}

interface Writer
{
  public void write(int c);
}
```

Модуль *copy* должен полагаться только на выработанные абстракции и не делать никаких предположений по поводу индивидуальных особенностей объектов ввода/вывода.

```
while ( (data = reader.read()) != -1 )
{
  writer.write(data);
}
```

Примерно следующим образом выглядит использование нашего модуля пользователем.

```
copier = new Copier();

// Копирование данных с клавиатуры в файл
copier.run(new KeyboardReader(), new FileWriter("./filename"));
```

```
// Отправка данных из файла системному обработчику логов
copier.run(new FileReader("./filename"), new LogWriter());
```

Теперь модуль *copy* можно использовать в различных контекстах копирования. Изменение поведения модуля-копировщика достигается путем ассоциации его с объектами других классов (но которые зависят от тех же абстракций).

Несмотря на простоту выполненных нами действий мы получили очень важный результат. Теперь наш код обладает следующими качествами:

- модуль может быть использован для копирования данных в контексте отличном от данного;
- мы можем добавлять новые устройства ввода/вывода не меняя при этом модуль copy.

Таким образом, снизилась **хрупкость** кода, повысилась его **мобильность** и **гибкость.**

Формы инверсии зависимостей

Существует две формы инверсии зависимостей: активная и пассивная. Различие между ними состоит в том, как объект узнает о своих зависимостях во время выполнения.

При использовании пассивной формы (Dependency Injection – инъекции зависимостей) объект получает необходимые сервисы неявно. Эти сервисы «впрыскиваются» в пользовательский объект контейнером или самим программистом. Зависимому объекту не надо прилагать никаких усилий, все нужные сервисы он получает через свой интерфейс. Различают следующие виды пассивной инверсии зависимостей:

- инъекция с помощью конструктора (constructor injection) использует конструктор для ассоциирования объекта с конкретными реализациями абстракций. При использовании этого типа инверсии зависимостей, необходимые объекты передаются в конструктор, в качестве аргументов;
- инъекция при помощи set-метода (setter injection) требует определения отдельного set-метода для каждого из инъецируемых объектов;
- в некоторых языках программирования (Java/C#) существует возможность получить доступ к private/protected полям объекта (field injection), что может быть использовано для внедрения сервисов в зависящий объект напрямую, без использования set-методов и конструктора;
- interface injection использует интерфейсы для осуществления связывания объектов.

Активная форма (Dependency Lookup – поиск зависимостей) предполагает, что зависящий объект будет сам получать свои зависимости при помощи вспомогательных объектов. Данный подход предполагает наличие в системе общедоступного объекта (локатора, или реестра сервисов), который знает обо всех используемых сервисах.

Возможен и комбинированный подход, при котором локатор сервисов инъецируется в объект контейнером, после чего объект разрешает свои зависимости методом поиска.

При использовании пассивной инверсии зависимостей разработчик не может контролировать процесс «впрыскивания» зависимых объектов, что может в ряде случаев затруднить отладку приложения.

Использование локатора ставит другую проблему. Локатор прячет за собой все зависимости, но появляется зависимость от самого локатора. Как показывает практика, интерфейс локатора очень редко меняется, поэтому на это можно закрыть глаза. В действительности никто не боится зависимостей от классов, которые являются частью языковой платформы (например, InitialContext или DOMDocument).

Аннотации в Java

В практике программирования часто нужна возможность связать с тем или иным элементом программы (классом, методом, полем и т.п.) некоторую дополнительную информацию. Эта информация может понадобиться следующим программным средствам:
1) Компиляторы и другие инструментальные средства, работающие с исходным кодом. Например, с каждым элементом программы можно связать сведения об авторе или отменить для элемента выдачу предупреждений при компиляции (аналог прагм компилятора в языке C);
2) Инструментальные средства обработки скомпилированного кода. Например, системы объектно-реляционного преобразования или преобразования объекты - XML-документы часто используют «улучшение» байт-кода (byte-code enhancement) для внедрения в уже скомпилированные классы обращений к соответствующим библиотекам;
3) Среда выполнения скомпилированного кода. Например, EJB-контейнер может использовать дополнительную информацию для передачи экземпляру компонента необходимых ресурсов (соединения с БД и т.п.).
В языке Java до версии 1.5 для решения первого вида задач использовались инструкции в документационных комментариях. Для остальных двух задач соответствующую информацию приходилось предоставлять не в исходном коде, а во внешних файлах (например, в формате XML).
В языке Java версии 1.5 появился особый тип данных – аннотационный, призванный решить все приведенные выше задачи.

Аннотация представляет собой модификатор, состоящий из имени типа аннотации и пар «элемент-значение». Назначение аннотации состоит в связывании определенной информации с аннотируемым элементом программы.
Аннотация должна содержать пару «элемент-значение» для каждого элемента соответствующего типа аннотации, за исключением элементов со значением по умолчанию.
Аннотации могут использоваться как модификаторы в объявлениях пакета, класса, интерфейса, поля, метода, параметра, конструктора или локальной переменной.
Ситуация, когда с объявлением связано несколько аннотаций, принадлежащих к одному типу, определяется как ошибка компиляции.
Аннотации могут присутствовать только в исходном коде или также в скомпилированных классах и интерфейсах. Аннотации последнего вида также могут быть доступны во время выполнения при помощи механизма рефлексии.

Есть три вида аннотаций: нормальные, маркерные и одноэлементные.
Пример нормальной аннотации:

```
@RequestForEnhancement(
    id = 2868724,
    synopsis = "Provide time-travel functionality",
    engineer = "Mr. Peabody",
    date = "4/1/2004"
)
public static void travelThroughTime(Date destination) { ... }
```

Маркерная аннотация является сокращением нормальной аннотации для случаев, когда тип аннотации не содержит элементов (или для всех элементов определены значения по умолчанию). Пример:

```
@Preliminary public class TimeTravel { ... }
```

Одноэлементная аннотация является сокращением нормальной аннотации для случаев, когда тип аннотации содержит единственный элемент с именем `value` (или для всех элементов определены значения по умолчанию). Пример:

```
@Copyright("2002 Yoyodyne Propulsion Systems, Inc., All rights reserved.")
public class OscillationOverthruster { ... }
```

Допустимые типы элементов аннотации: все примитивные типы, `String`, `Class`, перечислимые типы, типы аннотаций и одномерные массивы значений любого из указанных выше типов. Значение `null` не допускается использовать в качестве значения элемента аннотации. Примеры:

```
// одноэлементная аннотация с элементом типа String[]
@Endorsers({"Children", "Unscrupulous dentists"})
public class Lollipop { ... }
// одноэлементная аннотация с элементом аннотационного типа
@Author(@Name(first = "Joe", last = "Hacker"))
public class BitTwiddle { ... }
// одноэлементная аннотация, использующая перечислимый тип,
// определенный внутри типа аннотации
@Quality(Quality.Level.GOOD)
public class Karma { ... }
```

Использование инверсии зависимостей в технологии EJB

В технологии EJB используются обе формы инверсии зависимостей.

Все зависимости компонента (ссылки на представления используемых компонентов или такие ресурсы, как соединения с БД, очереди сообщений, параметры инициализации) вкупе с доступными ему службами контейнера формируют окружение компонента (bean environment), доступное в виде контекста имен JNDI во внутреннем пространстве имен контейнера. Зависимости компонента хранятся в контексте имен `java:comp/env`, который не разделяется между компонентами. В качестве локатора выступает объект класса `java.naming.InitialContext`, с помощью метода `lookup()` которых экземпляр компонента может выполнять поиск зависимостей в контексте имен `java:comp/env` или служб контейнера в контексте имен `java:comp`. Также в роли локатора выступает объект типа `javax.ejb.EJBContext` (например, контекст сессионного компонента), который предоставляет доступ к службам контейнера через свои методы, а также позволяет искать зависимости по относительному пути в контексте имен `java:comp/env` с помощью метода `lookup()`.

Пример получения ссылки на экземпляр EJB-компонента из класса другого компонента:

```
@EJB(beanInterface=GetPaymentLocal.class, name="GetPayment")
@Stateful
public class StatefulBean implements StatefulLocal {
    public void processRequest(...) {
        GetPaymentLocal result = null;
        try {
            InitialContext ic = new InitialContext();
            result = (GetPaymentLocal) ic.lookup("java:comp/env/GetPayment");
        } catch (NamingException ex) {
            throw new ServletException(ex);
```

```
        }
        ...
    }
    ...
}
```

Обратите внимание на аннотацию `@EJB` для класса компонента, которая обязывает EJB-контейнер поместить в JNDI-контекст компонента запись, позволяющую получить ссылку на экземпляр компонента.

Альтернативой поиску зависимостей в JNDI-контексте или в контексте компонента является инъекция зависимостей, которая описывается в дескрипторе модуля. Контейнер выполняет инъекцию при помощи `set`-методов или непосредственно в поля экземпляра компонента. Инъекция зависимостей для экземпляра компонента производится контейнером после создания экземпляра и до вызова любого из его бизнес-методов. Для упрощения разработки инъекцию зависимостей можно описать также в исходном коде компонента с помощью аннотаций `@Resource` или `@EJB`. Данные аннотации могут содержать JNDI-имена зависимых ресурсов, либо эти имена указываются в дескрипторе модуля.

Пример получения ссылки на экземпляр EJB-компонента с помощью инъекции зависимости:

```
@Stateful
public class StatefulBean implements StatefulLocal {
    @EJB(beanName="GetPayment") private GetPaymentLocal result;
    ...
}
```

Контекст сессионного компонента

Контекст используется компонентом для обратной связи с контейнером.
Контекст сессионного компонента представлен объектом типа `javax.ejb.SessionContext`, который автоматически создается и поддерживается EJB-контейнером. Этот объект может быть получен в результате инъекции зависимостей на этапе инициализации компонента:

```
@Resource SessionContext ctx;
```

Методы интерфейса `SessionContext` можно разделить на следующие группы:
1) получение объектов представления.
`getEJBLocalHome()` – реализация локального домашнего интерфейса
`getEJBLocalObject()` – реализация локального интерфейса
`getEJBHome()` – реализация удаленного домашнего интерфейса
`getEJBObject()` – реализация удаленного интерфейса
`getBusinessObject(Class businessInterface)` – реализация указанного бизнес-интерфейса
`getInvokedBusinessInterface()` – реализация бизнес-интерфейса, через который был вызван компонент
2) получение сведений о клиенте компонента. Данные методы могут быть использованы для реализации programmatic security, когда ограничение доступа к компонентам и методам реализуются самим приложением, а не сервером приложения;
3) программное управление транзакциями;
4) поиск зависимостей в контексте имен `java:comp/env` методом `lookup()`.

Операции, разрешенные или запрещенные в различных методах сессионных компонентов.

В спецификации определены ограничения на доступ к контексту компонента, контексту имен `java:comp/env`, менеджерам ресурсов, другим EJB-компонентам и сервисам EJB-контейнера из различных методов класса компонента. Основные из них приведены в табл. 1.

	Конструктор	Обработчик события ЖЦ		Бизнес-метод[1]
		SFSB	SSB	
Получение объектов представления компонента	–	+	+	+
Сведения о клиенте компонента	–	+	+[2]/–	+
Контекст имен java:comp/env	–	+	+	+
Вызов других EJB-компонентов	–	+	–	+
Доступ к менеджерам ресурсов	–	+	–	+
Доступ к менеджеру сущностей	–	+	–	+

Примечания:
1. В том числе обработчик вызова бизнес-метода.

Так как конструктор класса компонента не предназначен выполнения каких-либо иных действий, кроме создания экземпляра класса (например, для установки зависимостей компонента), в конструкторе запрещены все указанные операции.
Различие между обработчиками событий ЖЦ сессионных компонентов с состоянием и без состоянием обусловлены тем, что первые связываются с конкретным клиентом при создании компонента вплоть до его удаления, а вторые – только на время выполнения бизнес-метода от имени клиента.
При нарушении рассматриваемых ограничений выбрасывается исключение `java.lang.IllegalStateException`.

Помимо указанных выше ограничений на реализацию бизнес-методов накладываются следующие дополнительные ограничения (табл. 2), призванные обеспечить переносимость компонентов (возможность их выполнения в любом EJB-контейнере).

№	Запрет на	Причина			
		1	2	3	4
1	Сохранение значений в статических полях класса компонента	+			
2	Использование примитивов синхронизации для синхронизации выполнения нескольких экземпляров компонента	+			
3	Вывод информации на дисплей и ввод с клавиатуры функциями AWT (Swing)				[1]
4	Доступ к файлам и каталогам в файловой системе средствами пакета `java.io`				[2]
5	Прослушивание сокетов, прием соединения по сокетам и использование сокетов для широковещания				[3]
6	Использование рефлексии для получения информации о членах классов, недоступных по правилам безопасности языка Java (например, о закрытых или защищенных методах)		+		
7	Работа с загрузчиками классов и менеджерами безопасности, останов		+	+	

	JVM, изменение потоков стандартного ввода/вывода				
8	Установка фабрики сокетов и фабрики обработчиков потока данных для URL		+	+	
9	Управление потоками и группами потоков			+	
10	Прямое чтение и запись в файловый дескриптор		+		
11	Получение политики безопасности для конкретного источника кода		+		
12	Загрузка машинно-зависимых библиотек		+		
13	Доступ к пакетам и классам, недоступным компоненту по правилам языка Java		+		
14	Программное создание класса в пакете		+		
15	Доступ и изменение объектов конфигурации безопасности (`Policy`, `Security`, `Provider`, `Signer` и `Identity`)		+		
16	Использование возможностей протокола сериализации по подстановке объектов и подклассов		+		
17	Передача `this` как параметра при вызове метода и как результата вызова метода. Вместо этого нужно передавать ссылку на объект представления, который можно получить из контекста компонента. (Так, для получения представления сессионного компонента предназначены методы `getBusinessObject()`, `getEJBObject()` и `getEJBLocalObject()` интерфейса `SessionContext`.)				4

Причины:
1 - Экземпляры компонента могут выполняться в разных JVM
2 – Потенциальное нарушение безопасности
3 – Ухудшение управляемости среды выполнения
4 – Особые причины

Примечания:
[1] Большинство серверов работает в т.н. headless-режиме, без дисплея и клавиатуры.
[2] Файловая система не подходит для доступа к данным из компонентов. Для хранения данных следует использовать менеджеры ресурсов.
[3] Архитектура EJB позволяет экземпляру компонента быть сетевым клиентом, но не сетевым сервером, потому что последнее будет конфликтовать с основной функцией компонента – обслуживанием EJB-клиентов.
[4] Клиент может взаимодействовать с EJB-компонентом только через представление.