**Инверсия зависимостей**

**https://dev64.wordpress.com/2013/05/14/dependency-inversion/**

Инверсия зависимостей — одна из важнейших идиом программирования. В русскоязычном интернете описаний этой идиомы (принципа) на удивление мало. Поэтому я решил попробовать сделать описание. Примеры буду делать на Java, в данный момент мне так проще, хотя принцип инверсии зависимостей применим к любому языку программирования.

Данное описание разработано совместно с [Владимиром Матвеевым](http://dpx-infinity.github.io/index.html) в ходе подготовки к занятиям со студентами, изучающими Java.

**Другие статьи из этого цикла:**

* [Идеальный Java класс](https://dev64.wordpress.com/2013/04/16/ideal-java-class/)
* [Паттерн Builder (Идеальный Builder)](https://dev64.wordpress.com/2013/04/17/builder-pattern/)
* [Юнит-тестирование](https://dev64.wordpress.com/2013/04/18/unit-testing/)
* [Применение статических импортов для констант](https://dev64.wordpress.com/2012/10/08/constant-interface-java-antipattern/)

Начну с определения «зависимости». Что такое зависимость? Если ваш код использует внутри себя какой-то класс или явно обращается к статическому методу какого-то класса или функции — это зависимость. Поясню примерами:

Ниже класс A внутри метода с именем someMethod() явно создает объект класса B и обращается к его методу someMethodOfB()

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | public class A {     void someMethod() {         B b = new B();         b.someMethodOfB();     }  } |

Аналогично, например класс B обращается явно к статическим полям и методам класса System:

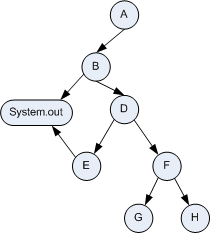
|  |  |
| --- | --- |
| 1  2  3  4  5 | public class B {      void someMethodOfB() {          System.out.println("Hello world");      }  } |

Во всех случаях когда любой класс (типа А) самостоятельно создает любой класс (типа B) или явно обращается к статическим полям или членам классов, это называют **прямой** зависимостью. Т.е. важно: если класс внутри себя работает внутри себя с другим классом — это зависимость. Если он еще и создает внутри себя этот класс, то это **прямая** зависимость.

Чем плохи прямые зависимости? Прямые зависимости плохи тем, что класс, самостоятельно создающий внутри себя другой класс, «намертво» привязывается к данному классу. Т.е. если явно написано, что B = new B();, то тогда класс А всегда будет работать именно с классом B и никаким иным классом. Или если написано System.out.println("..."); тогда класс всегда будет выводить в System.out и никуда больше.

Для небольших классов зависимости не являются страшными. Такой код вполне может работать. Но в ряде случаев, чтобы ваш класс A смог универсально работать в окружении разных классов, ему возможно могут потребоваться другие реализации классов — зависимостей. Т.е. нужен будет например не класс B, а другой класс с тем же интерфейсом, или не System.out, а например, вывод в логгер (например log4j).

Прямую зависимость можно графически отобразить таким образом:

[](https://dev64.files.wordpress.com/2013/05/dependency-inversion-1.png)

Т.е. когда вы в своем коде создаете класс А: A a = new A(); на самом деле создается не один класс А, а целая иерархия зависимых классов, пример которой на приведенной картинке. Данная иерархия «жесткая»: без изменения исходного кода отдельных классов нельзя подменить ни один из классов иерархии. Поэтому класс А в такой реализации плохо адаптируем для изменяющегося окружения. Скорее всего, его нельзя будет использовать ни в каком коде, кроме конкретно того, для которого вы его написали.

Чтобы отвязать класс А от конкретных зависимостей применяется *внедрение зависимости*. Что такое внедрение зависимости? Вместо того чтобы явно создавать нужный класс в коде, в класс А зависимости передаются через конструктор:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | public class A {      private final B b;      public A(B b) {         this.b = b;      }      public void someMethod() {         b.someMethodOfB();      }  } |

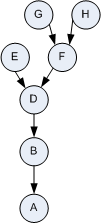
Т.о. класс А теперь получает свою зависимость через конструктор. Теперь чтобы создать класс А необходимо будет создать сначала его зависимый класс. В данном случае это B:

|  |  |
| --- | --- |
| 1  2  3 | B b = new B();  A a = new A(b);  a.someMethod(); |

Если ту же самую процедуру повторить для всех классов, т.е. в конструктор класса B передавать инстанс класса D, в конструктор класса D — его зависимости E и F, и т.д., то тогда получится код, все зависимости которого создаются в обратном порядке:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | G g = new G();  H h = new H();  F f = new (g,h);  E e = new E();  D d = new D(e,f);  B b = new B(d);  A a = new A(b);  a.someMethod(); |

Графически это можно отобразить так:

[](https://dev64.files.wordpress.com/2013/05/dependency-inversion-2.png)

Если сравнить 2 картинки — картинку выше с прямыми зависимостями и вторую картинку с внедрением зависимостей — то видно, что направление стрелочек поменялось на обратное. По этой причине идиома и называется «инверсией» зависимостей. Иными словами **инверсия зависимостей заключается в том, что, класс не создает зависимости самостоятельно, а получает их в созданном виде в конструкторе (или иным образом)**.

Чем инверсия зависимостей хороша? С инверсией зависимостей, в классе можно без изменения его кода заменить все зависимости. А это означает, что ваш класс А можно гибко настроить для применения в другой программе, отличной от той для которой он был написан изначально. Т.о. принцип инверсии зависимостей (иногда его еще называют принципом внедрения зависимостей) является ключевым для построения гибкого модульного многократно используемого кода.

Недостаток внедрения зависимостей виден тоже с первого взгляда — объекты классов, спроектированных с использованием этого паттерна, трудоемко конструировать. Поэтому обычно внедрение (инверсию) зависимостей применяют совместно с какой-либо библиотекой, предназначенной для облегчения этой задачи. Например одна из библиотек Google Guice. См. [Применение Google Guice для внедрения зависимостей](https://dev64.wordpress.com/2013/02/17/dependency-injection-google-guice/).