Введение в программирование на Java

Лекция 10. Организация проектов. Пакеты и модули.

Виталий Олегович Афанасьев

07 апреля 2025

Организация проектов

По мере того, как программы растут, возникает необходимость в организации всего написанного в них кода.

С одним из способов организации кода мы уже знакомы — это **классы** (и **интерфейсы**, и прочие типы данных).

Помимо этого, в Java:

- Классы/интерфейсы/... можно группировать в **пакеты**.
- Пакеты можно группировать в **модули**.

Пакеты

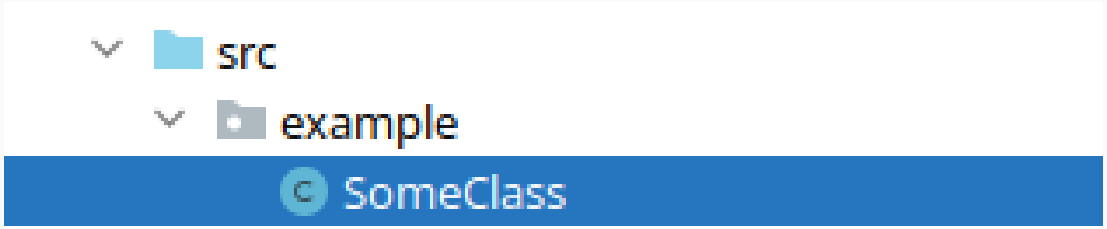
Пакеты в Java (1)

Для добавления класса/интерфейса/любого другого типа в пакет необходимо:

1. Добавить директиву package в начало файла:

```
1 package example;
2
3 public class SomeClass {
4 }
```

2. Поместить файл в директорию, соответствующую имени пакета:



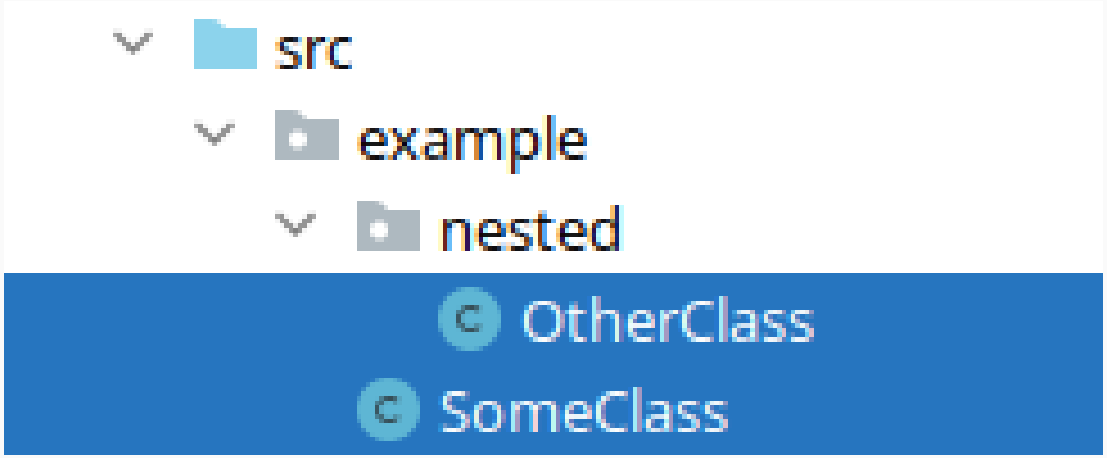
Пакеты в Java (2)

Пакеты можно вкладывать друг в друга.

В таком случае имя разделяется символом "точка":

```
1 package example.nested;
2
3 public class OtherClass {
4 }
```

Директории же просто вкладываются друг в друга:



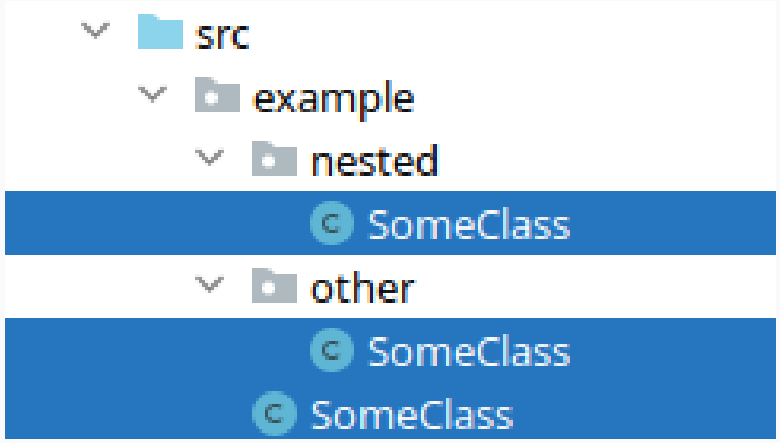
Пакеты в Java (3)

Все типы отличаются друг от друга по полному имени: имени пакета и собственному имени типа.

К примеру, полное имя следующего класса — `example.nested.OtherClass`.

```
1 package example.nested;
2
3 public class OtherClass {
4 }
```

Благодаря этому можно создавать типы с одинаковыми именами, но в разных пакетах:



Пакеты в Java (4)

Типы в одном пакете могут использовать друг друга.

```
1 package example;
2
3 public class SomeClass {
4 }
```

```
1 package example;
2
3 public class OtherClass {
4     private SomeClass someField = new SomeClass();
5 }
```

Использование типов из других пакетов (1)

Для использования типов из другого пакета есть два способа:

1. Использование директивы `import`:

```
1 package example.nested;
2
3 public class SomeClass {
4 }
```

```
1 package other.nested;
2
3 import example.nested.SomeClass;
4
5 public class OtherClass {
6     private SomeClass someField = new SomeClass();
7 }
```

Использование типов из других пакетов (2)

Для использования типов из другого пакета есть два способа:

2. Указание полного имени типа при любом использовании:

```
1 package example.nested;
2
3 public class SomeClass {
4 }
```

```
1 package other.nested;
2
3 public class OtherClass {
4     private example.nested.SomeClass someField =
5             new example.nested.SomeClass();
6 }
```

Множественное импортирование

Можно импортировать сразу все типы из пакета:

```
1 package other.nested;
2
3 import example.nested.*;
4
5 public class OtherClass {
6 }
```

Важно: если в пакет `example.nested` вложены другие пакеты, то типы из них **не импортируются**. Т.е. "звёздочка" импортирует все типы, но только на одном уровне.

Пакет java.lang

Каждая программа неявно импортирует все типы из пакета стандартной библиотеки java.lang.

```
1 package other.nested;
2
3 // import java.lang.*;
4
5 public class OtherClass {
6 }
```

В данном классе находятся уже известные нам классы:

- System
- Math
- Object
- String
- И многие другие

Статический import

При помощи конструкции `import static` можно импортировать статический метод из класса.

Тогда этот метод можно будет использовать просто по имени, не указывая при этом имя класса.

```
1 package other.nested;
2
3 import static java.lang.Math.cos;
4
5 public class OtherClass {
6     public static void main(String[] args){
7         double res = cos(0); // cos, а не Math.cos
8     }
9 }
```

Не стоит злоупотреблять этой конструкцией, т.к. код начинает хуже читаться.

Именование пакетов

Пакеты стандартной библиотеки начинаются с `java` и `javax`:

- `java.lang.Math`
- `java.util.Scanner`
- `java.math.BigInteger`
- `javax.swing.JDialog`

Для пользовательских пакетов стоит следовать официальным рекомендациям.

В частности, в качестве имени корневого пакета стоит использовать перевёрнутое доменное имя вашего веб-сайта/компании.

Например:

- `com.google.guava`
- `ru.hse.viafanasyev.coolapp`
- `org.example.myproject`

Неименованный пакет

Если директива `package` отсутствует, то такой класс принадлежит т.н. `unnamed`-пакету.

Файлы в таких пакетах могут использовать другие пакеты.

Но типы из `unnamed`-пакета могут использоваться только в `unnamed`-пакете. Другими словами, именованный пакет не может использовать неименованный.

По этой причине, при разработке даже небольших приложений **строго рекомендуется** использовать именованные пакеты.

Модификатор `package-private`

Последний модификатор доступа, с которым мы ещё не знакомы — `package-private`.

Если у типа или его члена не указан модификатор доступа, то ему присваивается модификатор `package-private`.

Сущности с таким модификатором **публичны** (доступны всем) **в рамках данного пакета**. Для файлов из других пакетов такие сущности не видны (приватны).

```
1 package example.nested;
2
3 public class SomeClass {
4     public static void foo() { ... } // Доступно всем
5     static void bar() { ... } // Доступно всем в пакете example.nested
6     private static void qux() { ... } // Доступно только в этом классе
7 }
```

Особенность модификатора protected (1)

Если сущности находятся в разных пакетах, то protected разрешает доступ только типам-наследникам.

```java
package example;
public class Car {
    protected int speed;
}
```

```java
package other;
public class Truck extends Car {
    public void accessInChild() {
        speed = 100; // OK: speed имеет protected доступ + наследование
    }
}
```

```java
package main;
public class Main {
    public static void main(String[] args){
        Car car = new Car();
        car.speed = 100; // ERROR: speed имеет protected доступ
    }
}
```

Особенность модификатора protected (2)

НО! если файлы находятся в одном пакете, то к protected-членам получить доступ может кто угодно (т.е. он становится эквивалентен модификатору public).

```
1 package example;
2 public class Car {
3     protected int speed;
4 }
```

```
1 package example; // Такой же пакет, что и у Car!
2 public class Main {
3     public static void main(String[] args){
4         Car car = new Car();
5         // OK: speed имеет protected доступ, но пакет совпадает
6         car.speed = 100;
7     }
8 }
```

Модификаторы доступа: резюме

Access Levels

Modifier	Class	Package	Subclass	World
`public`	Y	Y	Y	Y
`protected`	Y	Y	Y	N
no modifier	Y	Y	N	N
`private`	Y	N	N	N

Source:
https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

Модули

Понятие модуля

Под термином ”модуль” в программировании подразумевают разные понятия:

- Группу некоторых элементов кода
- Программные компоненты
- Пользовательские типы данных (в т.ч. классы)
- Элементы библиотек
- . . . и сами библиотеки

Понятие модуля

Под термином "модуль" в программировании подразумевают разные понятия:

- Группу некоторых элементов кода
- Программные компоненты
- Пользовательские типы данных (в т.ч. классы)
- Элементы библиотек
- ... и сами библиотеки

Модульное программирование подразумевает:

- Разделение кода на независимые и взаимозаменяемые **компоненты**
- Каждый модуль **инкапсулирует** свою внутреннюю логику
- Каждый модуль предоставляет **интерфейс** для работы с ним

Модули в Java: Причина появления (1)

Допустим, что разработчик написал библиотеку для сортировки строк по алфавиту, и решил опубликовать её.

Структура кода следующая:

- `some.sorter.StringSorter` — основной класс для сортировки. Делигирует сортировку классам из пакета `some.sorter.internal`.
- `some.sorter.internal.BubbleSorter` — внутренний класс для сортировки "пузырьком".
- `some.sorter.internal.InsertionSorter` — внутренний класс для сортировки вставками.

Классы из пакета `some.sorter.internal` являются публичными (иначе их невозможно будет использовать в другом пакете).

Модули в Java: Причина появления (2)

После публикации библиотеки разработчику захотелось изменить реализацию `StringSorter` на более эффективную и избавиться от сортировки "пузырьком".

Но так как классы были сделаны публичными, некоторые пользователи стали использовать `BubbleSorter` напрямую, хотя разработчик этого не подразумевал. Код таких пользователей перестал компилироваться.

Как тогда "спрятать" внутреннюю реализацию от пользователей?

Можно было бы хранить всё в одном пакете и использовать модификатор `package-private`, но это довольно неудобно.

Модули в Java: Причина появления (3)

Есть и другая проблема, с которой столкнулись и сами авторы JDK.

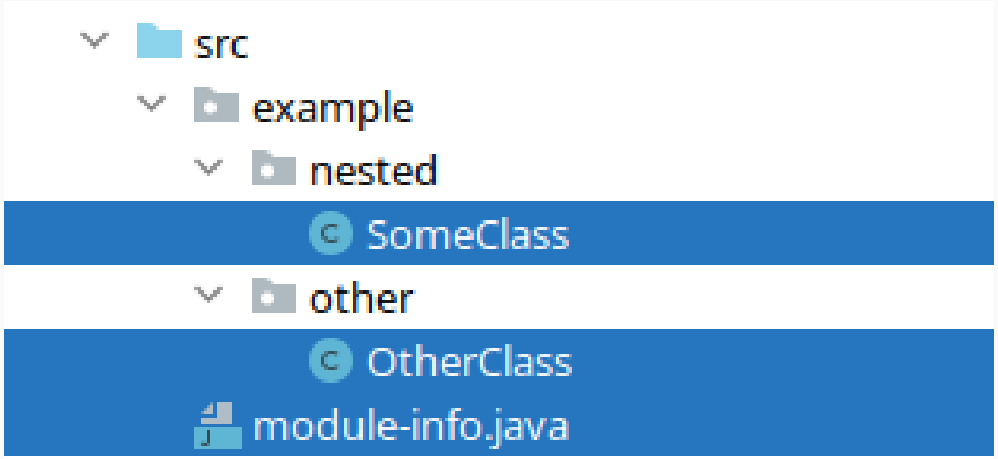
Если библиотека имеет большой размер, но пользователю необходима только малая её часть, как позволить ему не зависеть от всей огромной библиотеки?

До JDK 9 в стандартной библиотеке находились в т.ч. классы для работы с графическим интерфейсом. Но зачем консольному приложению графические библиотеки?

Модули в Java (1)

Java-модуль — это набор из пакетов.

Для создания модуля необходимо поместить файл `module-info.java` в корневую директорию модуля:



Файл `module-info.java` содержит определение модуля:

```
1 module example.module {
2 }
```

Важно: все файлы модуля должны принадлежать какому-то пакету. Неименованные пакеты в модуле не допускаются.

Модули в Java (2)

Модуль может иметь произвольное имя.

Если этот модуль планируется делать публичным, то нужно убедиться, что имя будет уникальным.

Часто используемый подход: именование по корневому пакету.

```
1 module com.google.guava {
2 }
```

Модули в Java (3)

Модули могут экспортировать те пакеты, которые должны быть доступны пользователям, при помощи директивы `exports`.

```
1 module example.module {
2     exports example.nested;
3 }
```

Модули в Java (4)

Модули могут зависеть от других модулей. Это указывается при помощи `requires`.

```
1 module user.module {
2     requires example.module;
3 }
```

В таком случае модуль `user.module` может использовать все пакеты, которые были экспортированы модулем `example.module`.

Важно: циклы в зависимостях между модулями запрещены. Если компилятор обнаружит циклическую зависимость, то такой модуль не будет скомпилирован.

Модули в Java (5)

Каждый модуль неявно зависит от стандартного модуля `java.base`.

```
1 module example.module {
2     // requires java.base;
3 }
```

В этот модуль входят, к примеру, пакеты `java.lang`, `java.util` и некоторые другие.

Безымянный модуль

Начиная с Java 9, если библиотека не определяет описание модуля, то все её классы автоматически принадлежат безымянному модулю.

Безымянный модуль экспортирует все свои пакеты и автоматически зависит от любого другого модуля.

При этом к пакетам безымянного модуля могут обращаться только другие безымянные модули. Именованные модули не могут зависеть от безымянных модулей.

Модули: Дополнительные материалы

- Horstmann K. Core Java. Volume II. 12th Edition. Chapter 9: The Java Platform Module System
- Java 9 Modules Quick Start Example
- Jenkov: Java Modules

Low Coupling, High Cohesion

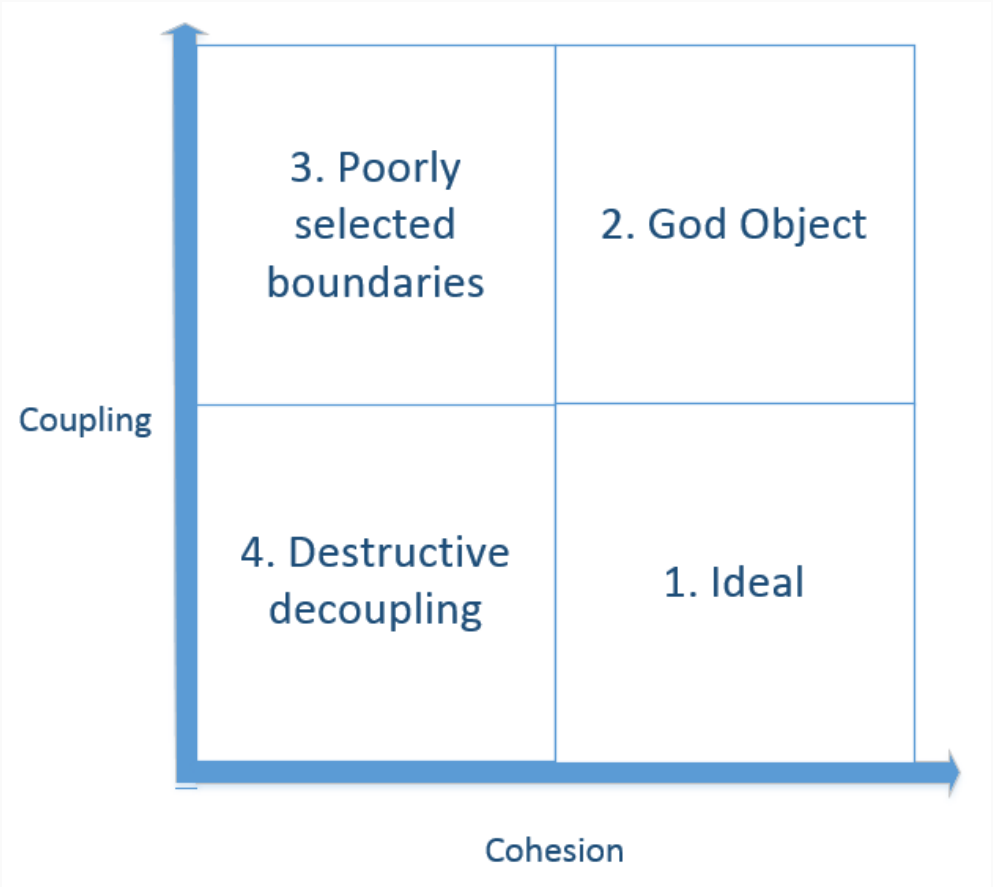
Low Coupling, High Cohesion (1)

Сущности внутри модуля должны быть сильно логически связаны (High Cohesion, Высокая Внутренняя Сцепленность/Сплочённость).

Сущности из разных модулей не должны сильно зависеть друг от друга (Low Coupling, Низкая Внешняя Связанность/Зависимость).

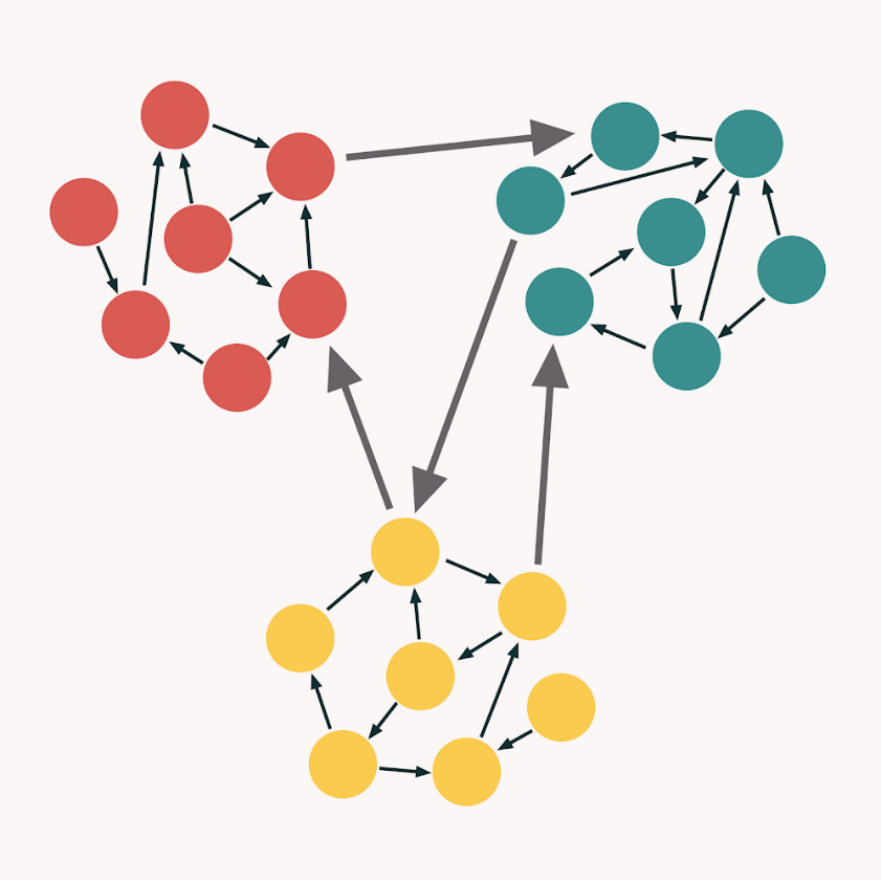
Low Coupling, High Cohesion (2)

Low Coupling, High Cohesion: идеальный случай.



Source: Cohesion и Coupling: отличия

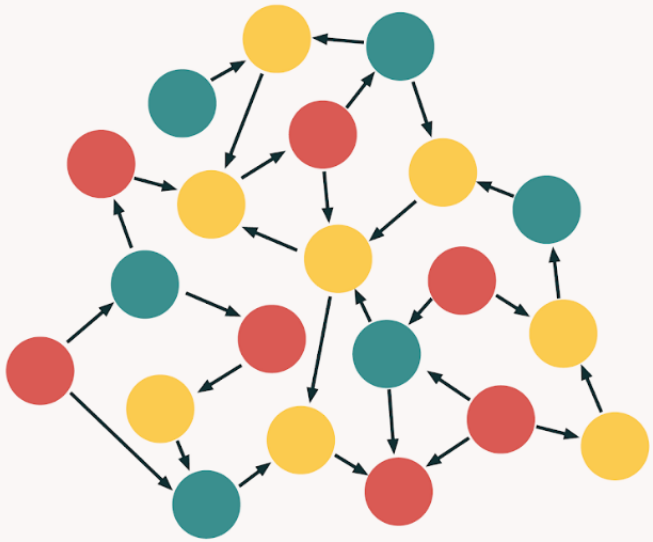
Low Coupling, High Cohesion (3)



Source: Cohesion и Coupling: отличия

Low Coupling, High Cohesion (4)

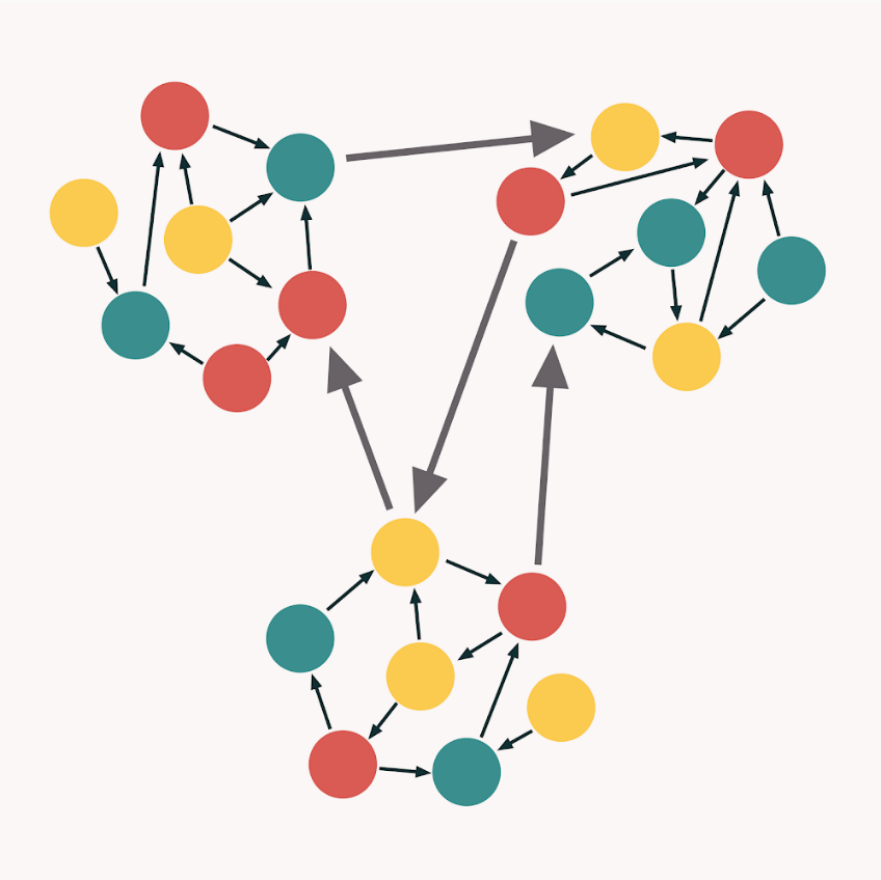
High Coupling, High Cohesion: God Object



Source: Cohesion и Coupling: отличия

Low Coupling, High Cohesion (5)

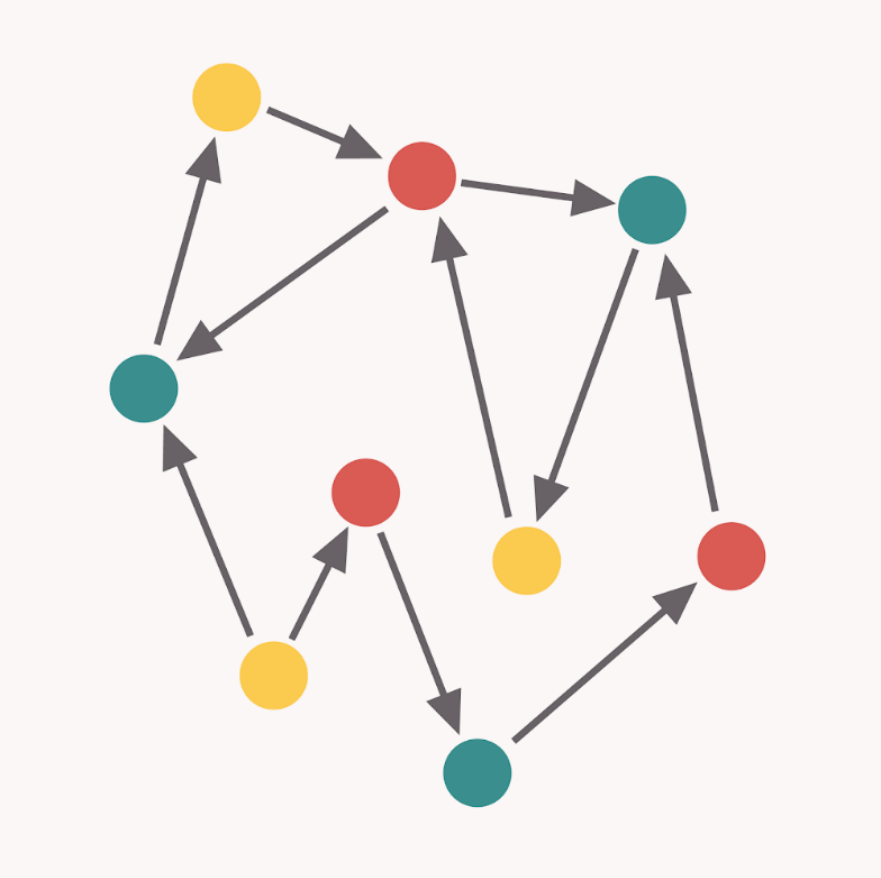
High Coupling, Low Cohesion: код, в котором присутствует какая-то модульность, но границы выбраны плохо.



Source: Cohesion и Coupling: отличия

Low Coupling, High Cohesion (6)

Low Coupling, Low Cohesion: деструктивно развязанный код. По такому коду сложно понять, чем вообще занимается система.



Source: Cohesion и Coupling: отличия