Design Principles

Don't Repeat Yourself [DRY]
Favor Composition over Inheritance
Design for inheritance, or prohibit it
High Cohesion / Low Coupling
Cohesion
Coupling
Data-encapsulation / Information Hiding
Use interfaces
Design by Interfaces
Test First Principle
Prefer many small entities over a few big ones
Law of Demeter
Single Responsibility Principle
Open-Closed Principle
Liskov Substitution Principle
Interface Segregation Principle

Don't Repeat Yourself [DRY]

Avoid duplication of code at any time.

Types of duplication:

- Simple duplication
  - Copied code fragments
- Subtle duplication
  - repeated, similar conditions
  - different implementations of the same algorithm

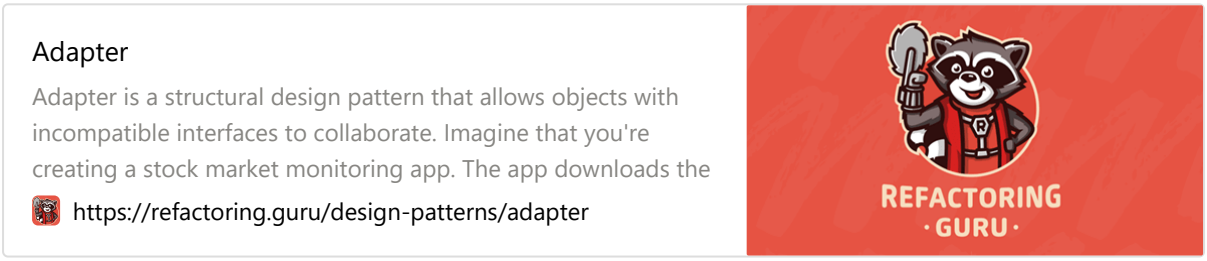
Favor Composition over Inheritance

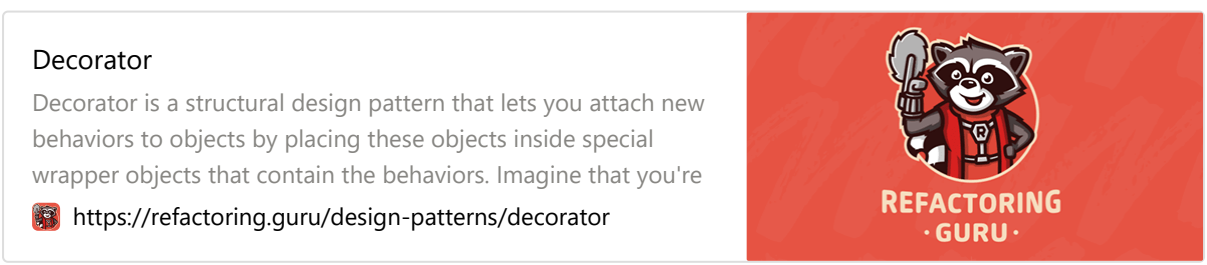
Favor Composition over Inheritance (FCoI)

> 💡 In case of doubt: Prefer composition (part-of) over inheritance (is-a)

Don't specialise any classes if they aren't specifically built for this purpose (e.g. mentioned in the docs) even if the don't restrict it (non-`final` class)

Composition leads to more flexible and maintainable code in the long run. Parts of the class / functionalities can be exchanged more easily with less side effects.
Patterns leveraging composition:





Design for inheritance, or prohibit it

If a class allows inheritance it should be designed for it and have the according docs which specify how the sub-class must be implemented. If this isn't the case prevent inheritance in the first place (`final` class, `private` -ctor)

If possible don't call methods which can be overridden, constructors or methods with similar behaviour (`clone()`) ⇒ danger of prenatal communication.

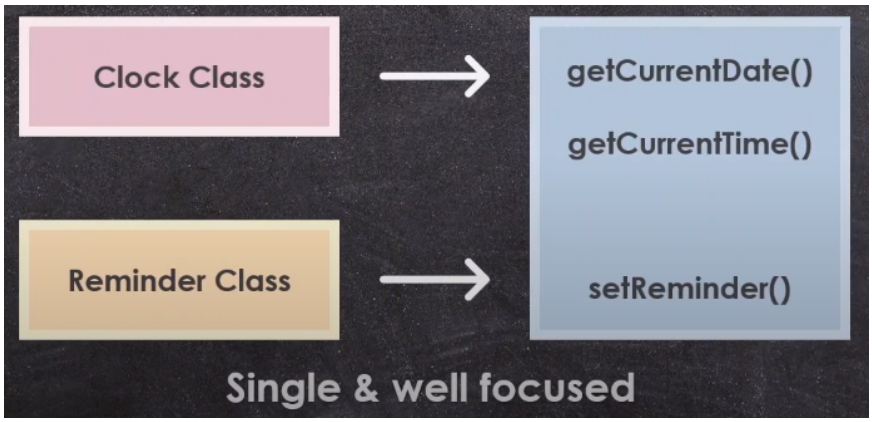
Only allow the implementation of abstract methods.

High Cohesion / Low Coupling

The cohesion of a class should be as high as possible and the coupling should be as minimal as possible.

Cohesion

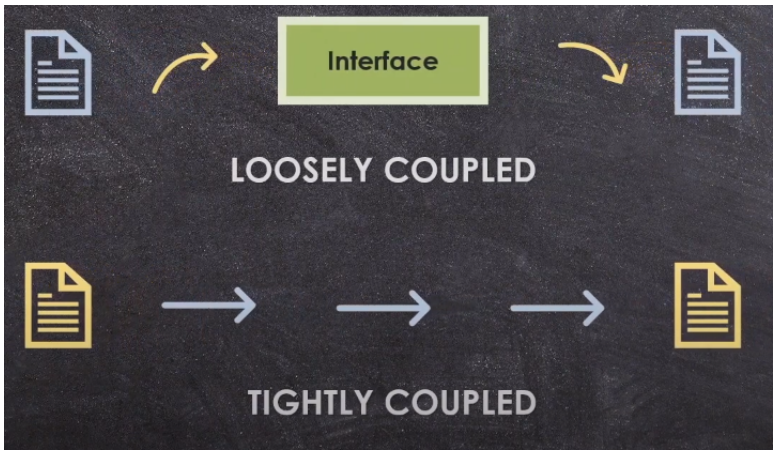
Cohesion identifies the relation between components of a module. If the module contains components which have weak correlation to the other components, or none at all, a low cohesion is achieved. A low cohesion implies that the module could be split up. The highest possible cohesion is achieved if the components of the module can't be split up any further.



⇒ the higher cohesion, the better

Coupling

Coupling identifies the relationships/dependencies of a component. The higher the coupling is, the stronger these component depend on each other. A high coupling leads to a bad maintainability and testability.



⇒ the lower (looser) the coupling, the better

Data-encapsulation / Information Hiding

Both concepts lead to code with a high cohesion and low coupling.

Data Encapsulation und Information Hiding sind beide Konzepte in der objektorientierten Programmierung (OOP), aber sie unterscheiden sich in ihrem Zweck und ihrer Anwendung.

Data Encapsulation bezieht sich auf das Verpacken von Daten und Funktionalität in einem einzigen Objekt. Dies hilft, die Daten und Funktionalität des Objekts vor unerwarteten Änderungen und Fehlern zu schützen, indem es den Zugriff auf die internen Daten und Funktionalität des Objekts beschränkt.

Information Hiding bezieht sich darauf, Teile eines Systems vor dem Zugriff oder der Änderung von anderen Teilen des Systems zu verbergen. Hier geht es darum, Abhängigkeiten zwischen den Teilen des Systems zu verringern und die Wartbarkeit und Skalierbarkeit des Systems zu verbessern.

Zusammenfassend sind Data Encapsulation und Information Hiding zwei wichtige Konzepte in der OOP, die beide dazu beitragen, eine saubere und gut strukturierte Architektur für softwarebasierte Systeme zu schaffen.

Use interfaces

Interfaces abstract the actual implementation details from the functionality it requires. Thus leading to loosely coupled code if done right.

Favor interfaces over abstract classes ⇒ Favor Composition over Inheritance

Benefits the Test First Principle

Design by Interfaces

One should focus first on what should be achieved and later on how one can achieve it. For that purpose designing the interfaces first can be hugely beneficial. It gives one a better understanding of the task at hand and can highlight some pitfalls and edge-cases early on. When applied in larger teams the task can be split up in different parts when the common interfaces (”contracts”) are defined ⇒ leverages the productivity.

Test First Principle

Test First Principle (TFP) / Test Driven Development (TDD)

> One doesn’t test to find bugs later. One does test to ensure everything runs in order.

Before implementing a feature, one should write the test cases first. With that one defines clear contracts how the code has to behave and gains a better understanding of the task at hand. In addition edge-cases become more obvious early on.

Prefer many small entities over a few big ones

An entities in this context stands for: modules, packages, classes and methods

Many smaller entities have a better testability and a higher cohesion in general, than a single big one. In addition one can understand smaller entities more easily than “god”-classes.

Refactoring is an ongoing process (see Boy Scout Rule).

Law of Demeter

Law of Dementer (LoD)

A module should not know about the innards of the objects it manipulates

The law states that a method F of a class C should only call the methods of the following:

- C
- An object created by F
- An object passed as argument to F
- An object held in an instance variable of C

The method should not invoke methods on objects that are returned by any of the allowed functions ⇒ talk to friends, not strangers

Bad examples

```
car.performance.parts.engine.start();

var path = ctx.getOptions().getScratchDir().getAbsolutePath();
```

Call chains (based on the first bad example with the car engine start)

```
public class Car{
  public Performance performance;
  public void StartEngine() {
    performance.StartEngine();
  }
}

public class Performance {
  public Parts parts;
  public void StartEngine() {
    parts.StartEngine();
  }
}

public class Parts{
  public Engine engine;
  public void StartEngine() {
    engine.start();
  }
}
```

Compromise to avoid long call chains:

```
public class Car{
  public Performance performance;
  public void StartEngine() {
    performance.parts.engine.start();
  }
}
```

This does not follow the Law of Dementer. But its a compromise to the call chains. It keeps the code fragile, but only in one spot in the application. Because whenever someone wants to start the car's engine, he can use the method in the Car class. If the Car structure changes, you will only need to edit it in one place.

Single Responsibility Principle

Single Responsibility Principle (SRP) ⇒ Unix Philosophy

> 💡 Do one thing and do it well

Every class should have one responsibility and therefore only one reason for changes

Changes or extensions should always include the least amount of classes possible ⇒ high cohesion remains

Rule of thumb: Prefer many small classes instead of few big ones ⇒ better testability. As well view Prefer many small entities over a few big ones

Open-Closed Principle

Open-Closed Principle (OC)

> 💡 Objects or entities should be open for extension but closed for modification

When taking a look at that statement one could be led to believe that inheritance is the silver bullet for being conform to the OC-principle. However this thought is rather dated: inheritance leads to tight coupling of classes. It can be sensible - but only in certain cases.

The revised version of this principle is the Polymorphic Open-Closed principle. The main difference is, that instead of superclasses interfaces are leveraged to abstract the implementation details. This leads to multiple potential benefits.

- Usage of composition (see Favor Composition over Inheritance)
- Loose coupling of code
- Base for Dependency Inversion
- Allows for inheritance if sensible (interface can inherit from each other)

A good example for a practical use-case of the polymorphic OC principle is the Strategy Pattern.

Every implementation is hidden behind the strategy interface and therefore is protected from changes. But if a new use-case arises it can be simply extended by a new implementation of the interface.

Liskov Substitution Principle

Liskov Substitution Principle (LSP)

> Let q(x) be a property provable about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T

This scientific definition is quite abstract. In simple terms it means that an object of a superclass must be exchangeable with a derived object without breaking the application. Therefore objects of a derived class must behave in the same way as objects from the superclass. Based on this LSP defines the following rules:

- Overridden method of a subclass must accept the same input parameters as the equivalent method of the superclass
- The implementation in the subclass must be equal or less restrictive than the one in the superclass
- The return-value of an overridden method of a subclass must comply with the same rules which apply for the return-value of the equivalent method of the superclass
    - One can’t apply stricter rules for the return value
    - Except:
        - The return value is a subtype of the return type in the superclass
        - The return value is a subset of valid return-values in the superclass

For example lets take a simple implementation of a Rectangle:

```
class Rectangle {
    protected int width;
    protected int height;

    void increaseHeight(int height) {
        this.height += height;
    }

    void increaseWidth(int width) {
        this.width += width;
    }
}
```

One could led to believe (from our basic understanding of geometry) that its only logical that the Square implementation must derive from Rectangle hence its a specific version of it. Such an implementation would look like this:

```
class Square extends Rectangle {
    @Override
    void increaseHeight(int height) {
        super.increaseHeight(height);
        this.width += height;
    }

    @Override
    void increaseWidth(int width) {
        super.increaseWidth(width);
        this.height += width;
    }
}
```

One must override both increaseHeight and increaseWidth methods in order to stay compliant with the rules of the square shape. But this pitfall leads to a clear violation of the LSP. With the implementation of Square one introduced a side effect to both methods, that always both height and width are updated accordingly. If one attempts to switch out a Rectangle-Instance with a Square-Instance the application will crash or at least will cause unpredictable side effects and errors.

Interface Segregation Principle

Interface Segregation Principle (ISP)

> 💡 A client should never be forced to implement an interface that isn't used or clients shouldn't be forced to depend on methods they do not use

In evolving applications it's often the case that codebases grow without the focus on quality therefore interfaces can become cluttered. For example lets take this Shape-Interface:

```
interface Shape {
    public int area();
    public int volume();
}

 class Square implements Shape {
    @Override
    public int area() {
        return width * height;
    }

    @Override
    public int volume() {
        throw new UnsupportedOperationException()
    }
}

class Cube implements Shape {
    @Override
    public int area() {
        // calculate area...
    }

    @Override
    public int volume() {
        // calculate volume...
    }
}
```

In the example above the Shape-Interface is too general. Although a square is a shape it isn't three-dimensional so it doesn't have a volume. The design of the Shape-Interface clearly violets the ISP. A better way of doing things would be when the shape interface is segregated into multiple more specific interfaces.

```
interface Shape {
    int area();
}

interface ThreeDimensionalShape {
    int volume();
}

class Square implements Shape {
    @Override
    public int area() {
        return width * height;
    }
}

class Cube implements Shape, ThreeDimensionalShape {
    @Override
    public int area() {
        // calculate area...
    }

    @Override
    public int volume() {
        // calculate volume...
    }
}
```

Now every consumer only has to implement the functionalities he can and has, which makes the code more concise and more predictable. For the Cube and other 3D shapes one can composite the according interface thus achieving the same effect as in the first example.