Codurance.com

Coupling & Cohesion

- What is Coupling?
- Types of Coupling
- Cohesion
- Types of Cohesion

What is coupling?

Degree of interdependence between software modules

Types of Coupling

- ➔ Interaction Coupling: When the dependency is based on a component calling methods of another
- ➔ Inheritance Coupling: When the dependency is based on a component inheriting from another

codurance

Interaction Coupling

- **Message Coupling (Best):** Components communicate by using messages
- **Data Coupling**: Components share data through parameters in order to perform a method call on another component. Compared to message passing, a component needs to know a bit more detail about the other component
- **Stamp Coupling:** Components share a common data structure. We should avoid passing data structures amongst components. (*Remember first class collections rule from Object Calisthenics*)
- **Control Coupling:** One component is responsible for controlling the flow of another component. Refactoring to *design patterns* such as state or strategy can help fix this type of Coupling
- **External Coupling:** Components share an external data format or protocol. Abstracting the knowledge of external data allows us to remove this type of Coupling
- **Common Coupling:** Components share global state; any change in this global data format has a ripple effect, breaking multiple components. We can abstract common data to minimize this Coupling or remove global data and move to Data Coupling or message-passing,
- **Content Coupling (Worst):** Components depend on other components' internal elements' data. This is the worst since one component can mutate another components data. Using encapsulation and abstractions is an effective strategy to minimize this type of Coupling

Law of Demeter

Law of Demeter/ AKA Message Chains Code smell. Messages should be sent only by an object (methods should only be invoked on objects):

- to itself
- to objects contained in attributes of itself or a superclass
- to an object that is passed as a parameter to the method
- to an object that is created by the method
- to an object that is stored in a global variable
- Or - You can only play with toys
    - you own
    - you build
    - you are given

Method Coupling Premises

Another important aspect of Interaction Coupling is the number and kind of parameters passed into the methods, which we can think of as the **impact** of that method on the overall Coupling.

Method’s Parameters	Relative coupling impact
void method()	0
void method(int x)	1
int method()	1
void method(ClassA a)	1 + FanIN
ClassA method()	1+ FanOUT
void method(int x, int y)	2
int method(int x)	2
void method(ClassA a, ClassB b)	2 + 2FanIN
ClassA method(ClassB b)	2 + FanIn + FanOut

Inheritance Coupling

Inheritance Coupling is the strongest type of Coupling and happens every time a class inherits from another one. When that happens, the subclass binds itself completely to the superclass. If the definition is modified, all the subclasses are affected

- Ideally, we should **“use inheritance only for generalization/specialization semantics”** (A-Kind-Of relationship)
- For any other case, prefer **“Composition over inheritance”**

Cohesion

"Cohesion of an element A is defined as 'if I change sub-element A.1, do I have to change all the other sub-elements A.2-n?' So, cohesion is coupling within an element". **Kent Beck**

Cohesion is one of the most important concepts in software design. Cohesion is at the core of the vast majority of good design principles and patterns out there, guiding separation of concerns and maintainability.

We have 2 main distinct categories of Cohesion:

- **Class Cohesion:** As the holistic Cohesion at class level deducted from fields, methods and their interactions
- **Method Cohesion:** As the Cohesion of the individual method focusing on how ‘single-minded’ they are.

Class Cohesion

- **Ideal:** A class has an ideal Cohesion when it doesn’t show any other types of mixed Cohesion
- **Mixed-Role:** The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g., the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class.
- **Mixed-Domain:** The class has one or more attributes that relate objects of the class to other objects on a different layer. As such, they have nothing to do with the underlying semantics of the thing that the class represents. For example, a port attribute located in a problem domain class should be in a system architecture class that is related to the problem domain class.
- **Mixed-Instance:** The class represents two different types of objects and should be split in two separate modules. Typically, different instances use only subsets of fields and methods of the class.

Cheatsheet

Mixed-Role

Same domain lang. not related

```
Pizza_toppings.class
  ├── 🧀
  ├── 🍅
  └── 🌾 (wheat)
```

Mixed-Domain

Same exec flow, different layer

```
Police_report.class
  ├── 🛂 passport status
  ├── 🛃 luggage status
  └── 🌐 "/reports-v2/{id}"
```

Mixed-Instance

Different responsibility (2 classes in one)

```
Serving_modality.class
  ├── 🧀
  ├── 🛵 (courier number)
  ├── 🪑 (restaurant table number)
  └── 🍅
```

codurance

Cohesion (Method Cohesion)

- **Coincidental (worst)**: Processing elements are grouped arbitrarily and have no significant relationship. There is no relationship between the processing elements. E.g.: update a customer record, calculate a loan payment, print a report. Coincidental cohesion is quite common in modules called “Utils” or “Helpers”
- **Logical**: At a module level, processing elements are grouped because they belong to the same logical class of related functions. At each invocation of the module one of the processing elements is invoked. E.g.: grouping all I/O operations, all database operations, etc. At a processing element level, the calling module passes a control flag and that flag decides which piece of behaviour will be invoked by the processing element. E.g: A flag indicating if a discount should be calculated, a piece of behaviour should be skipped, etc.

Cohesion (Method Cohesion)

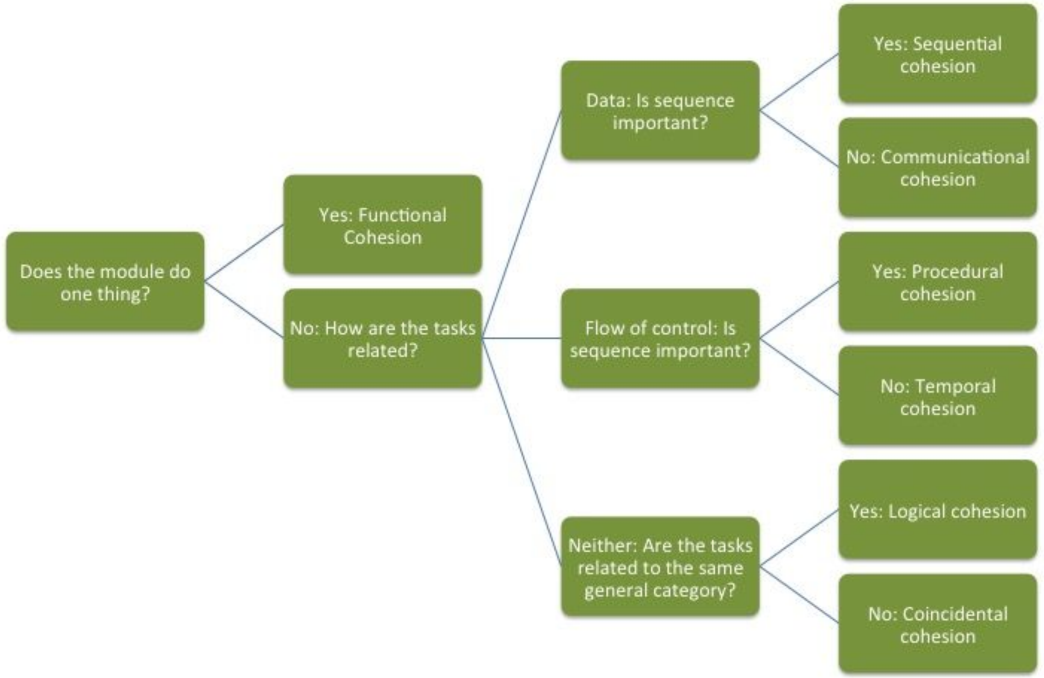
- **Temporal:** Processing elements are related in time. They are grouped together because they are invoked together at a particular time in a program execution but in fact they are unrelated to each other. A different business requirement may require a different sequence or combination of processing elements. E.g.: data persistence / validation, audit trail, notifications via email, etc.
- **Procedural**: Processing elements are sequentially part of the same business unit but do not share data. They are grouped because they always follow a certain sequence of execution. E.g.: validate user, process a payment, trigger stock inventory system to send purchase orders to suppliers, write logs.

Cohesion (Method Cohesion)

- **Communicational**: Processing elements contribute to activities that use the same inputs or outputs.
- **Sequential:** Processing elements are grouped when the output of one processing element can be used as input for another processing element. E.g.: formatting and validating data.
- **Functional (Best):** All processing elements of a module are essential to the performance of a single and well-defined task.

E.g.: processing elements that would take a shopping basket and calculate discounts, promotions, money saved, delivery costs, and return the total price.

Cohesion


Does the module do one thing?
Yes: Functional Cohesion
No: How are the tasks related?
Data: Is sequence important?
Flow of control: Is sequence important?
Neither: Are the tasks related to the same general category?
Yes: Sequential cohesion
No: Communicational cohesion
Yes: Procedural cohesion
No: Temporal cohesion
Yes: Logical cohesion
No: Coincidental cohesion

SOLID principles

SOLID is an acronym for the first **five object-oriented design principles** by **Robert C. Martin**, also known as **Uncle Bob**

SOLID stands for:

- **S:** Single-responsibility principle
- **O:** Open-closed principle
- **L:** Liskov substitution principle
- **I:** Interface segregation principle
- **D:** Dependency Inversion Principle

Single Responsibility principle

A class should have one and only one reason to change, meaning that a class should have only one job (does anyone read Cohesion here?)

```
class Book {
   getTitle() {
       return "A Great Book";
   }


   getAuthor() {
       return "John Doe";
   }


   turnPage() {
       // pointer to next page
   }


   printCurrentPage() {
       return "current page content";
   }
}
```

codurance

Single Responsibility principle

```
class Book {
   getTitle() {
       return "A Great Book";
   }

   getAuthor() {
       return "John Doe";
   }
}

class Pager {

   gotoPrevPage() {
       // pointer to prev page
   }

   gotoNextPage() {
       // pointer to next page
   }

   gotoPageByPageNumber(pagerNumber: number)
{
       // pointer to specific page
   }
}
```

```
class Printer {
   printPageInHTML(pageContent: any) {
       // your logic
   }

   printPageInJSON(pageContent: any) {
       // your logic
   }

   printPageInXML(pageContent: any) {
       // your logic
   }

   printPageUnformatted(pageContent: any) {
       // your logic
   }
}
```

codurance

Open Closed principle

Objects or entities should be open for extension, but closed for modification

```
class Book {
   getAuthor() {
       return {
           name: 'Javier Chacana',
           age: 34,
           address: 'Some place in Chile'
       };
   }
}
```

Does it look right? What happens to existing clients using this API?

Open Closed principle

```
class Book {
   …
   getAuthor() { return 'John Doe'; }
}

class Book1 extends Book {

   constructor() {
       super();
   }

   getAuthor() {

       return {
           name: super.getAuthor(),
           age: ''
       }
   }
}
```

```
class Book2 extends Book {

   getAuthor() {

       return {
           name: super.getAuthor(),
           age: '',
           address: ''
       }
   }
}
```

This looks promising. It keeps the old contract while extending its behavior for further requirements.

In short, inheritance fulfills the second principle

Liskov Substitution principle

Derived types must be completely substitutable for their base types

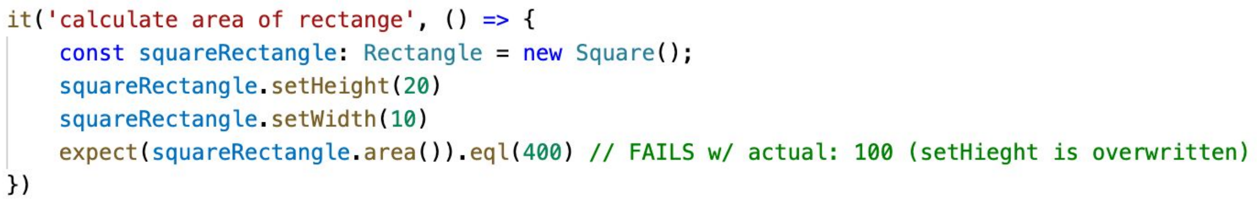
Can we call a Square “a kind of Rectangle” ?

YES !

Liskov Substitution principle

Derived types must be completely substitutable for their base types

Can we call a Square “a kind of Rectangle” ?



```
it('calculate area of rectange', () => {
    const squareRectangle: Rectangle = new Square();
    squareRectangle.setHeight(20)
    squareRectangle.setWidth(10)
    expect(squareRectangle.area()).eql(400) // FAILS w/ actual: 100 (setHieght is overwritten)
})
```

Liskov Substitution principle

```
class Bird {
   fly() {
       console.log('I can fly!');
   }
}

class Kingfisher extends Bird {
   constructor() {
       super()
   }
}

class Ostrich extends Bird {
   constructor() {
       super()
   }
   fly() {
      throw new Error("I don't fly rather I run");
   }
   run() {}
}

let kingfisherBird: Bird = new Kingfisher();
let ostrichBird: Bird = new Ostrich();
kingfisherBird.fly(); // kingfisher can fly.
ostrichBird.fly()// I don't fly rather I run
```

The new output states an ostrich does not fly, which is correct in terms of behavior.
Can you come up with a better solution?

Liskov Substitution principle

Derived types must be completely substitutable for their base types

```
class Bird {
   fly() {
       console.log('I can fly!');
   }
}

class Kingfisher extends Bird {
   constructor() {
       super()
   }
}

class Ostrich extends Bird {
   constructor() {
       super()
   }
}

let kingfisherBird: Bird = new Kingfisher();
let ostrichBird: Bird = new Ostrich();
kingfisherBird.fly(); // kingfisher can fly.
ostrichBird.fly()// ostrich can fly
```

Does it look right? Do Ostriches really fly?

What Liskov Substitution principle states - Each derived class must replace its parent without affecting parent's behavior.

Interface Segregation principle

A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they don't use

```
interface Bird {
   fly();
   run();
}


class Kingfisher implements Bird {
   fly() { }
   run() { }
}


class Ostrich implements Bird {
   fly() { }
   run() { }
}
```

Why does a Kingfisher need to implement run? And why does an Ostrich need to implement fly?

Clients shouldn't be forced to depend on methods they do not use.

Interface Segregation principle

```
interface FlyingBird {
   fly();
}

interface RunningBird {
   run();
}

class Kingfisher implements FlyingBird {
   fly() { }
}

class Ostrich implements RunningBird {
   run() { }
}
```

Now we have segregated interfaces both for Flying and Running birds.

Can you think of a better or alternative way?

Dependency Inversion principle

High level modules should not depend upon low level modules. Rather, both should depend upon abstractions.

```
class SocialLogin {
   login(googleLogin: any) {
     // some code which will be used for google login.
   }
}
```

In this example, we have a Login class that has an implementation for a Social Login with Google.

What will happen if we get a change in the requirements and now we need to support Facebook Social login?

Dependency Inversion principle

```
class Login {
   login(socialLogin: SocialLogin){
      socialLogin.doSomething()
   }
}

interface SociaLogin {
   doSomething();
}

class GoogleLogin implements SocialLogin {
   doSomething() {
   // some code which will be used for google login.
   }
}

class FBLogin implements SocialLogin {
   doSomething() {
   // some code which will be used for fb login.
   }
}
```

One alternative could be to make SocialLogin a dependency of the login method, then implement different versions for Google and Facebook social logins.

Can you propose another alternative?

Dependency Inversion principle plays an important role on a future topic: Test Doubles

BYOC: Bring your own code

Let’s analyze some of your daily code, a library you use or any code you’d like to analyze (including previous exercises) and spot the different degrees of coupling and cohesion the code has