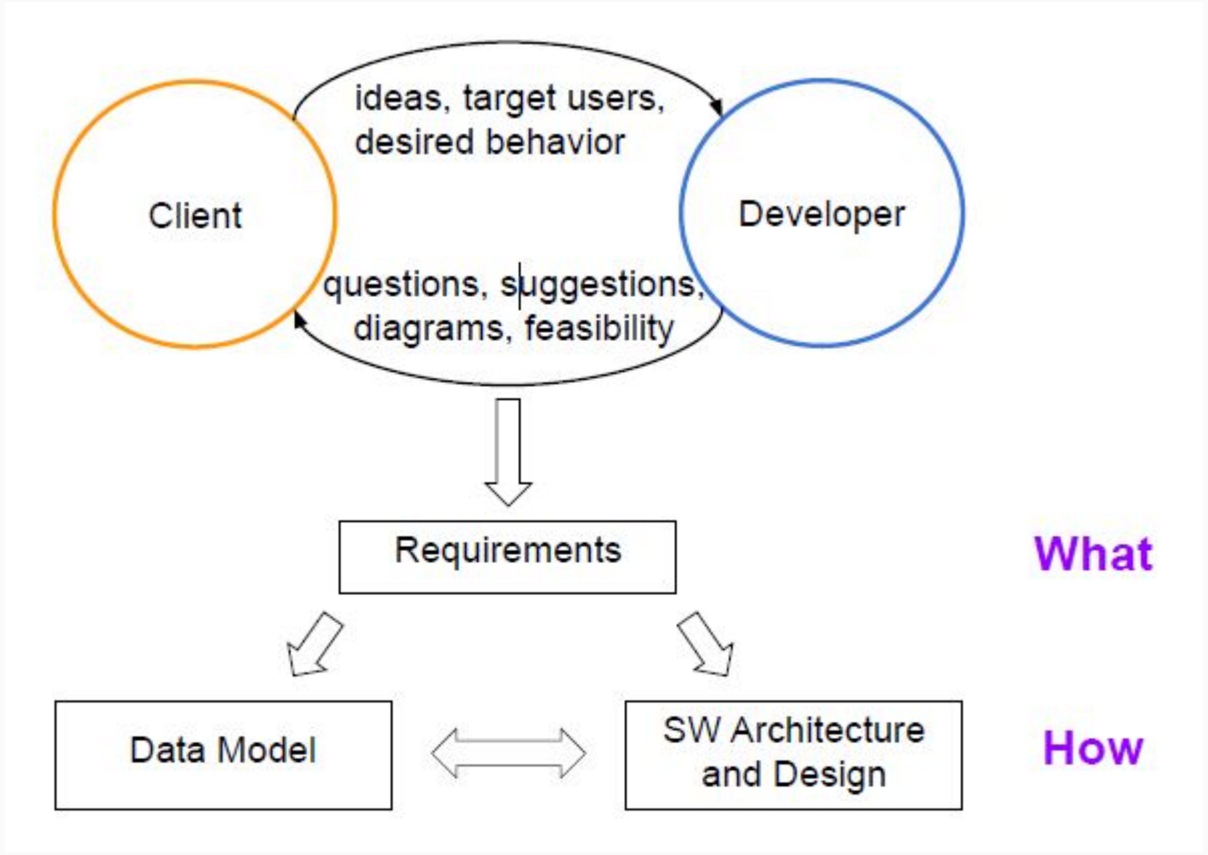
Lesson 6-:Software Design

CS 438
Ali Aburas PhD

Today’s Goals

- Define design
- What is UML?

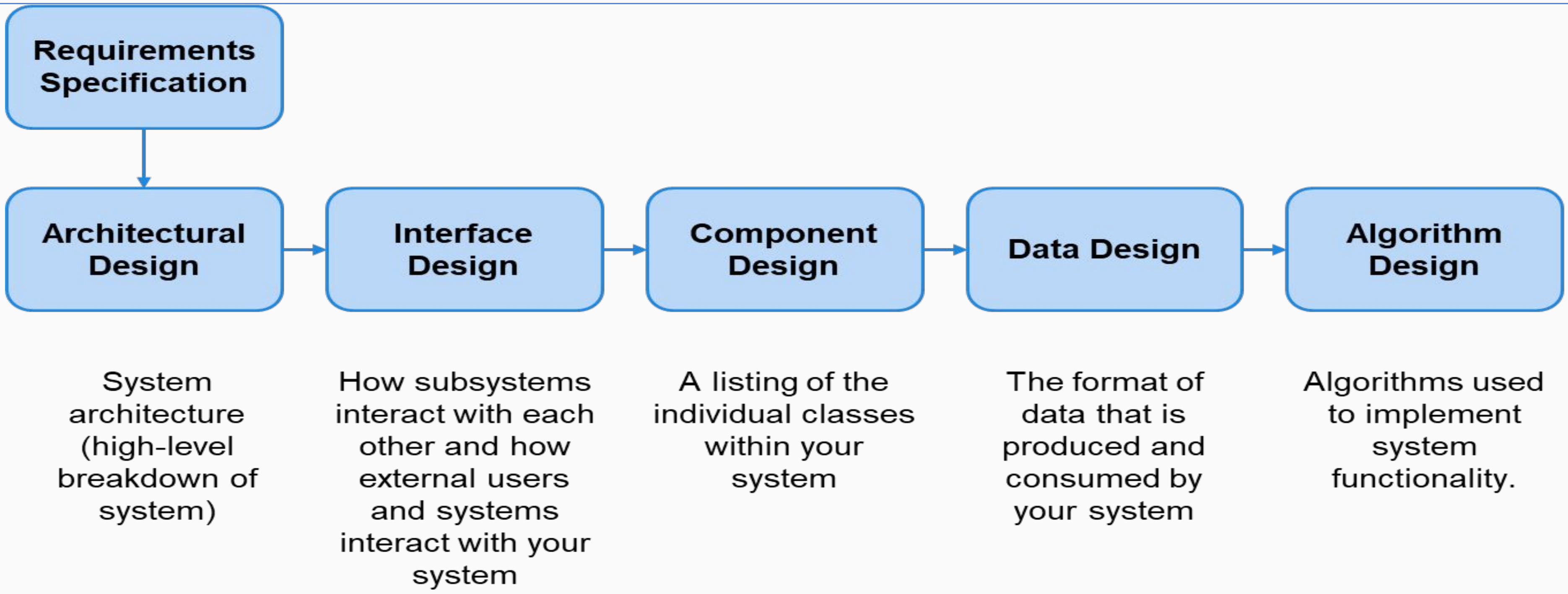
From Requirements to System Design


ideas, target users, desired behavior
Client
Developer
questions, suggestions, diagrams, feasibility
Requirements
What
Data Model
SW Architecture and Design
How

What is Design?

- Design is the creative process of transforming a problem into a solution.
- In our case, transforming a requirements specification into a detailed description of the software to be implemented.
- **Requirements** - ***what*** we're going to build.
- **Design** - ***how*** to build it.
    - A description of the structure of the solution
- Design activities are an essential part of the software design and development process. They help to create a blueprint for the software, ensure that it meets the requirements of its users, and provide a roadmap for the development team.

Design Activities


Requirements Specification
Architectural Design
Interface Design
Component Design
Data Design
Algorithm Design
System architecture (high-level breakdown of system)
How subsystems interact with each other and how external users and systems interact with your system
A listing of the individual classes within your system
The format of data that is produced and consumed by your system
Algorithms used to implement system functionality.

- The **output** of these processes is a **set of models** and **artifacts** that record the major decisions that have been taken, along with an **explanation** of the rationale for each **nontrivial decision**.

A Complete Guide to The Software Design Process

1. **Modularity**: modularity is one of the major keys to maintainable, technical & robust Software design. It helps to divide a big project into smaller modules.
2. **Coupling**: Another major software design principle is low coupling. It mainly refers to the interconnection between the software design modules & how closely they are connected.
3. **Anticipation of change**: The demand for software development is always increasing. This means constant changes are required for the overall design requirements. The ability to accept new changes and adjust them pleasantly is crucial in developing an appropriate software design.
4. **Simplicity**: One of the major goals of a proper software design is to make things simple. Every task in software design process has a separate module that can be independently used as well as modified. The simplicity of the software makes codes easier & user-friendly. It also reduces the risk of introducing new errors.

What makes a good design?

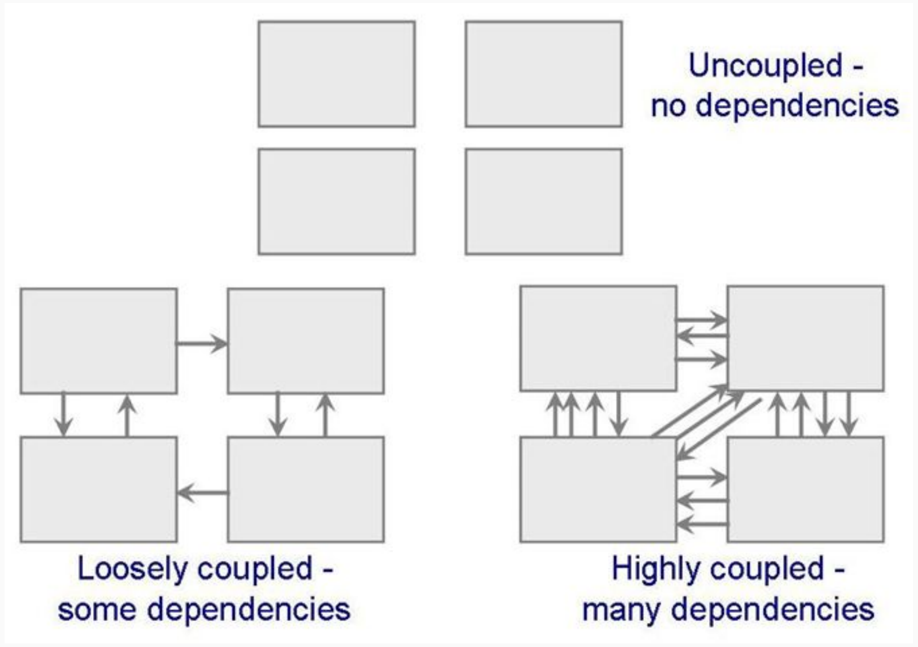
Modularity & Decomposition

- **Decomposing** and **modularizing** means that **large software** is **divided** into a number of smaller named **components** having **well-defined interfaces** that describe component interactions.
- Usually the goal is to place **different functionalities** and **responsibilities** in **different components**.

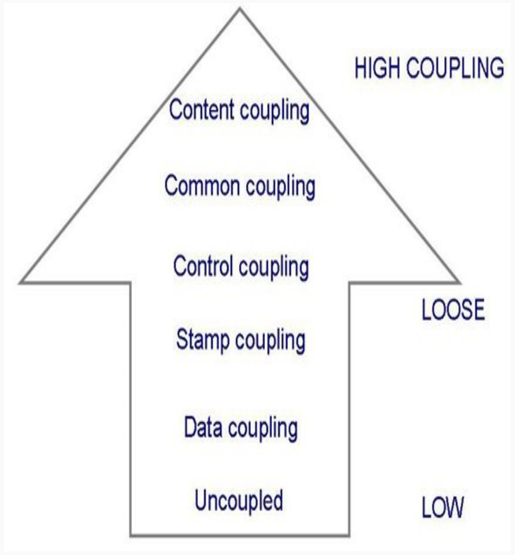
What makes a good split?

Low coupling

- **Coupling** is a measure of the interdependence among components in a computer program.
    - Two modules that are tightly coupled are strongly dependent on each other.
- Modules should be as independent as possible from other modules, so that changes to module don't heavily impact other modules.

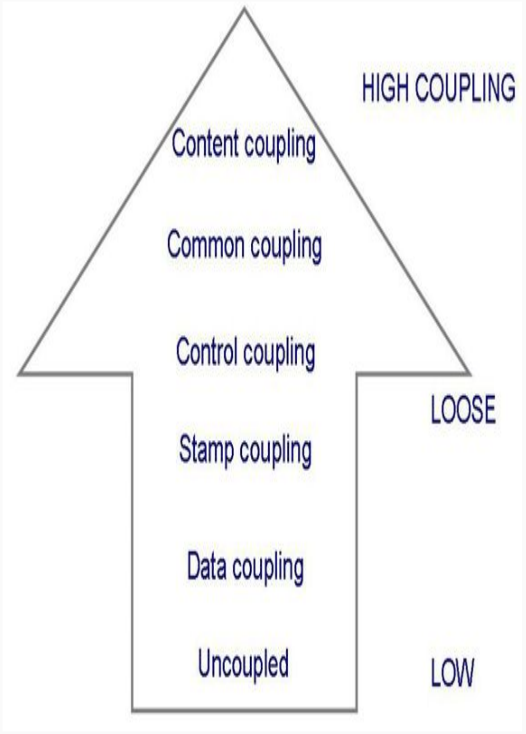


Uncoupled - no dependencies
Loosely coupled - some dependencies
Highly coupled - many dependencies



Low coupling

- **Data Coupling:** If the dependency between the modules is based on the fact that they communicate by passing only data, then the modules are said to be data coupled.
- **Stamp Coupling** In stamp coupling, the complete data structure is passed from one module to another module, and the receiving module that may use only a few specific parts (**some data members of the data structure**) from that data structure.. This data structure could be arrays, structures, unions, and so on.
- **Control Coupling** simply means to control data sharing (**By passing control data concerning which execution flow of control will be determine within other module**) between modules.
- **Common Coupling:** The modules have shared data such as global data structures. The changes in global data mean tracing back to all modules which access that data to evaluate the effect of the change.
- **Content Coupling:** In a content coupling, one module can modify the data of another module, or control flow is passed from one module to the other module.



High cohesion

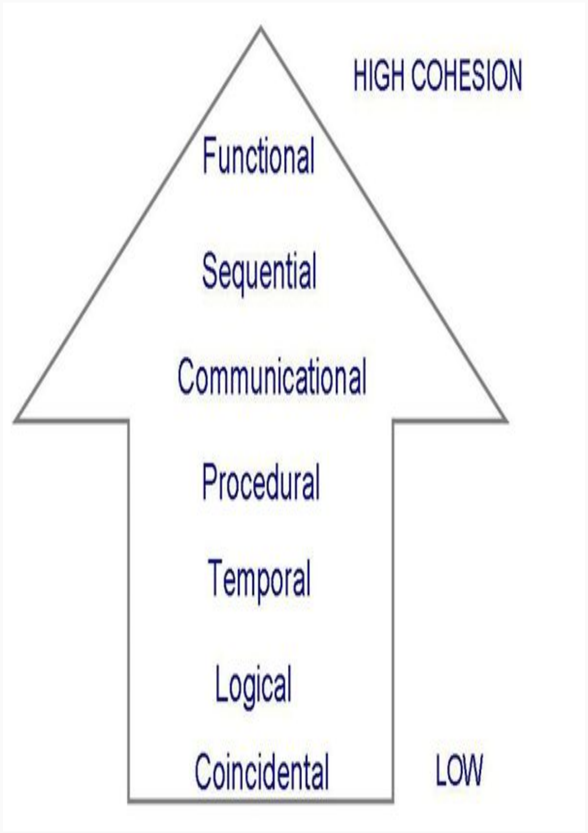
- **Cohesion** is a measure of the strength of association of the elements within a component.
- Cohesion refers to the degree to which elements within a module work together to fulfill a single, well-defined purpose.
  - Single responsibility (focus on on doing one thing well – high cohesion)
- High cohesion means that elements are closely related and focused on a single purpose, while low cohesion means that elements are loosely related and serve multiple purposes.

- We want **high** cohesion and **low** coupling. Why?

High cohesion

A good software design will have high cohesion.

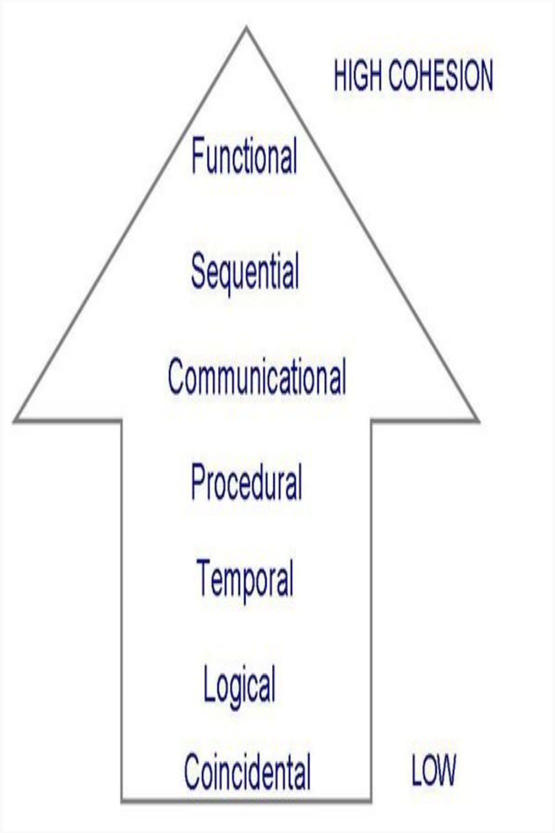
- **Logical Cohesion:** The elements are logically related and not functionally. Example, a Transaction module, which groups components that perform different kinds of business transactions such as creating and managing users, customers, products, orders, payments, etc.
- **Temporal Cohesion:** Temporal cohesion occurs when elements or tasks are grouped together in a module based on their timing or frequency of execution, such as a module that handles starting up, initializing, or shutting down.
- **Procedural cohesion** involves grouping methods of a class based on their sequence of execution. Forr example, a **RemoteMessageSender** class with a **ping method** and **send_message method**. The RemoteMessageSender is procedurally cohesive because before a message is sent, the ping method is invoked to ensure the host is alive.
- **Communicational cohesion** occurs when elements or tasks are grouped together in a module based on their interactions. For example, a data structure that represents the contents of a customer's shopping basket might be used by a variety of elements in a single module. The elements might calculate discounts, shipping, and taxes based on the same data structure.

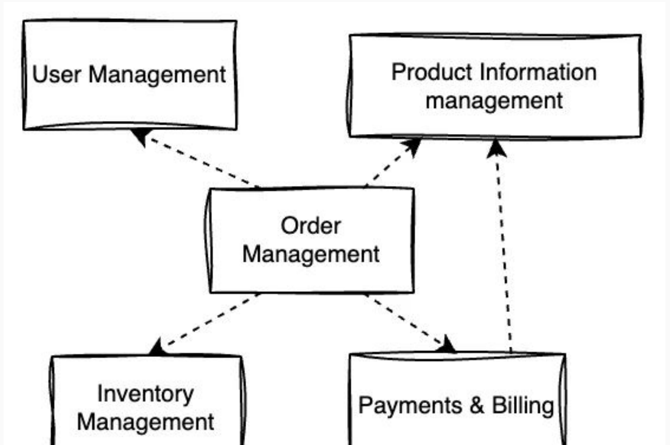


High cohesion

- **Sequential cohesion** occurs when the elements within a module are arranged in a specific sequence, with the output of one element serves as the input for the next element.
    - For example, a module that calculates a student's GPA score based on their marks.
        - Add individual mark of each subject
        - Calculate GPA score
        - Calculate cumulative GPA score

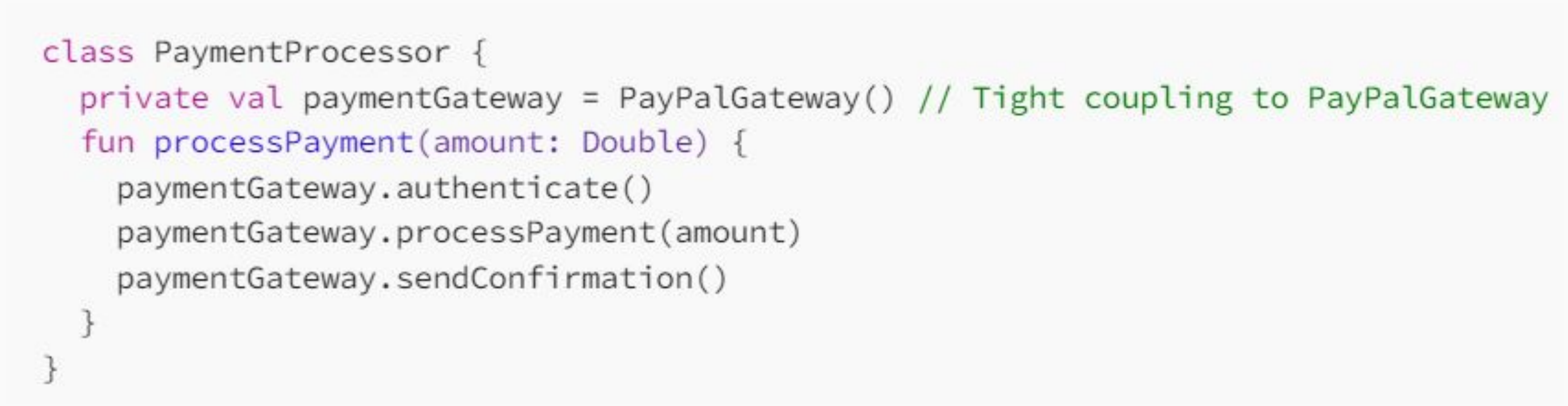
- **Functional Cohesion:** all the elements of the module are grouped together because they work together towards fulfilling a single, common business function.
    - In the example shown in the figure, each module is responsible for a single business function.





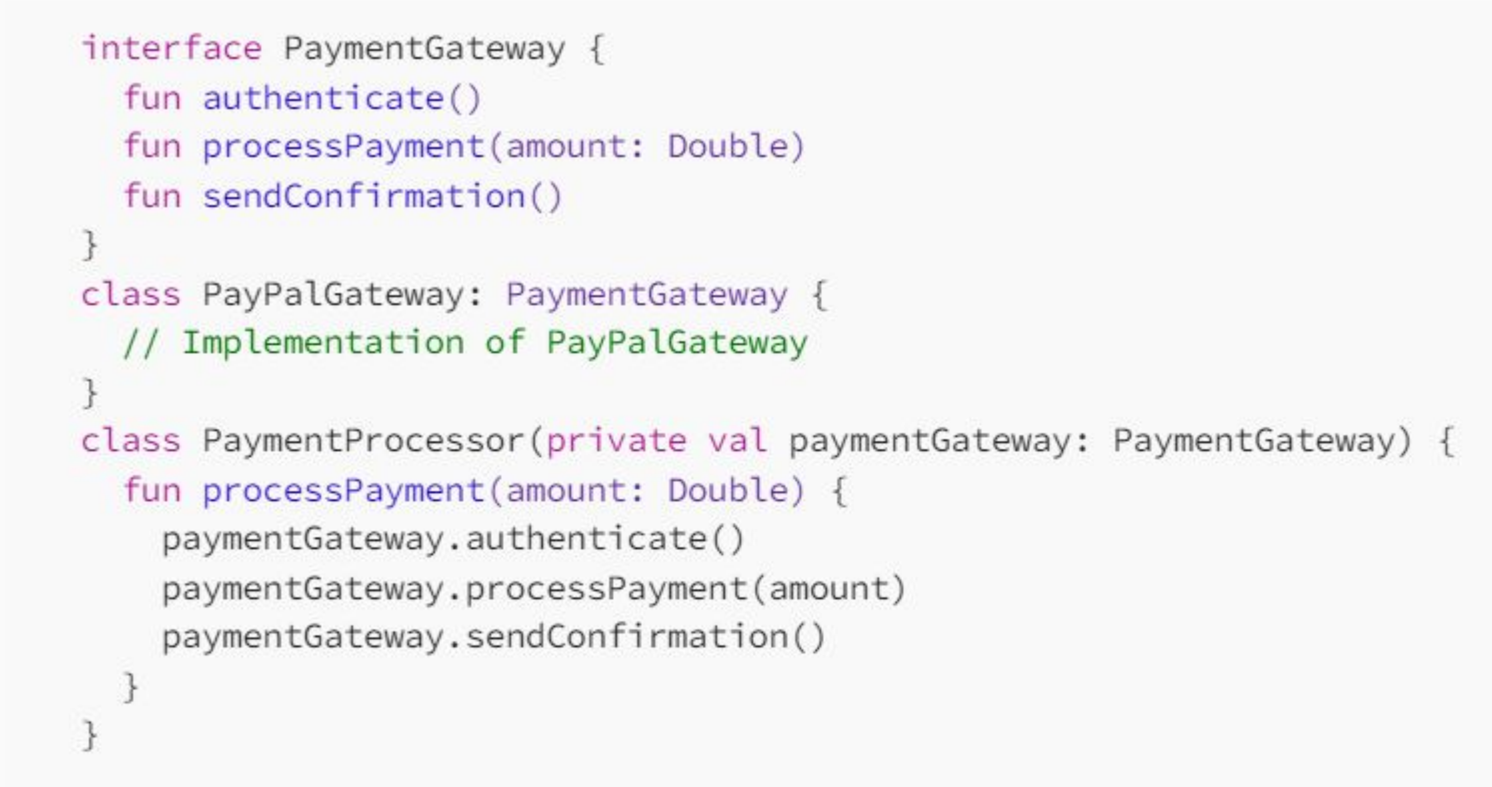
Example of Tight Coupling

Consider an example where a `PaymentProcessor` class depends on a specific `PaymentGateway` implementation:



```
class PaymentProcessor {
  private val paymentGateway = PayPalGateway() // Tight coupling to PayPalGateway
  fun processPayment(amount: Double) {
    paymentGateway.authenticate()
    paymentGateway.processPayment(amount)
    paymentGateway.sendConfirmation()
  }
}
```

Example of Tight Coupling



```kotlin
interface PaymentGateway {
  fun authenticate()
  fun processPayment(amount: Double)
  fun sendConfirmation()
}
class PayPalGateway: PaymentGateway {
  // Implementation of PayPalGateway
}
class PaymentProcessor(private val paymentGateway: PaymentGateway) {
  fun processPayment(amount: Double) {
    paymentGateway.authenticate()
    paymentGateway.processPayment(amount)
    paymentGateway.sendConfirmation()
  }
}
```

In this refactored code, the `PaymentProcessor` class now depends on the `PaymentGateway` interface rather than a specific implementation.

OO design principles

Understanding Object-Oriented Design Principles

- What is Object-Oriented Design?
  - Object-Oriented Design (OOD) is a method of designing software by conceptualizing it as a group of interacting objects, each representing an instance of a class.
  - These objects encapsulate both data (attributes) and behavior (methods).
  - OOD aims to create a system that is modular, reusable, and easy to maintain.
- Why is Object-Oriented Design Important?
  - **Modularity**: Breaks down complex problems into manageable pieces.
  - **Reusability**: Promotes reuse of existing components.
  - **Scalability**: Makes it easier to extend and scale software.
  - **Maintainability**: Enhances code readability and maintainability.
- Core Object-Oriented Design Principles
  - There are several principles that form the foundation of OOD.
  - These principles help in creating software that is flexible and easy to modify.
  - There are many Object-Oriented Design Principles, but here are some important ones.

SOLID principles

The SOLID ideas are

- The **S**ingle-responsibility principle: "There should never be more than one reason for a class to change." In other words, every class should have only one responsibility.
- The **O**pen–closed principle: "Software entities ... should be open for extension, but closed for modification.”
- The **L**iskov substitution principle: "Functions that use pointers or references to base classes must be able to use objects of derived classes without knowing it."
- The **I**nterface segregation principle: "Clients should not be forced to depend upon interfaces that they do not use."
- The **D**ependency inversion principle: "Depend upon abstractions, [not] concretes”

Source: Wikipedia

What is a Good Software Design?

Goal/Feature	Effective Principles/Methods
Correct Integration	Correctness, Abstraction, Architecture
Efficiency	Optimal Resource Consumption, Refinement, KISS (Keep It Simple, Stupid)
Scalability	Modularity and Scalability, Patterns, DRY (Don't Repeat Yourself)
Adaptability to Change	Maintainability, Refactoring, YAGNI (You Ain't Gonna Need It)
High-quality Design	Completeness, Data Protection, SOLID Principles
Security	Data Protection, Architecture, SOLID Principles (particularly Dependency Inversion Principle for modular security design)
Reusable and Maintainable Code	DRY (Don't Repeat Yourself), Modularity and Scalability, SOLID Principles (Single Responsibility Principle for focused modularity, Open/Closed Principle for extensibility)

Information Hiding Example

```
int[] sortAscending(int[] unsorted, int length);
```

- We do not know what sort routine is used.
- All we know is what the interface is and what the module accomplishes.
- Allows designers to focus on one part of the system at a time, without worrying about other components.

Data Encapsulation

- Protect the data from unauthorized access.
- Nobody else can mess with the data.
- Makes the design more robust.

Version 1:

```
class Adder{

     int total;
          void addNum(int number){
                    total += number;
          }
};


int main( )
{
   Adder a;

   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.total <<endl;
   return 0;
}
```

Version 2:

```
class Adder{

     private int total;
          void addNum(int number){
                    total += number;
          }

     int getTotal(){

          return total;

     }
};
int main( )
{
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);
   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

Understandability

- The design should be understandable by the developers - unambiguous and easy to follow. Related to many component characteristics:
- **Cohesion**
    - Can each component be understood on its own?
- Naming
    - Are meaningful component (class, method, variable) names used?
- Documentation
    - Is the design well-documented? Are decisions justified? Rationale noted?

Adaptability and Inheritance

- Inheritance improves adaptability.
- Components may be expanded without change by deriving a sub-class and modifying that derived class.
- However, as the depth of the inheritance hierarchy increases, so does complexity.
    - Complexity must be periodically reviewed and restructured.

Next Time

- UML
- Version control and Git