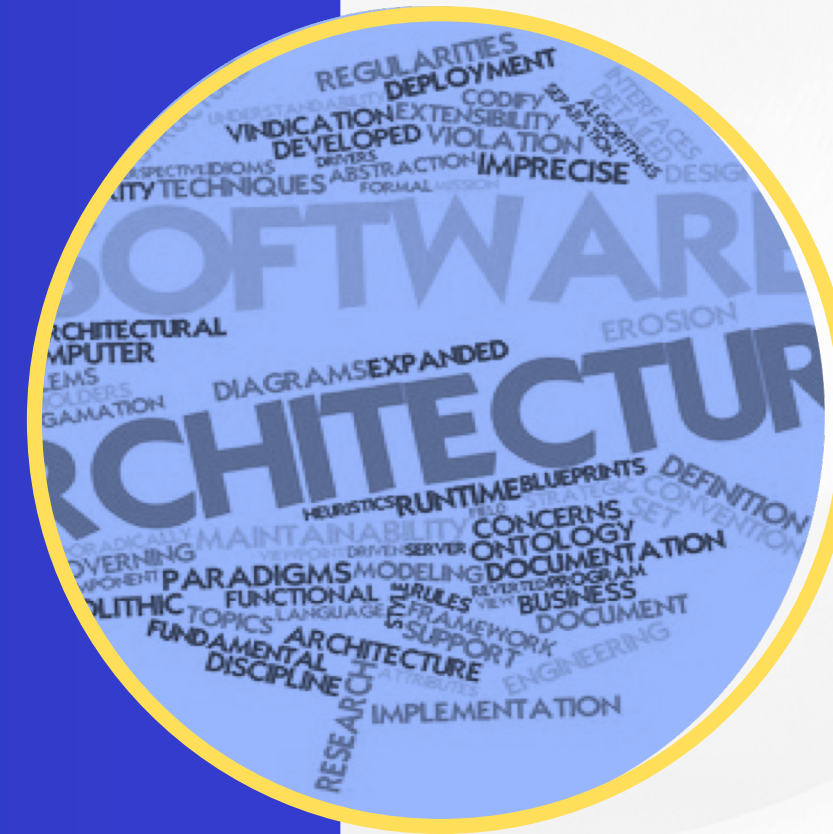


University of Tripoli – Faculty of Information Technology

Software Engineering Department

Software Architecture & Design

ITSE411



Software architecture and design

for modern large-scale systems

Lecture 3 :

Design Concepts



What We Learn In This Lecture

- Overviews
- SW Design Concepts.

Overviews

Why Software Design matters?

- Because it always takes more time to write something from scratch than changing something that is already there.
- With good design the time you spend to change (and test) something is also minimized

Why Software Design matters?

- **Maintainable software**
 - Can implement new requirements with less cost
 - Can change existing implementation with less cost
 - Because requirement keep changing
 - Unit Test Programs also need to be changed
 - They also need to have good design

Why Software Design matters?

What is a bad designed code?

- If you change something
 - you break something else that is not related to the change
- if you change something
 - you need to change something else

How Software designers solve problems?

- ✓ Dividing the problem
- ✓ Isolating parts (each part has one goal/responsibility)
- ✓ Reducing dependencies between different related elements

SW Design Concepts

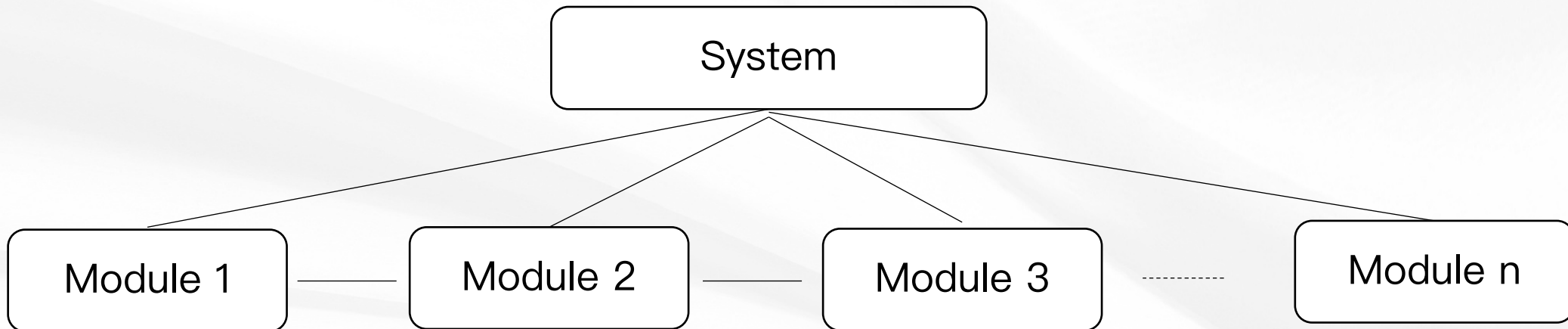
- Dividing the problem
 - **Modularization**
- Isolating parts (each part has one goal/responsibility)
 - **High Cohesion**
 - High cohesion is when you have a class that does a well-defined job. Low cohesion is when a class does a lot of jobs that don't have much in common.
 - **Cohesion** refers to the degree to which the elements of an entity belong together

SW Design Concepts

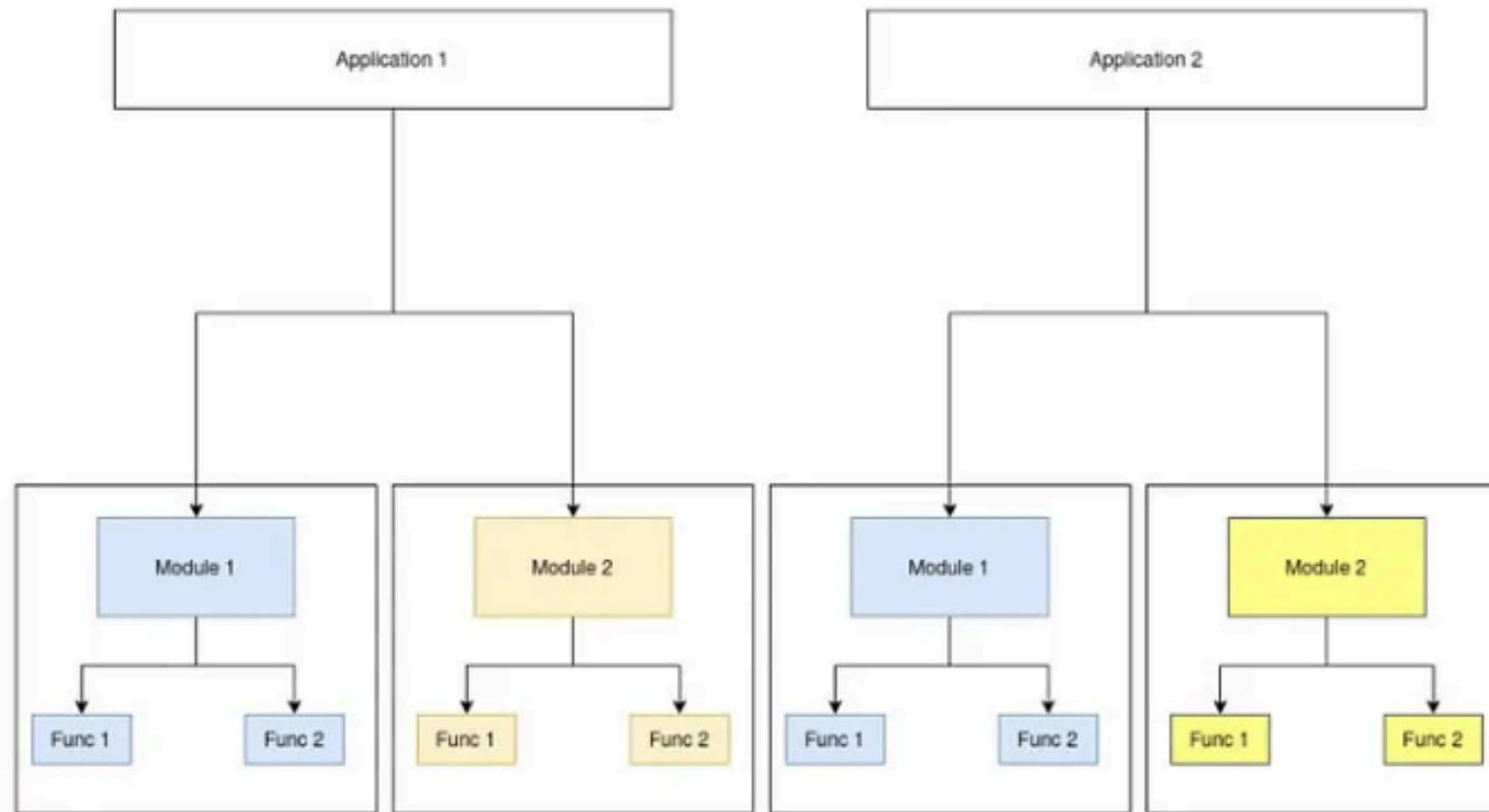
- Reducing dependencies between different related elements
 - Lower Coupling
 - The goal of a loose/low coupling architecture is to reduce the risk that a change made within one element will create unanticipated changes within other elements.
 - **Coupling** refers to degree of interdependence between software entities.

Modularization

Modularization is a technique to divide a software system into multiple independent modules, where each module works independently. Designers tend to design modules such that they can be executed and/or compiled separately and independently.

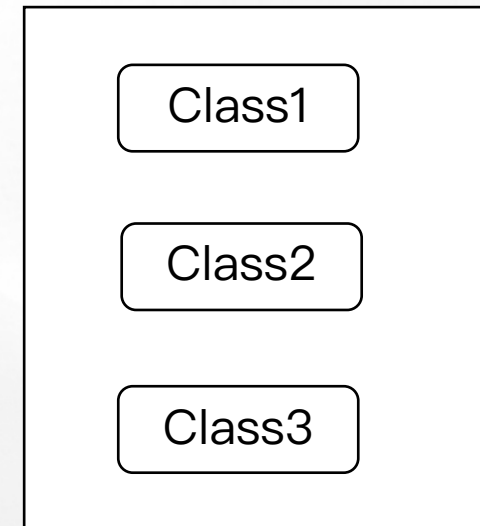
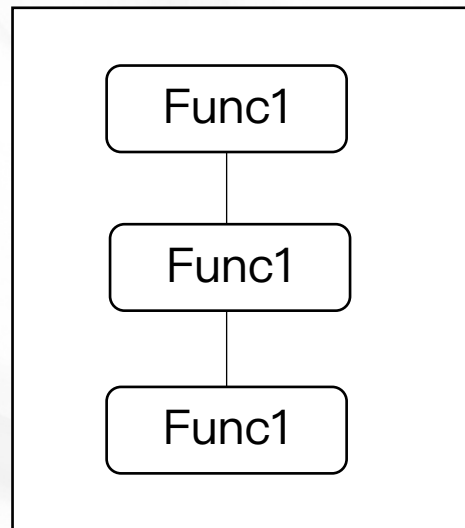
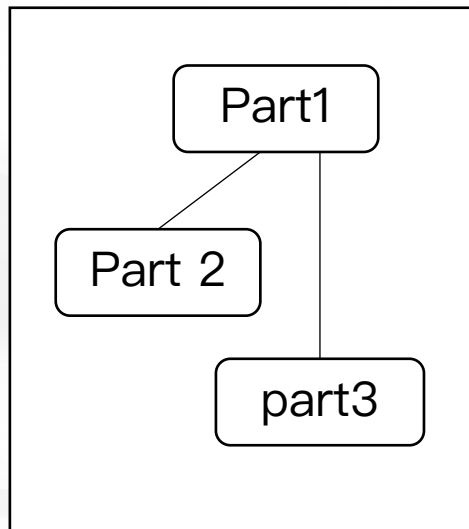


Modularization



What Is a Module?

A **module** refers to a self-contained and reusable piece of software that performs a specific set of related functions.



Module

What Is a Module?

A **module** is a program unit with parts.

- *Immediate parts* are those directly below the whole in the parts hierarchy.
- Program
 - Sub–programs or sub–systems
 - Packages, compilation units
 - Classes, functions
 - Attributes, operations, blocks
 - Lines of code

Advantages of Modularization

There are many advantages of Modularization in software engineering. Some of these are given below:

- Easier to understand each module and their purpose. So easy to understand the system,
- Smaller components are easier to maintain, so System maintenance is easy
- Easier to reuse and refactor modules.
- A module can be used many times as their requirements

SW Design Concepts

Coupling and **Cohesion** are two key concepts in software engineering that are used to measure the quality of a software system's design.



SW Design Concepts

□ **Cohesion** the extent to which a component has a single purpose or function

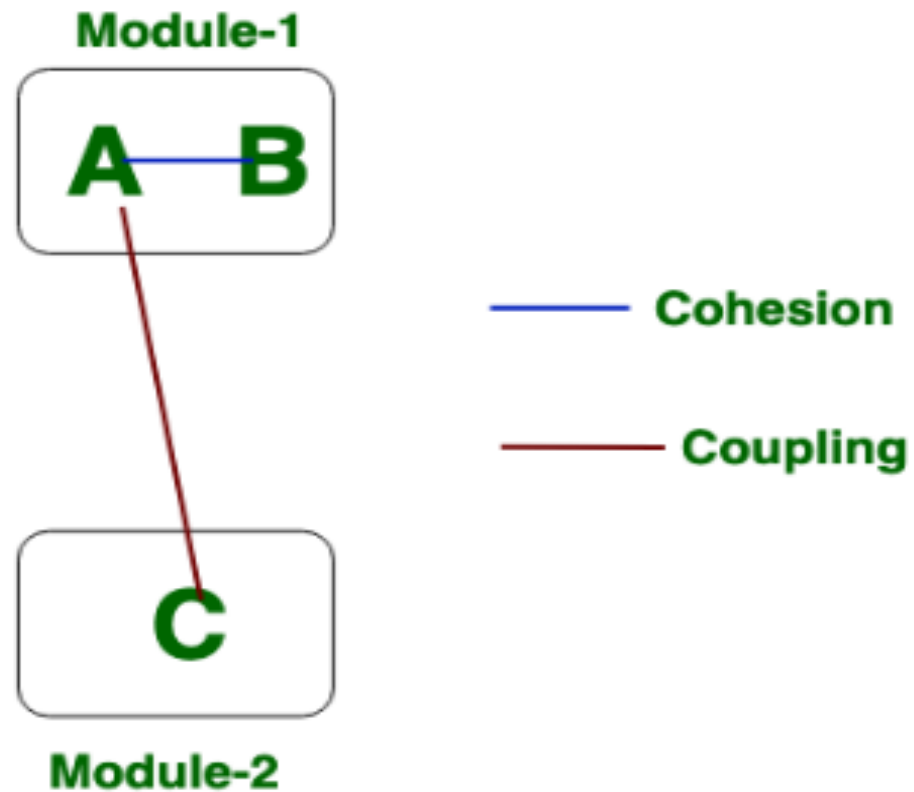
– High cohesion is good



□ **Coupling** the extent to which two components depend on each other for successful execution

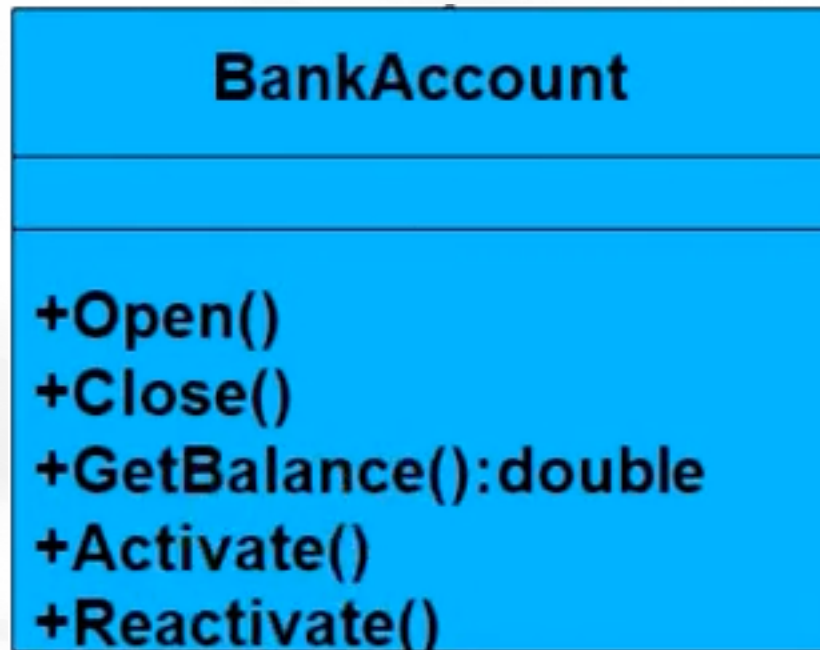
– Low coupling is good

SW Design Concepts



Cohesion

Cohesion **measures** how much a design makes the **design** and **architectural** elements to be **responsible** of **one and only**.



High Cohesion?! 

Cohesion

BankAccount
+Open() +Close() +GetBalance():double +Activate() +Reactivate() +GetLoanBalance():double

BankLoan
+Grant() +Decrease() +GetInstallment():double +Activate() +GetLoanBalance():double

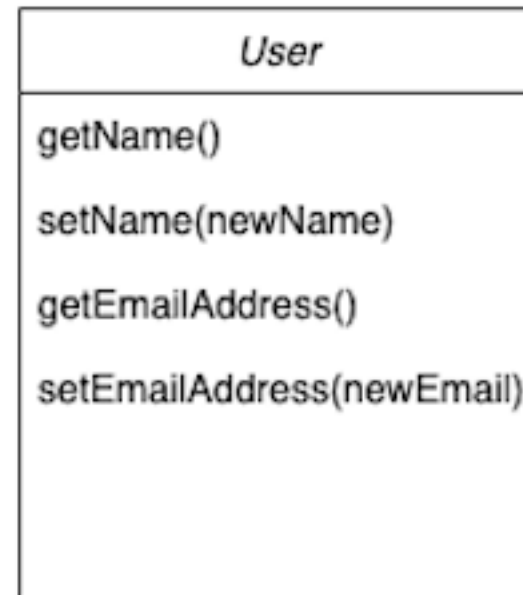
High Cohesion?! 

Cohesion

For example 2, a *User* class containing a method on how to validate the email address. *User* class can be responsible for storing the email address of the user but not for validating it or sending an email:



Low Cohesion



High Cohesion

Cohesion

- Refers to the degree to which the elements within a module or component of a software system are related to one another.
- It measures how closely the functions, procedures, or classes within a module work together to achieve a common purpose or goal.
- High cohesion is generally considered a desirable quality in software design because it leads to more maintainable, modular, and understandable code.
- Modules with high cohesion are easier to comprehend, modify as they encapsulate a specific and well-defined functionality

Types of Cohesion

1. Functional Cohesion:

This type of cohesion occurs when all elements or tasks in a module contribute to a single **well-defined function or purpose**, and there is little or no coupling between the elements.

- Functional cohesion is considered the most desirable type of cohesion as it leads to more maintainable and reusable code.

Note: Single Responsibility Principle (SRP): Ensure that each function or method within the module has a single responsibility or task.

Functional Cohesion

For example, both functions are focused on calculations related to a circle – one calculates the area, and the other calculates the circumference. The module is functionally cohesive because both functions serve a common purpose of circle-related calculations.

```
python Copy code  
  
# Example in Python  
def calculate_area(radius):  
    return 3.14 * radius * radius  
  
def calculate_circumference(radius):  
    return 2 * 3.14 * radius
```

Functional cohesion
Grouped related to
operations of a task.

Sequential Cohesion

2. Sequential Cohesion:

- Functions or methods within a module are arranged in a sequential order, where the output of one function is the input to the next.
- The output of an element is the input of other element in a module i.e., data flow between the parts.

Sequential Cohesion

Example: with Order Processing: : functions represent the steps involved in processing an order: receiving the order, validating it, processing payment, and shipping the order.

```
python Copy code  
  
# Example in Python  
def receive_order(order_details):  
    # ...  
  
def validate_order(order):  
    # ...  
  
def process_payment(order):  
    # ...  
  
def ship_order(order):  
    # ...
```


Communicational Cohesion


3. Communicational Cohesion: Functions or methods within a module operate on the same set of data.

Functions within a communicational cohesive module share data structures, variables, or parameters.

Changes to the structure or format of the shared data may affect multiple functions, potentially leading to maintenance challenges (**Drawbacks**).

For example, both functions operate on the same set of data (the list of numbers). The data is shared between the functions, and each function performs a different operation on that shared data.

python

 Copy code

```
# Example in Python
def calculate_average(nums):
    total = sum(nums)
    count = len(nums)
    return total / count

def find_maximum(nums):
    return max(nums)
```

Advantages of high cohesion

- **Improved readability and understandability:** High cohesion results in clear, focused modules with a single, well-defined purpose, making it easier for developers to understand the code and make changes.
- **Better error isolation:** High cohesion reduces the likelihood that a change in one part of a module will affect other parts, making it easier to
 - isolate and fix errors.
- **Improved reliability:** High cohesion leads to modules that are less prone to errors and that function more consistently,
 - leading to an overall improvement in the reliability of the system.

Disadvantages of Low cohesion

- Increased code duplication:

Can lead to the duplication of code, as elements that belong together are split into separate modules.

- Reduced functionality:

Can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.

- Difficulty in understanding the module:

Can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

Coupling

- Coupling refers to the degree of interdependence between software modules.
- It measures how much one module knows about or relies on the internals of another module.
- High coupling means that modules are closely connected and changes in one module may affect other modules.
- Low coupling means that modules are independent and changes in one module have little impact on other modules.

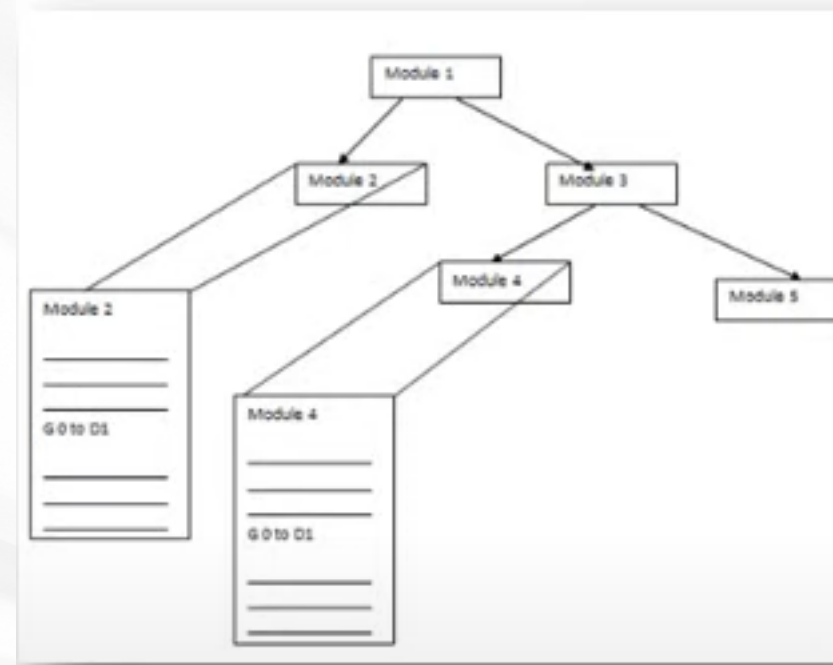
Coupling

- In general, low coupling is desirable because it leads to more modular, maintainable, and flexible software. On the other hand, high coupling can make the system more rigid, harder to maintain, and less adaptable to changes.

Types of Coupling

Type 1: Content Coupling

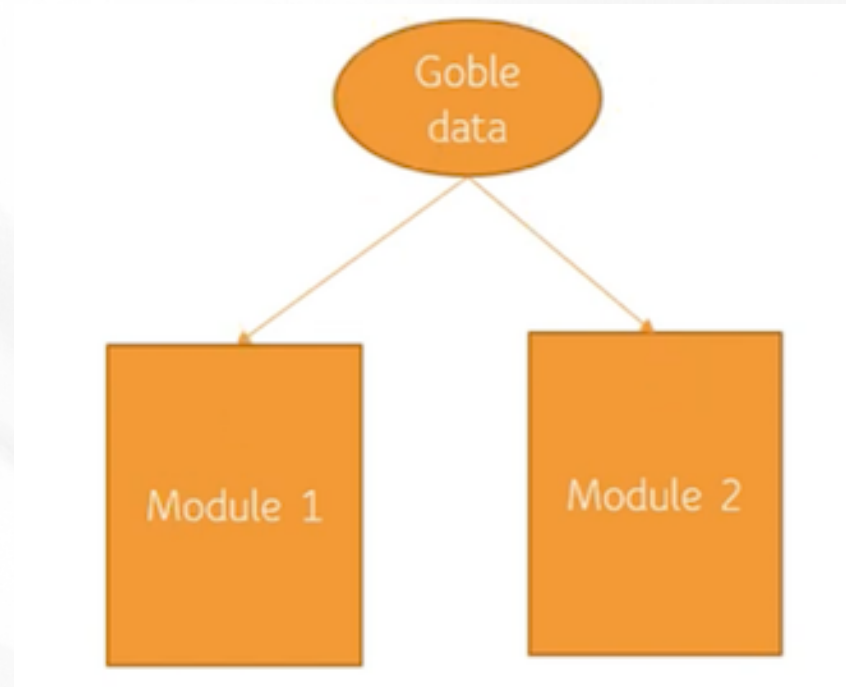
- Here, Two modules are connected as they share the same content like functions, methods.
- When a change is made in one module the other module needs to be updated as well.



Types of Coupling

Type 2: Common Coupling:

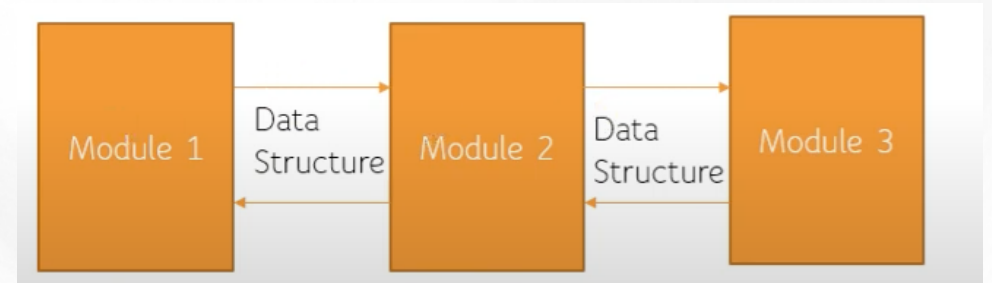
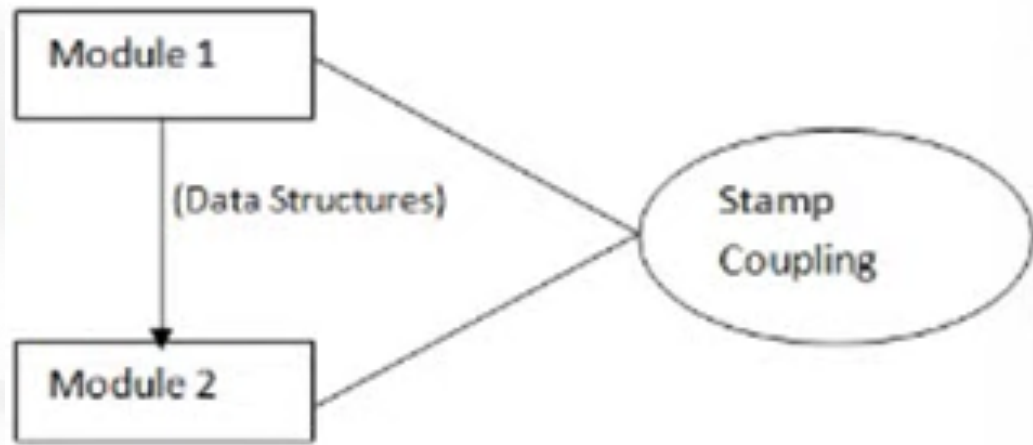
Two modules are common coupled if they share information through some global data items.



Types of Coupling

Type 3: Stamp Coupling:

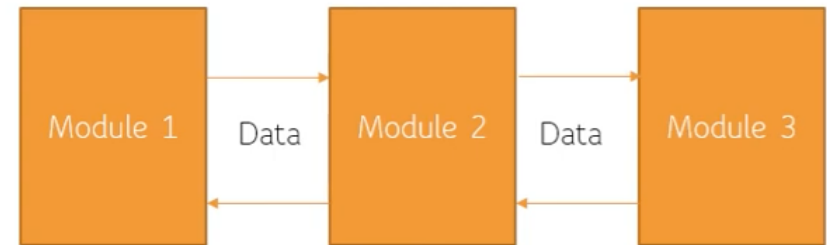
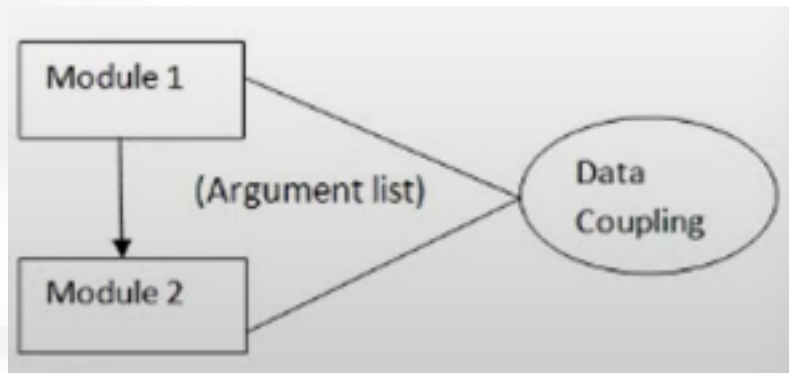
Two modules are stamp coupled if they communicate using composite data items such as Complete Data Structure & objects. The complete data structure is passed from one module to another module.



Types of Coupling

Type 4: Data Coupling

- When data are passed from one modules to another module via argument list or parameters through functional blocks.
- If the dependency between the modules is based on the data only, then the modules are said to be data coupled.



Advantages of Low coupling

- **Improved maintainability:**

Reduces the impact of changes in one module on other modules, making it easier to modify or replace individual components without affecting the entire system.

- **Enhanced modularity:**

Allows modules to be developed and tested in isolation, improving the modularity and reusability of code.

- **Better scalability:**

- facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed. Easier adaptability to new requirements.

Disadvantages of high coupling:

- Increased complexity:

Increases the interdependence between modules, making the system more complex and difficult to understand.

- Reduced flexibility:

makes it more difficult to modify or replace individual components without affecting the entire system.

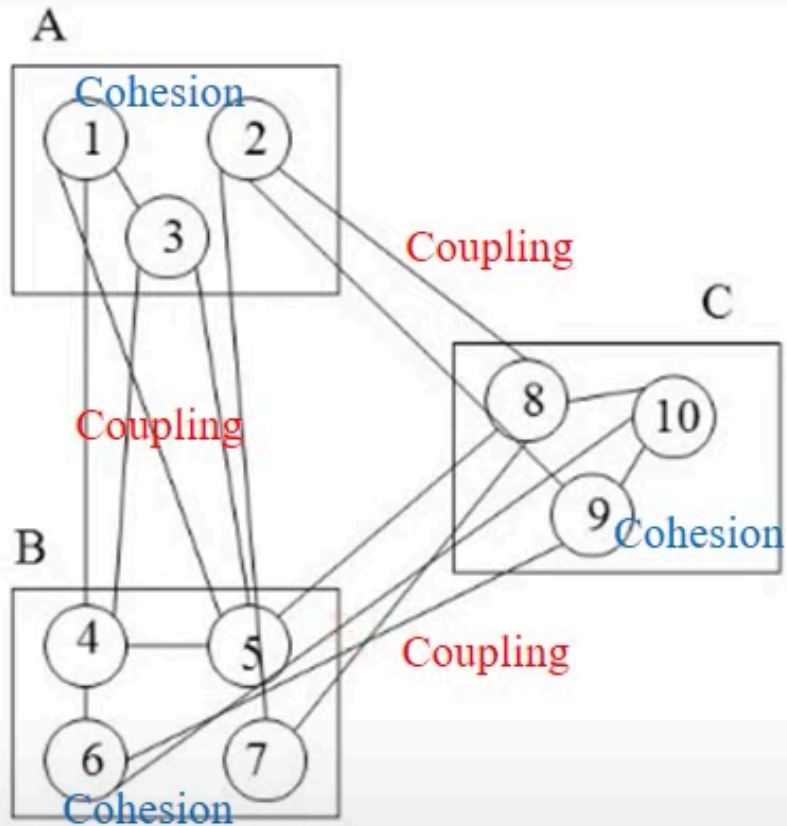
- Decreased modularity:

Makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

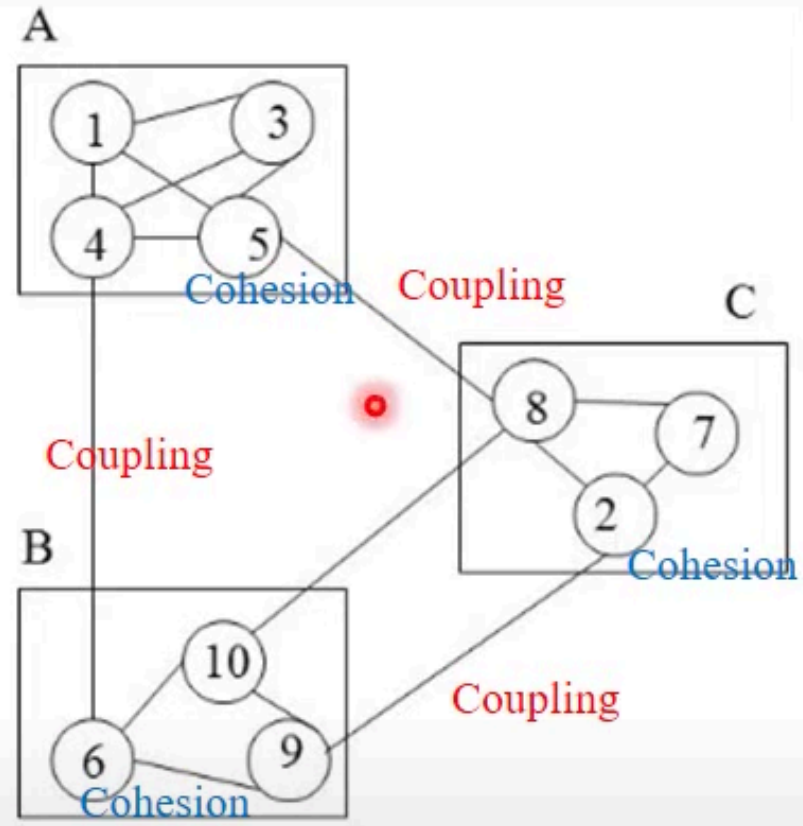
Differences between Coupling and Cohesion

Cohesion	Coupling
Cohesion represents the relationship within a module.	Coupling represents the relationships between modules.
Increasing cohesion is good for software.	Increasing coupling is avoided for software.
Cohesion represents the functional strength of module.	Coupling represents the independence among modules.
Highly cohesive gives the best software.	Whereas loosely coupling gives the best software.
In cohesion, the module focuses on a single thing.	In coupling, modules are connected to the other modules.
Cohesion is created between the same module.	Coupling is created between two different modules.
Types of Cohesion 1. Functional Cohesion. 2. Sequential Cohesion. 3. Communication Cohesion 4. Layer Cohesion.	Types of Coupling 1. Data Coupling 2. Stamp Coupling 3. Common Coupling. 4. External Coupling.

Good and Bad software design



Bad modularization:
low cohesion, high coupling

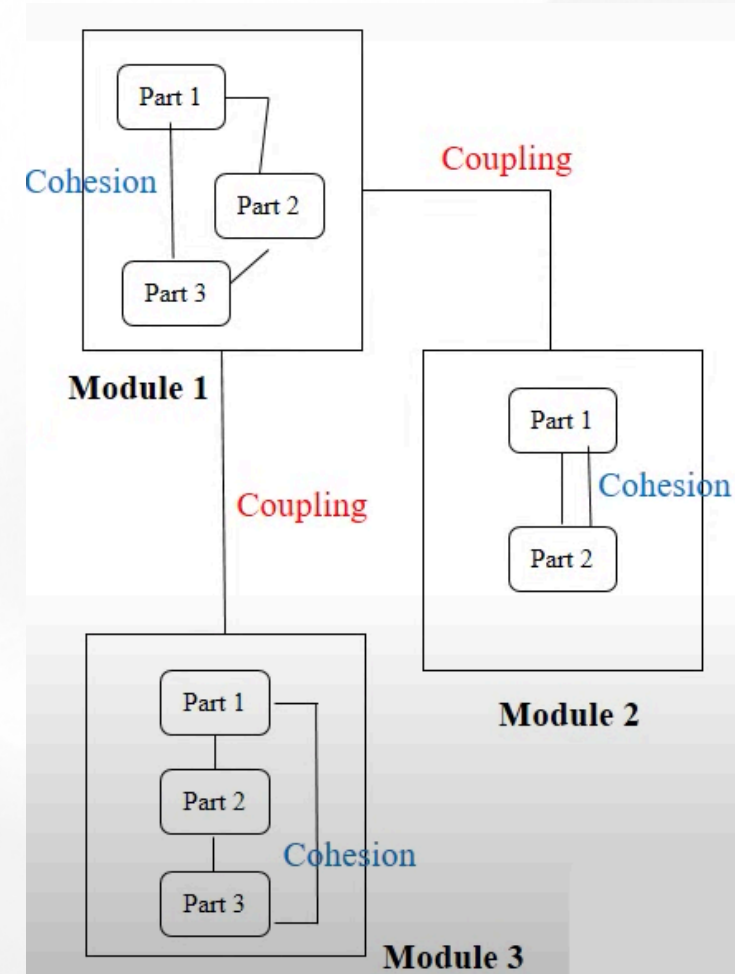


Good modularization:
high cohesion, low coupling

Why High Cohesive & Low Coupling generate good design?

➤ Due to Low Coupling

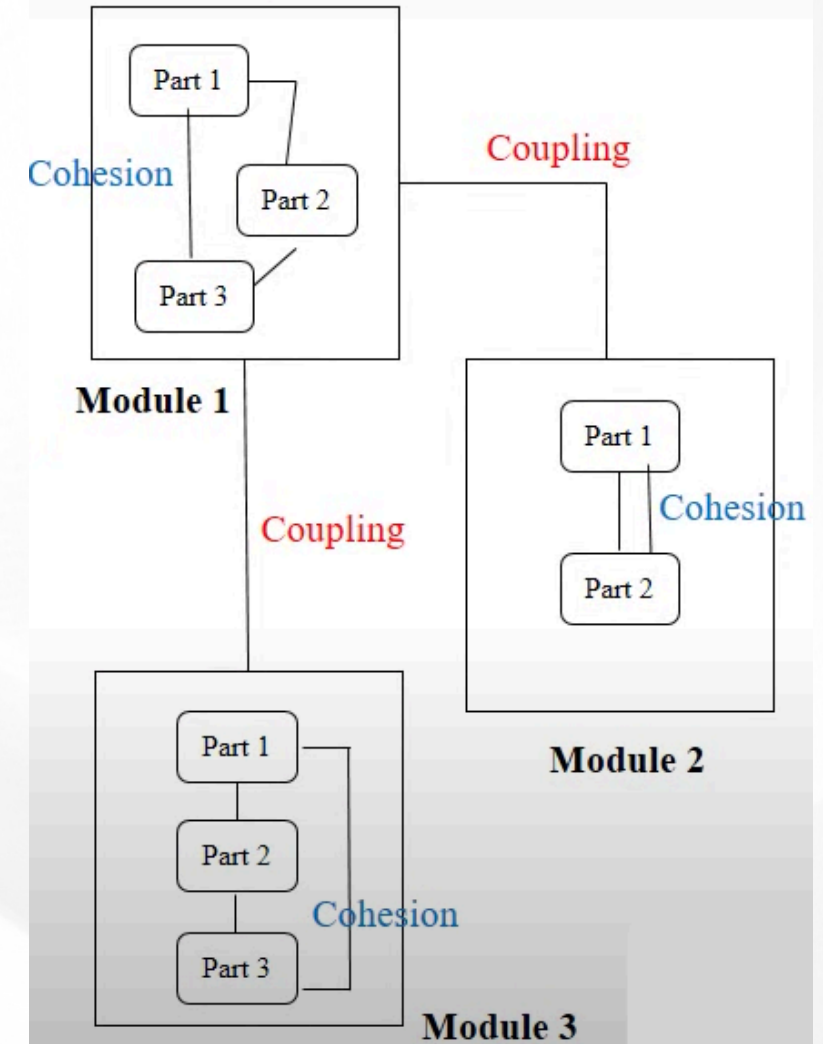
- Readability: Modules are easy to understand not complex.
- Maintainability: Changes in one module little impact on other.
- Modularity: Enhance modules development.
- Scalability: Adding new module remove existing one easy.
- Testability: Modules are easy to test & debug.



Why High Cohesive & Low Coupling generate good design?

➤ Due to High Cohesion

- Readability: Related functions easy to understand.
- Reusability: Easily Reuse module in another system.
- Reliability: Generate overall improvement of system.
- Testability: Modules are easy to test & debug.



Example 1

Let's consider a hypothetical example where we have a module responsible for handling customer data, including both personal information and purchase history. This module contains various functions, and each function deals with a different aspect of customer data:

Class: handling customer data,

```
class CustomerDataHandler:
    def get_personal_info(customer_id):
        # Code to retrieve and display personal information

    def get_purchase_history(customer_id):
        # Code to retrieve and display purchase history

    def update_personal_info(customer_id, new_info):
        # Code to update personal information

    def process_payment(customer_id, amount):
        # Code to process a payment

# ... other functions related to customer data
```

Low or High cohesion ??

Example 1

➤ Why is the `CustomerDataHandler` class not cohesive?

Answer: deal with different aspects of customer data, such as personal information, purchase history, updating personal information, and processing payments. .

Solution ?: Make three classes:

```
class CustomerInfoHandler
class PurchaseHistoryHandler.
class PaymentProcessor.
```

Example 1

```
class CustomerInfoHandler:  
    def get_personal_info(customer_id):  
        # Code to retrieve and display personal information  
  
    def update_personal_info(customer_id, new_info):  
        # Code to update personal information  
  
    # ... other functions related to personal information
```

```
class PurchaseHistoryHandler:  
    def get_purchase_history(customer_id):  
        # Code to retrieve and display purchase history  
  
    # ... other functions related to purchase history
```

```
class PaymentProcessor:  
    def process_payment(customer_id, amount):  
        # Code to process a payment  
  
    # ... other functions related to payment processing
```

High cohesion !

Example 2

In this example, the `LibraryManager` class contains functions related to adding/removing books, searching for books, calculating fines, and displaying member information

```
class LibraryManager:
    def add_book(book):
        # Code to add a book to the library

    def remove_book(book_id):
        # Code to remove a book from the library
        # ...

    def search_book_by_title(title):
        # Code to search for a book by title
        # ...

    def calculate_fine(member_id, days_overdue):
        # Code to calculate the fine for a member based on overdue days
        # ...

    def display_member_info(member_id):
        # Code to display information about a library member
        # ...

# ... other functions related to various concerns
```

Example 2

➤ Why is the LibraryManager class not cohesive?

Answer: The functions cover a wide range of concerns, resulting in lower cohesion. The class is trying to manage various aspects of the library system, making it less maintainable and harder to understand.

Solution ?: Separated concerns into three different classes:

```
class BookManager
```

```
class FineCalculator.
```

```
class MemberInfoDisplay.
```

Example 2

```
class BookManager:
    def add_book(book):
        # Code to add a book to the library

    def remove_book(book_id):
        # Code to remove a book from the library

    def search_book_by_title(title):
        # Code to search for a book by title
```

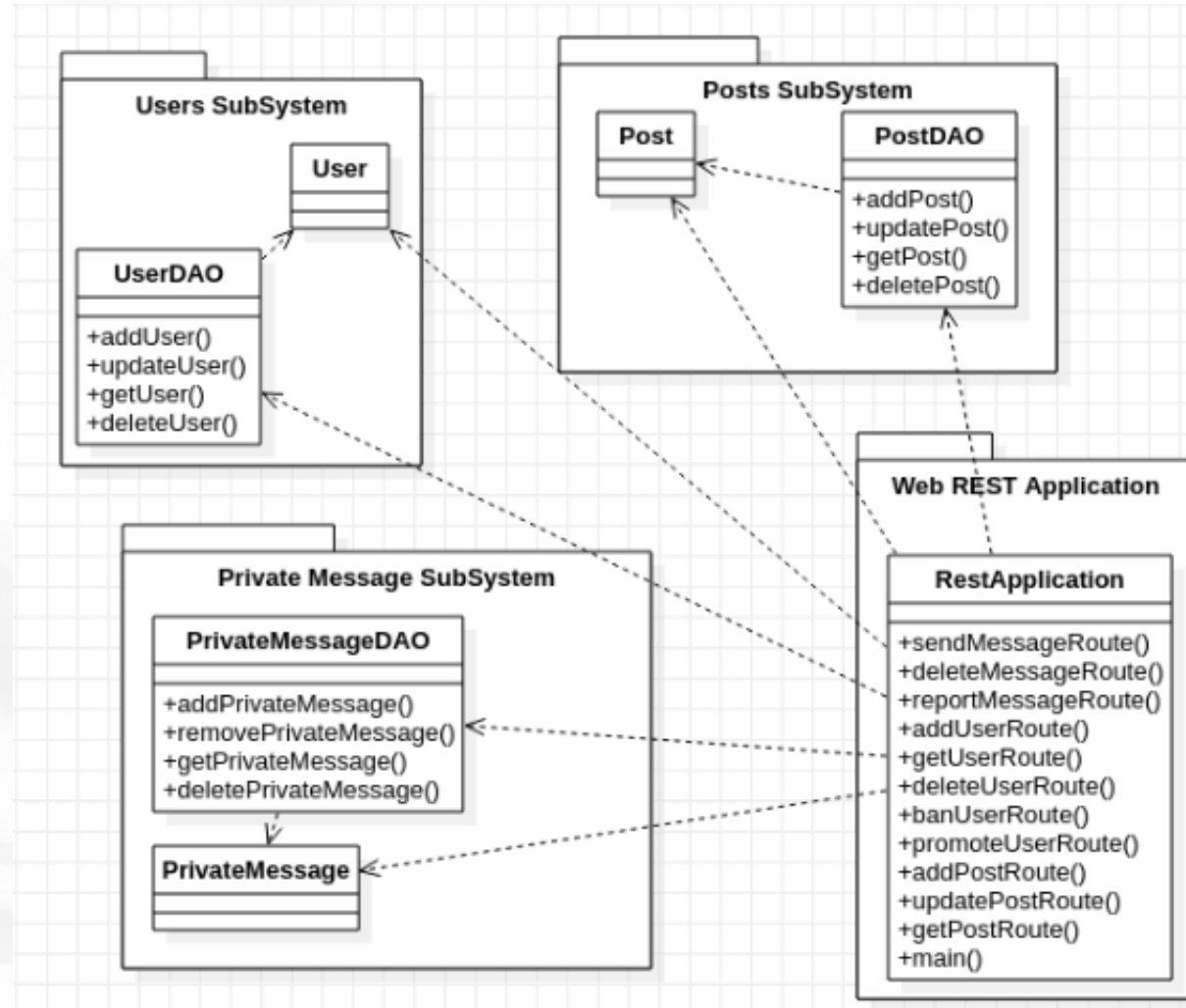
```
class FineCalculator:
    def calculate_fine(member_id, days_overdue):
        # Code to calculate the fine for a member based on overdue days
```

```
class MemberInfoDisplay:
    def display_member_info(member_id):
        # Code to display information about a library member
```

High cohesion !

Example of High Cohesion and Low Coupling

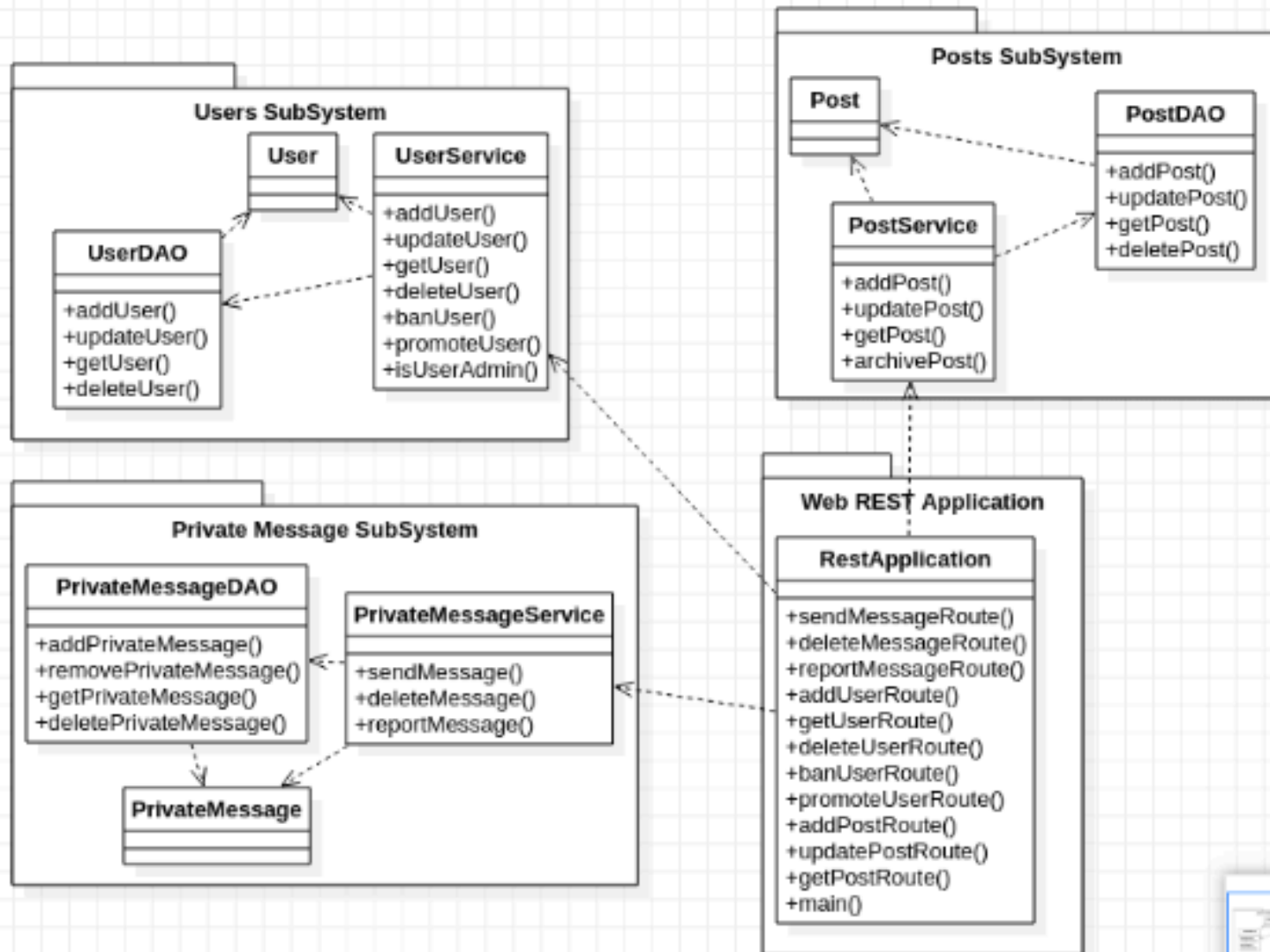
A REST API that have to manage Users, Posts and Private Message between users.



Example of High Cohesion and Low Coupling

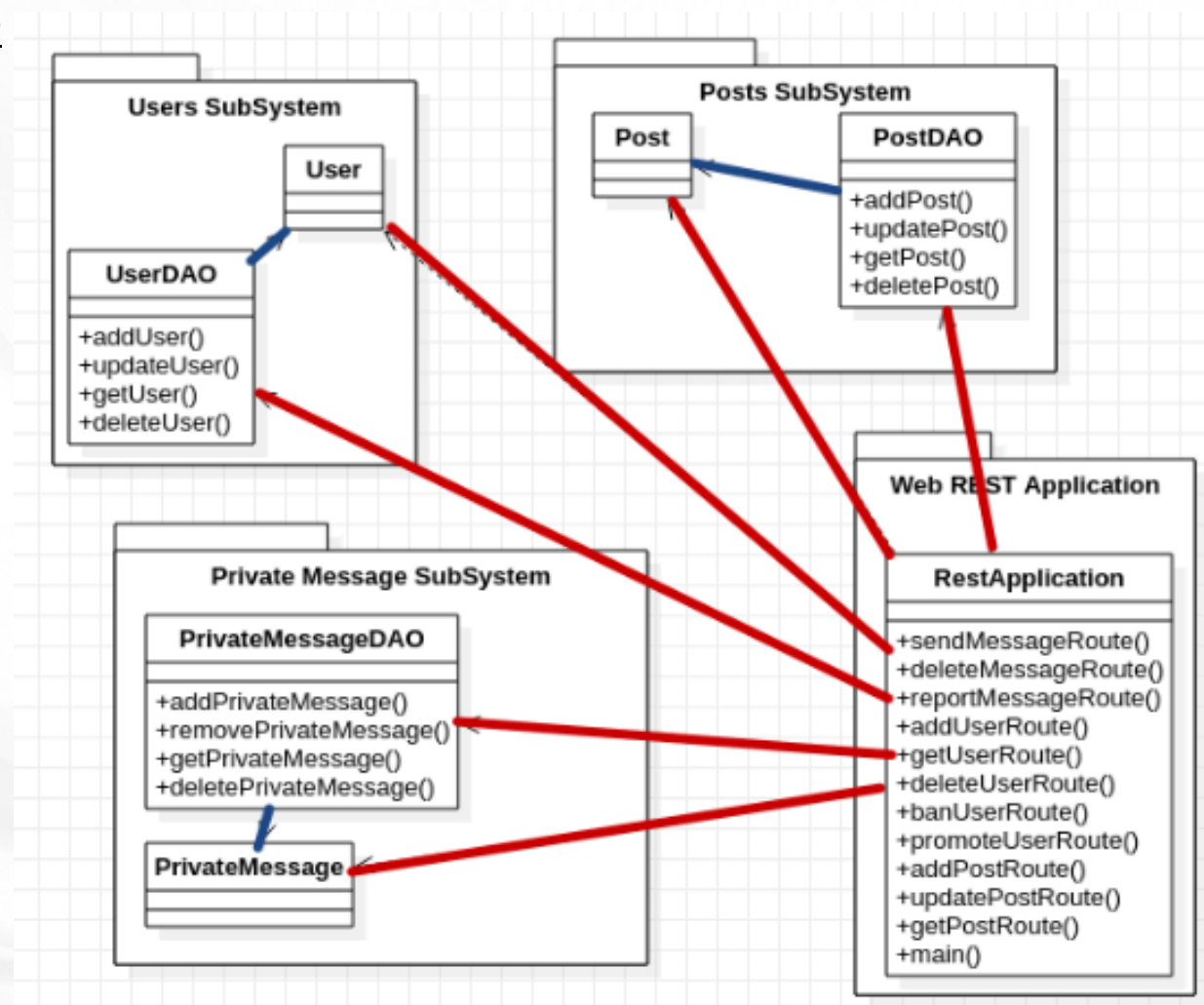
- the RestApplication class is managing the REST API requests correctly,
- but the issue here is that it also depends on all the other classes and it is managing the logic behind every single action (for example, the logic behind banning an user, reporting a message, sending a message, etc...).
- This is a clear example of **low cohesion and high coupling** because it depends heavily on internal classes of other modules.
- One bad thing about this is: what if some of the internal classes of the other modules changes? Then you will have to change the Application class to make this work

Example of High Cohesion and Low Coupling



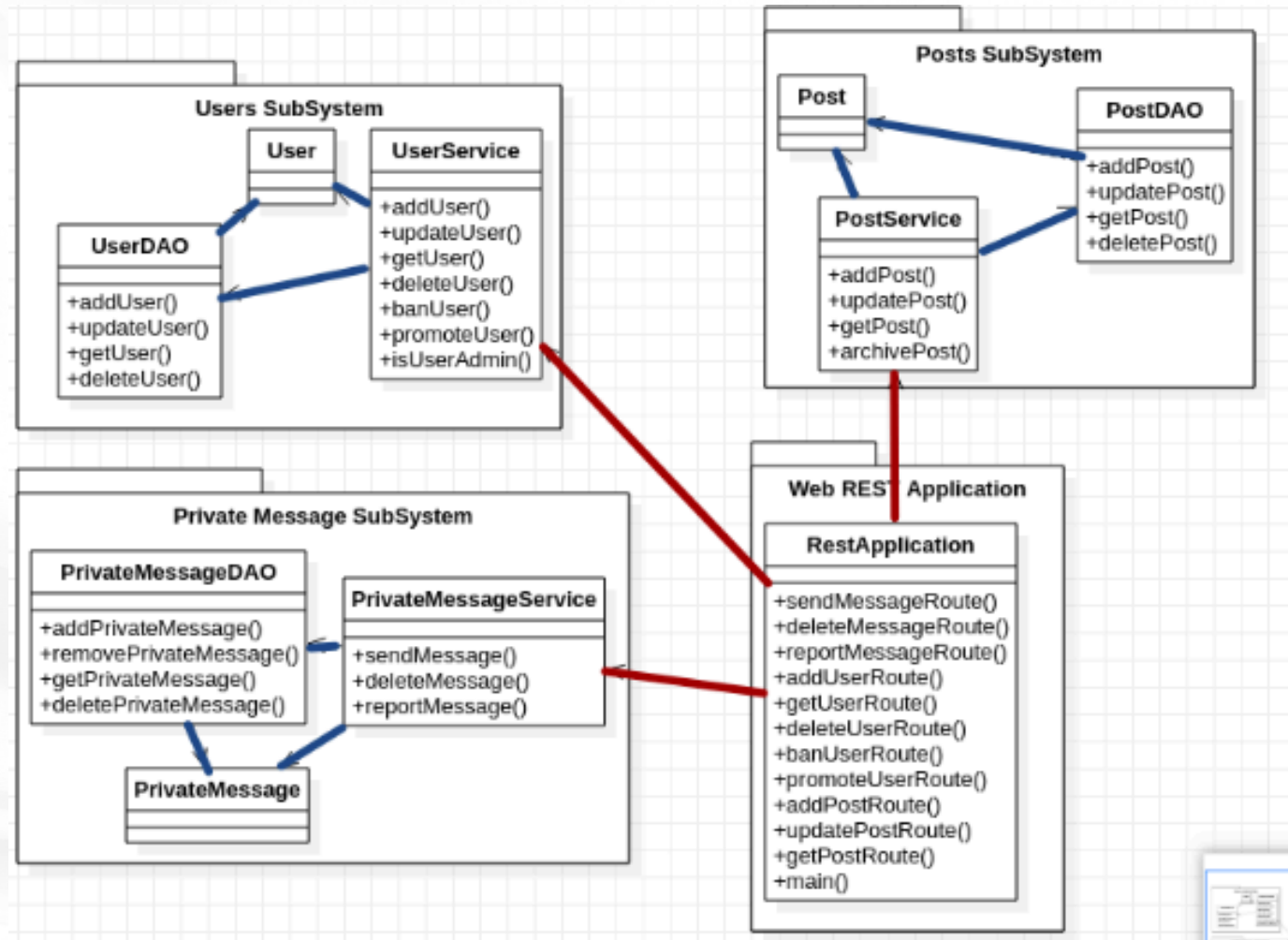
Example of High Cohesion and Low Coupling

Low Cohesion (**BLUE**), High Coupling (**RED**) - BAD



Example of High Cohesion and Low Coupling

High Cohesion (**BLUE**), Low Coupling (**RED**) - BETTER



Summary

- Coupling and cohesion are two important principles in software design that can be used to improve the quality of a system's architecture.
- Cohesion: "degree to which the elements inside a module belong together"
- Coupling "degree of interdependence between software modules"

Exercises

1) is a measure of the degree of interdependence between modules.

a) Cohesion

b) Coupling

c) None of the mentioned

d) All of the mentioned

The End

