Mid 2 SDA

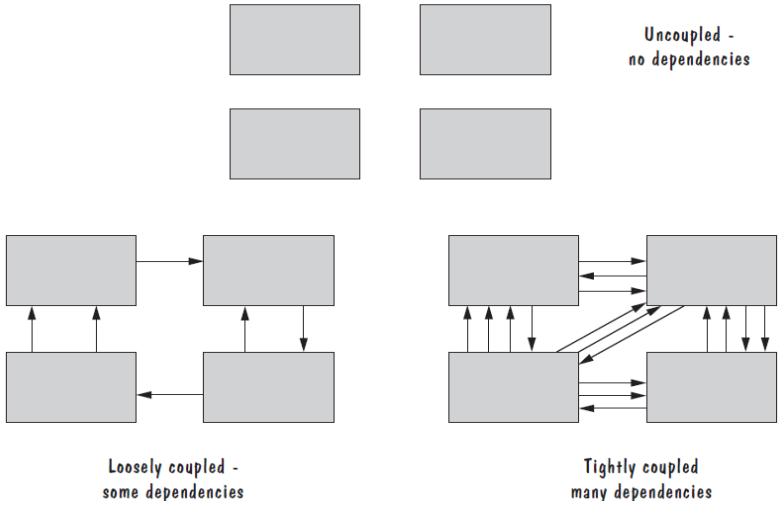
- **Modularity** is the principle of keeping the unrelated aspects of a system separate from each other, each aspect can be studied in isolation (also called **separation of concerns**)
- If the principle is applied well, each resulting module will have a **single purpose** and will be relatively **independent** of the others

1. Each module will be easy to **understand** and **develop**
2. Easier to **locate faults**
3. Easier to **change** the system

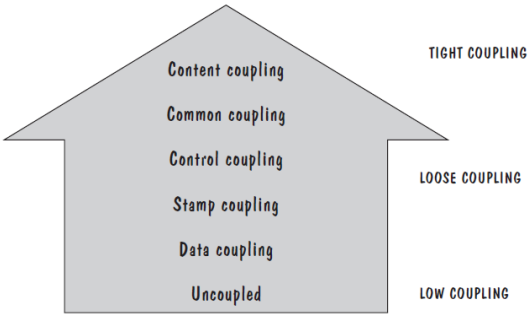
- Two concepts that measure **module independence**: coupling and cohesion

Coupling

- Two modules are **tightly coupled** when they depend a great deal on each other
- **Loosely coupled** modules have some dependence, but their interconnections are weak
- **Uncoupled** modules have no interconnections at all; they are completely unrelated



- There are many ways that modules can depend on each other:
    1. The references made from one module to another
    2. The amount of data passed from one module to another
    3. The amount of control that one module has over the other



High coupling is not desired

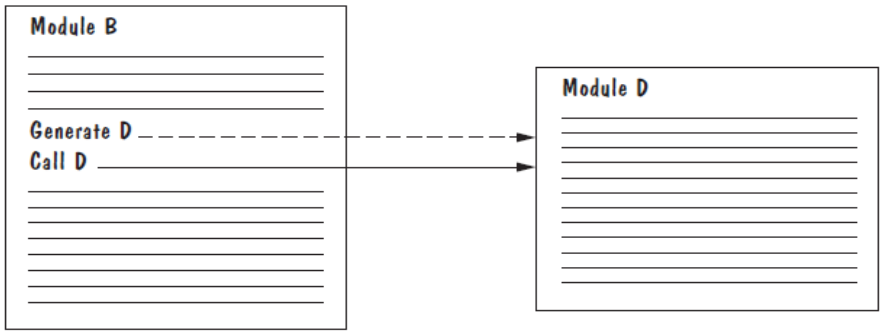
Content Coupling

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. This is the worst form of coupling and should be avoided.

1. One **module modifies another**. The modified module is completely dependent on the modifying one
2. One class modifies the content of another class. For example, in C++, friend classes can access each other's private members.
3. Content coupling might occur when one module is imported into another module, modifies the code of another module, or branches into the middle of another module

Suppose Class A has a private instance member int age; Class B sets the value of age directly by a.age = 15; This is content coupling.

Content coupling, as the name says, is a case where two modules share their contents, and when a change occurs in one module, the other module needs updating as well

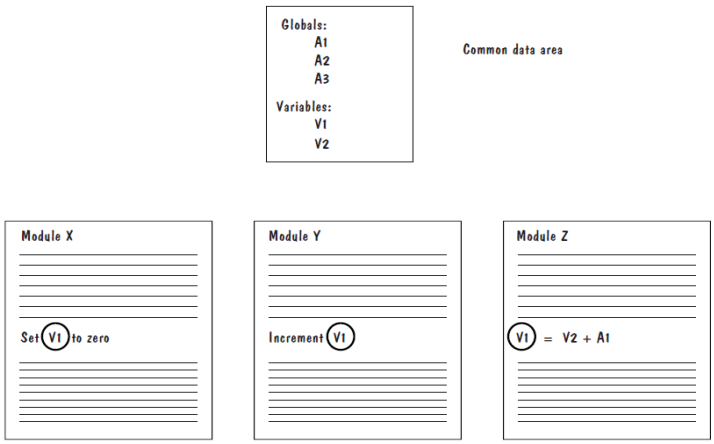


Common Coupling

We can reduce the amount of coupling somewhat by organizing our design so that **data is accessible from a common data store**.

Dependence still exists; making a change to the common data means that, to evaluate the effect of the change, we must look at all modules that access those data.

With common coupling, it can be difficult to determine which module is responsible for having set a variable to a particular value.



Control Coupling

A module is said to be control coupled when the flow of execution is decided by a variable of another class. This means which lines of code should be executed is decided by some parameter or a variable. This is more commonly found where there are control structures like if-else conditions.

```
public class ClassA {
    public void method() {
    CouplingExample couplingExample = new CouplingExample();
        couplingExample.methodA("A");
    }
}

public class CouplingExample {
    public void methodA(String arg) {
        if (args[0].equals("A")) {
            PrintA printA = new PrintA();
            printA.print();
        } else {
            PrintB printB = new PrintB();
            printB.print();
        }
```

ClassA is controlling the execution flow of CouplingExample class.

1. When one module passes **parameters** or a **return code** to control the behavior of another module
2. It is impossible for the controlled module to function without some direction from the controlling module
3. Limit each module to be responsible for only one function or one activity.
4. Restriction minimizes the amount of information that is passed to a controlled module
5. It simplifies the module's interface to a fixed and recognizable set of parameters and return values.

```
bool foo(int x){
    if (x == 0)
        return false;
    else
        return true;
}

void bar(){
    // Calling foo() by passing a value which controls its flow:
    foo(1);
}
```

Stamp Coupling

In stamp coupling, the complete data structure is passed from one module to another module.

- When complex data structures are passed between modules, we say there is **stamp coupling** between the modules

```python
# Module A
def calculate_area(rectangle):
    return rectangle['length'] * rectangle['width']

# Module B
def calculate_perimeter(rectangle):
    return 2 * (rectangle['length'] + rectangle['width'])

# Main program
if __name__ == "__main__":
    my_rectangle = {'length': 5, 'width': 3}
    area = calculate_area(my_rectangle)
    perimeter = calculate_perimeter(my_rectangle)
    print(f"Area: {area}, Perimeter: {perimeter}")
```

In this example, Module A and Module B have a stamp coupling relationship. They both receive a data structure (the 'rectangle' dictionary) as input, and they operate on that data to perform their respective calculations (area and perimeter).

```
class A{
   // Code for class A.
};

class B{
    // Data member of class A type: Type-use coupling
    A var;

    // Argument of type A: Stamp coupling
    void calculate(A data){
        // Do something.
    }
};
```

Data Coupling

If only data values, and not structured data, are passed, then the modules are connected by **data coupling**

Data coupling is simpler and less likely to be affected by changes in data representation.

Easiest to trace data through and to make changes to data coupled modules.

```python
# Module A
def calculate_area(length, width):
    return length * width

# Module B
def calculate_perimeter(length, width):
    return 2 * (length + width)

# Main program
if __name__ == "__main__":
    length = 5
    width = 3
    area = calculate_area(length, width)
    perimeter = calculate_perimeter(length, width)
    print(f"Area: {area}, Perimeter: {perimeter}")
```

Cohesion

Cohesion refers to the dependence within and among a module's internal elements (e.g., data, functions, internal modules)

The more cohesive a module, the more closely related its pieces are.

Coincidental Cohesion

The worst degree of cohesion, **coincidental**, is found in a module whose parts are unrelated to one another

Unrelated functions, processes, or data are combined in the same module for reasons of convenience

It is accidental and the worst form of cohesion.

Example:

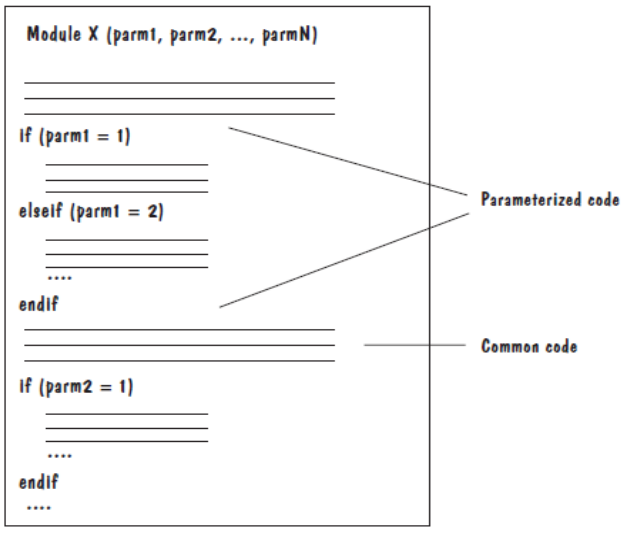
1. Fix Car
2. Bake Cake
3. Walk Dog
4. Fill our Astronaut-Application Form
5. Get out of Bed

Logical Cohesion

A module has **logical cohesion** if its parts are related only by the logic structure of its code.

The elements are logically related and not functionally.

Example: *A component reads inputs from tape, disk, and network. All the code for these functions is in the same component. Operations are related, but the functions are significantly different.*



1. Go by Car
2. Go by Train
3. Go by Boat
4. Go by Plane

Temporal Cohesion

Elements of component are related by timing

A module has temporal cohesion when it performs a series of operations related in time

The elements are related by their timing involved. A module connected with temporal cohesion all the tasks must be executed in the same time span. This cohesion contains the code for initializing all the parts of the system. Lots of different activities occur, all at unit time.

```
void logMessage(const std::string& message) {
    // Log the message}
void sendNotification(const std::string& message) {
    // Send a notification
}
void displayError(const std::string& errorMessage) {
    // Display an error message

int main() {
    logMessage("Application started.");
    sendNotification("Important notification.");
    displayError("An error occurred.");
    logMessage("Application finished.");}
```

The functions **logMessage**, **sendNotification**, and **displayError** are grouped together in the **main** routine and are executed at specific moments in time during the execution of the program. These functions are not logically related; they are simply called at particular times or events within the program's execution flow.

Procedural Cohesion

When functions are grouped together in a module to encapsulate the order of their execution, we say that the module is **procedurally cohesive**.

Elements of procedural cohesion ensure the order of execution. Actions are still weakly connected and unlikely to be reusable.

Example- *calculate student GPA, print student record, calculate cumulative GPA, print cumulative GPA.*

Procedural cohesion is a type of cohesion in software design where the elements or functions within a module or component are grouped together because they perform a sequence of related steps or operations as part of a common procedure. In other words, functions within a module with procedural cohesion work together to achieve a specific task or procedure.

```cpp
// Function to calculate the area of a rectangle
double calculateRectangleArea(double Length, double width) {
    return Length * width;
}

// Function to calculate the perimeter of a rectangle
double calculateRectanglePerimeter(double Length, double width) {
    return 2 * (Length + width);
}

int main() {
    double length = 5.0;
    double width = 3.0;

    // Calculate the area and perimeter of a rectangle
    double area = calculateRectangleArea(length, width);
    double perimeter = calculateRectanglePerimeter(length, width);

    // Display the results
    std::cout << "Rectangle Area: " << area << std::endl;
    std::cout << "Rectangle Perimeter: " << perimeter << std::endl;
```

The functions **calculateRectangleArea** and **calculateRectanglePerimeter** are grouped together because they work together to perform a sequence of related steps to calculate the area and perimeter of a rectangle. These functions are logically related and serve a common purpose: computing properties of a rectangle.

Communicational Cohesion

Associate certain functions because they operate on the same data set

Two elements operate on the same input data or contribute towards the same output data. Example- update record in the database and send it to the printer.

1. Find Title of Book
2. Find Price of Book
3. Find Publisher of Book
4. Find Author of Book

```cpp
// Function to read a list of numbers from the user
void readNumbers(std::vector<int>& numbers) {
    int num;
    std::cout << "Enter numbers: "
        numbers.push_back(num);

}

// Function to calculate and display the sum of numbers
void calculateAndDisplaySum(const std::vector<int>& numbers) {
    int sum = 0;
    for (int num : numbers) {
        sum += num;
    }
    std::cout << "Sum of numbers: " << sum << std::endl;
}

int main() {
    std::vector<int> numbers;
    readNumbers(numbers);
    calculateAndDisplaySum(numbers);
```

The functions **readNumbers** and **calculateAndDisplaySum** are grouped together because they share the same data (the **numbers** vector) and communicate with each other to achieve a common objective, which is to read a list of numbers from the user, calculate their sum, and display the result. **Desirable form of Cohesion.**

Functional Cohesion

All elements essential to a single function are contained in one module, and all of that module's elements are essential to the performance of that function

A functionally cohesive module performs only the function for which it is designed, and nothing else

Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. It is an ideal situation.

```cpp
// Function to calculate the area of a rectangle
double calculateRectangleArea(double length, double width) {
    return length * width;
}

int main() {
    double length = 5.0;
    double width = 3.0;

    // Calculate the area of a rectangle
    double area = calculateRectangleArea(length, width);
    // Display the result
    std::cout << "Rectangle Area: " << area << std::endl;
```

The function **calculateRectangleArea** serves a single, clear purpose: to calculate the area of a rectangle. The module contains only this function, and it is designed to perform this specific task.

Functional cohesion is the most desirable form of cohesion because it results in highly modular and organized code.

Sequential Cohesion

Sequential cohesion is when parts of a module are grouped because the output from one part is the input to another part like an assembly line

An element outputs some data that becomes the input for other element, i.e., data flow between the parts. It occurs naturally in functional programming languages.

```cpp
// Function to initialize a data structure
void initializeDataStructure() {
    std::cout << "Data structure initialized." << std::endl;
}
// Function to populate data into the data structure
void populateData() {
    std::cout << "Data populated." << std::endl;
}
// Function to process data
void processData() {
    std::cout << "Data processed." << std::endl;
}

int main() {
    initializeDataStructure();  // First step
    populateData();             // Second step
    processData();              // Third step
```

The functions **initializeDataStructure**, **populateData**, and **processData** must be executed in a specific sequence to achieve the desired outcome. Their order of execution is important, and changing the sequence may lead to incorrect results.

Sequential cohesion is typically used when certain operations or tasks must occur in a specific order and their individual functionality is less important than the sequence itself. It is important to document and maintain the order of execution in such modules to ensure the intended behavior is achieved.

Informational Cohesion

Informational cohesion occurs when elements or tasks are grouped together in a module based on their relationship to a specific data structure or object, such as a module that operates on a specific data type or object. Informational cohesion is commonly used in object-oriented programming.

Informational cohesion is a type of cohesion in software design where the elements or functions within a module or component are grouped together because they all operate on the same set of data or share a common purpose or theme.

```cpp
// Function to read and display a list of names
void readAndDisplayNames(const std::vector<std::string>& names) {
    std::cout << "List of Names:" << std::endl;
    for (const std::string& name : names) {
        std::cout << name << std::endl;
    }
}

int main() {
    std::vector<std::string> names = {"Alice", "Bob", "Charlie", "David"};
    // Read and display the list of names
    readAndDisplayNames(names);
```

In this example, the module exhibits informational cohesion. The function **readAndDisplayNames** is focused on the common theme of handling and displaying a list of names. The entire module revolves around this theme, and the functions within the module are all related to the same data or purpose.

Informational cohesion is a desirable form of cohesion because it organizes functions that work with the same data or serve a common theme

Advantages of Low coupling

- **Improved maintainability:** Low coupling 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**: Low coupling allows modules to be developed and tested in isolation, improving the modularity and reusability of code.
- **Better scalability:** Low coupling facilitates the addition of new modules and the removal of existing ones, making it easier to scale the system as needed.

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 High Coupling

- **Increased complexity:** High coupling increases the interdependence between modules, making the system more complex and difficult to understand.
- **Reduced flexibility**: High coupling makes it more difficult to modify or replace individual components without affecting the entire system.
- **Decreased modularity**: High coupling makes it more difficult to develop and test modules in isolation, reducing the modularity and reusability of code.

Disadvantages of Low Cohesion

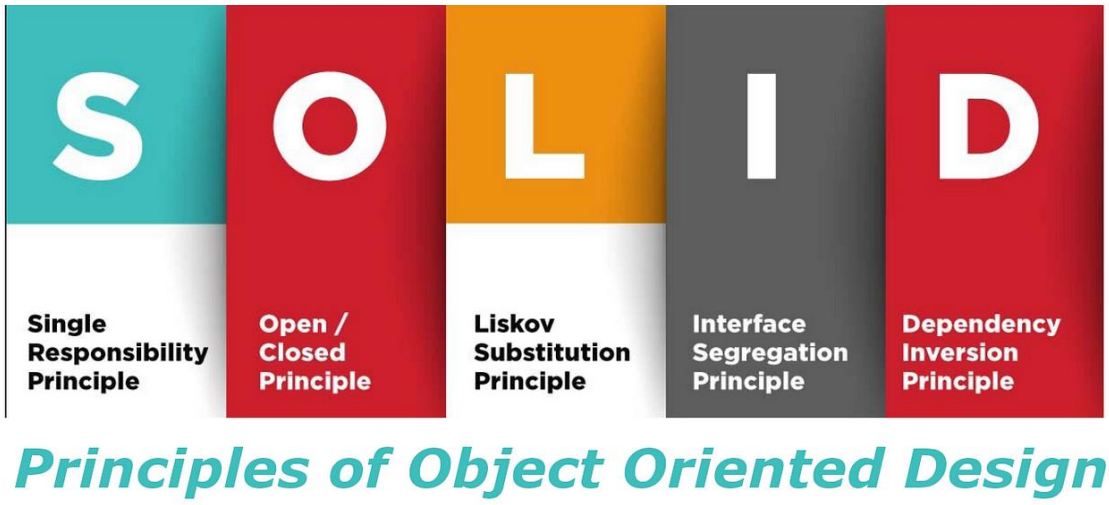
- **Increased code duplication:** Low cohesion can lead to the duplication of code, as elements that belong together are split into separate modules.
- **Reduced functionality**: Low cohesion can result in modules that lack a clear purpose and contain elements that don't belong together, reducing their functionality and making them harder to maintain.
- **Difficulty in understanding the module:** Low cohesion can make it harder for developers to understand the purpose and behavior of a module, leading to errors and a lack of clarity.

Solid Principles

The SOLID principles tell us how to **arrange our functions and data structures** into classes, and how those classes should be **interconnected**

The goal of the principles is the creation of mid-level software structures that:

1. Tolerate change
2. Are easy to understand
3. Are the basis of components that can be used in many software systems



SRP-Single Responsibility Principle

This principle states that ***"a class should have only one reason to change"*** which means every class should have a single responsibility or single job or single purpose.

- Take the example of developing software. The task is divided into different members doing different things as front-end designers do design, the tester does testing and backend developer takes care of backend development part then we can say that everyone has a single job or responsibility.
- A module should have a reason to change by only **one actor.**
- A module should be responsible to **one and only one actor**
- Only one potential change (database logic, logging logic, and so on.) in the software's specification should be able to affect the specification of the class.
- Many different teams can work on the same project and edit the same class for different reasons, this could lead to **incompatible modules**.
- It makes **version control easier**. For example, say we have a persistence class that handles database operations, and we see a change in that file in the GitHub commits. By following the SRP, we will know that it is related to storage or database-related stuff.
- **Merge conflicts** are another example. They appear when different teams change the same file. But if the SRP is followed, fewer conflicts will appear – files will have a single reason to change, and conflicts that do exist will be easier to resolve.

OCP-Open Closed Principle

This principle states that "***software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification***" which means you should be able to extend a class behavior, without modifying it.

- Suppose developer A needs to release an update for a library or framework and developer B wants some modification or add some feature on that then developer B is allowed to extend the existing class created by developer A but developer B is not supposed to modify the class directly. Using this principle separates the existing code from the modified code so it provides better stability, maintainability and minimizes changes as in your code.
- Modification means **changing the code** of an existing class, and extension means **adding new functionality**. It is usually done with the help of **interfaces** and **abstract classes.**
- We should be able to add new functionality without touching the existing code for the class.
- This is because whenever we modify the existing code, we are taking the **risk of creating potential bugs**.
- We should avoid touching the tested and reliable (mostly) production code if possible.

Liskov's Substitution Principle

The principle was introduced by Barbara Liskov in 1987 and according to this principle ***"Derived or child classes must be substitutable for their base or parent classes"***.

- This principle ensures that any class that is the child of a parent class should be usable in place of its parent without any unexpected behavior.
- You can understand it in a way that a farmer's son should inherit farming skills from his father and should be able to replace his father if needed. If the son wants to become a farmer then he can replace his father but if he wants to become a cricketer then definitely the son can't replace his father even though they both belong to the same family hierarchy.

```
class Bird {
public:
    virtual void fly() {
        cout << "A bird can fly" << endl;
    }
};

class Sparrow : public Bird {
public:
    void fly() override {
         cout << "Sparrow can fly" << endl;
    }
};

class Penguin : public Bird {
public:
    void fly() override {
        cout << "Penguin cannot fly" << endl;
```

```
    }
};

void makeBirdFly(Bird* bird) {
    bird->fly();
}

int main() {
Bird bird;
    Sparrow sparrow;
Penguin penguin;

    makeBirdFly(&bird);       // Output: A bird can fly
    makeBirdFly(&sparrow);    // Output: Sparrow can fly
    makeBirdFly(&penguin);    // Output: Penguin cannot fly
}
```

In this C++ example, we have a base class **Bird** with a virtual function **fly**, and two derived classes, **Sparrow** and **Penguin**, which override the **fly** function. The **makeBirdFly** function takes a pointer to a **Bird** object and calls its **fly** method.

When we pass objects of **Bird**, **Sparrow**, or **Penguin** to the **makeBirdFly** function, the program works as expected and demonstrates the Liskov Substitution Principle. Objects of derived classes (**Sparrow** and **Penguin**) can be used interchangeably with the base class (**Bird**) without causing any issues, and the behavior of the program remains consistent.

Interface Segregation Principle:

This principle is the first principle that applies to Interfaces instead of classes in SOLID and it is similar to the single responsibility principle.
It states that "***do not force any client to implement an interface which is irrelevant to them***".

- Here your main goal is to focus on avoiding fat interface and give preference to many small client-specific interfaces.
- You should prefer many client interfaces rather than one general interface and each interface should have a specific responsibility.
- Suppose if you enter a restaurant and you are pure vegetarian. The waiter in that restaurant gave you the menu card which includes vegetarian items, non-vegetarian items, drinks, and sweets.
- In this case, as a customer, you should have a menu card which includes only vegetarian items, not everything which you don't eat in your food. Here the menu should be different for different types of customers. The common or general menu card for everyone can be divided into multiple cards instead of just one. Using this principle helps in reducing the side effects and frequency of required changes.

Dependency Inversion Principle

Now two key points are here to keep in mind about this principle

- High-level modules/classes should not depend on low-level modules/classes. Both should depend upon abstractions.
- Abstractions should not depend upon details. Details should depend upon abstractions.

The above lines simply state that if a high module or class will be dependent more on low-level modules or class then your code would have tight coupling and if you will try to make a change in one class it can break another class which is risky at the production level. So always try to make classes loosely coupled as much as you can and you can achieve this through *abstraction*. The main motive of this principle is decoupling the dependencies so if class A changes the class B doesn't need to care or know about the changes.
You can consider the real-life example of a TV remote battery. Your remote needs a battery but it's not dependent on the battery brand. You can use any XYZ brand that you want and it will work. So we can say that the TV remote is loosely coupled with the brand name. Dependency Inversion makes your code more reusable