# Module Coupling and Cohesion in Software Design

A Guide to Building Better Software

# Understanding Coupling

- The degree of interdependence between modules.

# Content Coupling (Worst)

- Description: One module directly modifies data or instructions within another module.
- Desirability: Avoid

# Common Coupling

- Description: Modules share global data.
- Desirability: Minimize

Imagine a team working on a big project, like building a website. They have a shared document with everyone's login passwords for different tools and systems.

# Control Coupling

- Description: One module passes flags or parameters to control the flow of another.
- Desirability: Use with caution

Module A sends specific instructions to Module B, like:

- "Save this data now."
- "Display this message in a red box."
- Essentially we want to give Module B its own autonomy to make decisions.

# Stamp Coupling

- Description: Modules share a composite data structure and use only parts of it.
- Desirability: Use with caution

  Imagine a system with two modules:

  Module A: Collects user information (name, address, age, purchase history, etc.) and stores it in a "User" object.

  Module A sends the entire "User" object to Module B, even though Module B only needs the "purchase history" field. Now Module B has access to unnecessary information like the user's age and address.

# Data Coupling (Best)

- Description: Modules communicate only through parameters, passing only necessary data.
- Desirability: Strive for this

  When the Dovahkin successfully hits the dragon, the "Sword Swing" module sends the damage value (data) to the "Dragon Health" module. The "Dragon Health" module then subtracts that damage from the dragon's total health.

# Understanding Cohesion

- The degree to which elements within a module belong together.

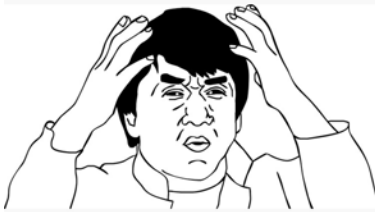
You create a module called "CircleDrawer". This module has everything needed to draw circles:

- Draw function: Takes in the circle's size and position and draws it on the screen.
- Color options: Lets the user pick the color of the circle.
- Fill function: Decides if the circle should be filled with color or just an outline.

# Coincidental (Worst)

- Description: Elements within a module have no clear relationship.
- Desirability: Avoid

A toolbox containing: a hammer, a toothbrush, a spoon, a lightbulb, and a single sock.



# Logical

- Description: Elements perform similar activities but are grouped **arbitrarily**.
- Desirability: Improve

Think of a "utility" module in software. It might contain functions for:

String manipulation (e.g., trimming whitespace, converting to uppercase)

Date and time formatting

Basic mathematical operations

# Temporal

- Description: Elements are grouped by **when** they are processed.
- Desirability: Improve

  Think of it like your morning routine:

  You might have a set of actions you perform every morning: brushing your teeth, taking a shower, getting dressed, eating breakfast. These actions are temporally related because you do them in sequence at the start of your day, but they don't **necessarily** have a strong functional relationship.

# Procedural

- Description: Elements are grouped by their **order** of execution.
- Desirability: Improve

Think of it like following a recipe:

A recipe outlines a series of steps to prepare a dish: chopping vegetables, boiling water, mixing ingredients, and so on. These steps are procedurally related, but they might involve different tools and techniques.

# Communicational

- Description: Elements operate on the same **data**.
- Desirability: Good

You have carpenters, electricians, plumbers, and painters all working on the same house. They have different skills and tasks, but they all contribute to the shared goal of completing the house. The “data structure” in this case is the house.

# Functional (Best)

- Description: All elements contribute to a single, well-defined **task**.
- Desirability: Strive for this

  Each musician plays a specific instrument, and they all follow the conductor to create a harmonious musical piece. Every instrument and musician contributes to the single purpose of performing the music.

# Why it Matters

- - Increased maintainability
- - Improved reusability
- - Enhanced testability
- - Reduced complexity

# Modularization

- Description: Breaking down a system into smaller, independent modules.

```
# Example: Separate modules for calculations and display
# calculations.py
def add(x, y):
    return x + y

# display.py
import calculations
result = calculations.add(5, 3)
print(f'The result is: {result}')
```

# Abstraction

Hiding complex implementation details behind simple interfaces.

```
# Example: Using a function to hide the details of file reading
def get_file_contents(filename):
    # Code to open and read the file (hidden from the user)
    with open(filename, 'r') as f:
        contents = f.read()
    return contents
contents = get_file_contents('my_file.txt')
print(contents)
```

# Encapsulation

Bundling data and methods that operate on that data within a class.

```
# Example: A Dog class with name and bark() method
class Dog:
    def __init__(self, name):
        self.name = name

    def bark(self):
        return 'Woof!'
    my_dog = Dog('Buddy')
print(my_dog.name)  # Output: Buddy
print(my_dog.bark())  # Output: Woof!
```

# Information Hiding

Restrict access to internal details.

```
# Example: Using a private variable within a class
class BankAccount:
    def __init__(self, initial_balance):
        self.__balance = initial_balance  # Private variable

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance
```

# Interface Design

- Clear and consistent ways to interact.

```
# Example: A simple function with clear parameters
def calculate_area(length, width):
    '''Calculates the area of a rectangle.'''
    return length * width

area = calculate_area(5, 10)  # Clear usage of the function
```

# Design Patterns

## proven solutions to common problems.

```
# Example: Using the Factory pattern to create different shapes
class Shape:
    def draw(self):
        pass
    class Circle(Shape):
    def draw(self):
        print('Drawing a circle')
    ...
    def shape_factory(shape_type):
    if shape_type == 'circle':
        return Circle()
    elif shape_type == 'square':
        return Square()
circle = shape_factory('circle')
circle.draw()  # Output: Drawing a circle
```