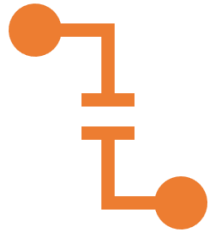




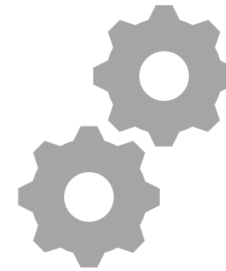
# Cohension and Coupling

# Modules



## What is a module?

lexically contiguous sequence of program statements, bounded by boundary elements, with aggregate identifier

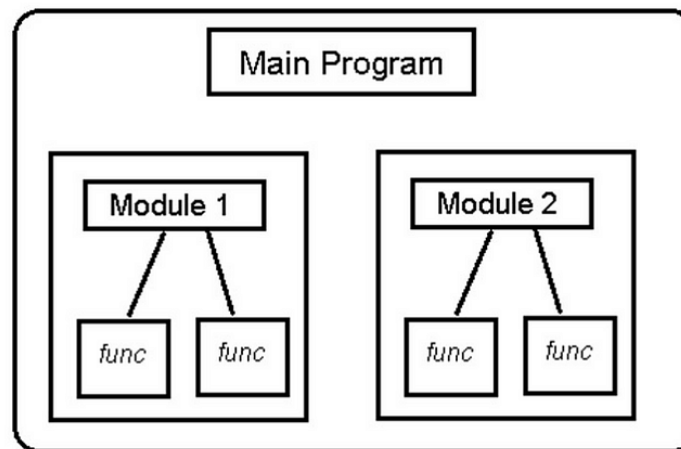


## Examples

procedures & functions in classical PLs  
objects & methods within objects in OO PLs

# Modular programming

Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable **modules**, such that each contains everything necessary to execute only one aspect of the desired functionality. This means that if you are not writing the new Star Wars Movie script, and if you are developing software, you do not write the whole software script. By dividing the whole concept into smaller parts, modules, you deal with them separately and construct the communication of these modules with each other while creating the whole.



# Modularization

- The main purpose of **modularization** is to make the software project simpler and more manageable and scalable. Let's modulate the benefits:
- Your code will be easier to **read**.
- You will **test** your code easily.
- In fact, since we also categorize the responsibilities in modularization, you will be able to easily **access** a code snippet or function that you are looking for.
- **Reusability**. You will not need to write the same code again and again.
- If a function is problematic, then you will just **fix** that specific function. You do not need to visit all parts of the software which are using this function.
- It will be very easier to **refactor** your code. Besides, it will be easier to work together with **team** members. Each person can work on a different module, etc.

# Coupling and cohesion

- **Coupling and cohesion** are two higher-level terms that we should consider when constructing the inter-module and intra-module structure in a software design. They give information about how easily your code can be changed or extended.

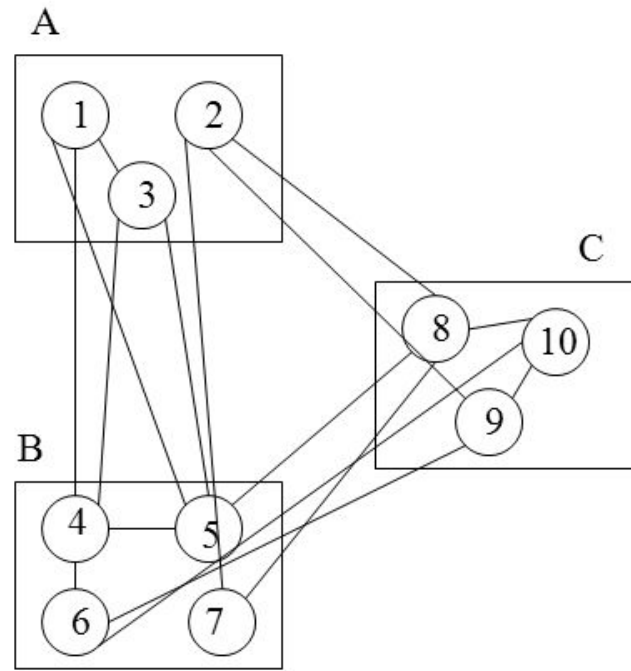
# Coupling and cohesion

- **Coupling** – the extent to which two components depend on each other for successful execution
  - Low coupling is good

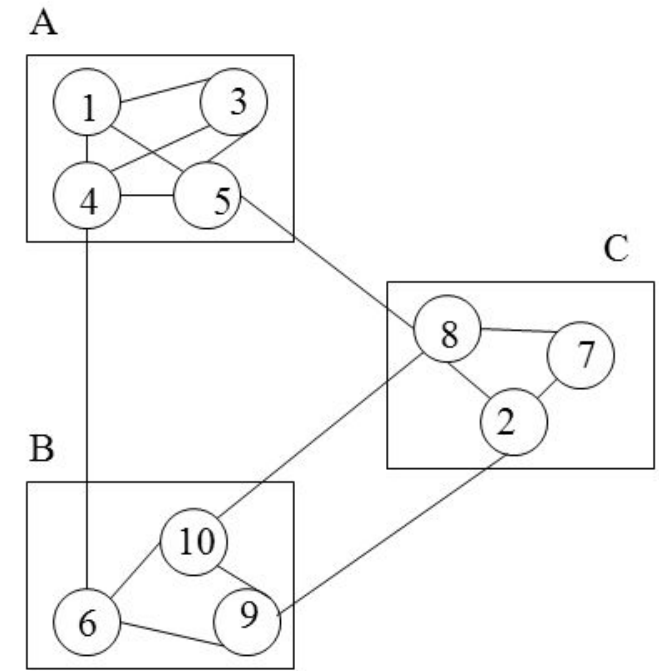


- **Cohesion** – the extent to which a component has a single purpose or function
  - High cohesion is good

# Good vs. Bad Modules



Bad modularization:  
low cohesion, high coupling



Good modularization:  
high cohesion, low coupling

# Good vs. Bad Modules

- Two designs functionally equivalent, but the 1<sup>st</sup> is
  - hard to understand
  - hard to locate faults
  - difficult to extend or enhance
  - cannot be reused in another product. Implication?
  - -> expensive to perform maintenance
- Good modules must be like the 2<sup>nd</sup> design
  - maximal relationships within modules (cohesion)
  - minimal relationships between modules (coupling)
  - this is the main contribution of structured design



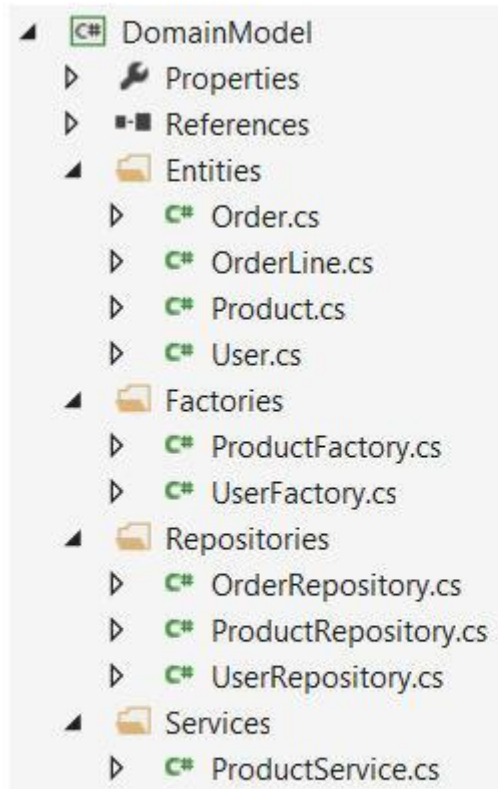
# Modules interactions

- When a large software is decomposed into smaller modules, it's inevitable that these modules will interact with one another.
- If the boundaries of these modules have been poorly identified, then the modules will heavily depend and frequently interact with one another.
- In a poor design, it might also happen that classes and methods within a module perform diverse tasks and therefore don't seem to belong together.

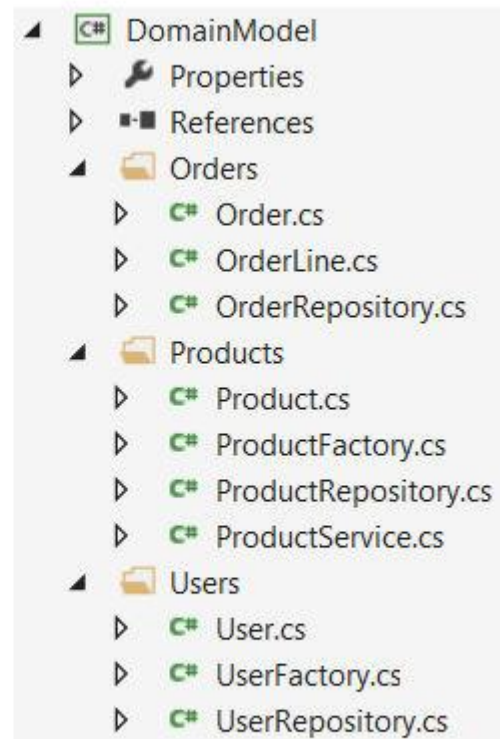
# Cohesion and Coupling

- **Cohesion** is about how well elements within a module belong together and serve a common purpose.
- **Coupling** is about how much one module depends or interacts with other modules.
- Thus, cohesion is an intra-module concern whereas coupling cuts across modules.
- To manage the complexity of an application, a software designer must find the right balance of cohesion and coupling.
- This is relevant to object-oriented design, the design of APIs and microservices.

# Could you explain cohesion and coupling with an example?



(a) Low cohesion, high coupling



(b) High cohesion, low coupling

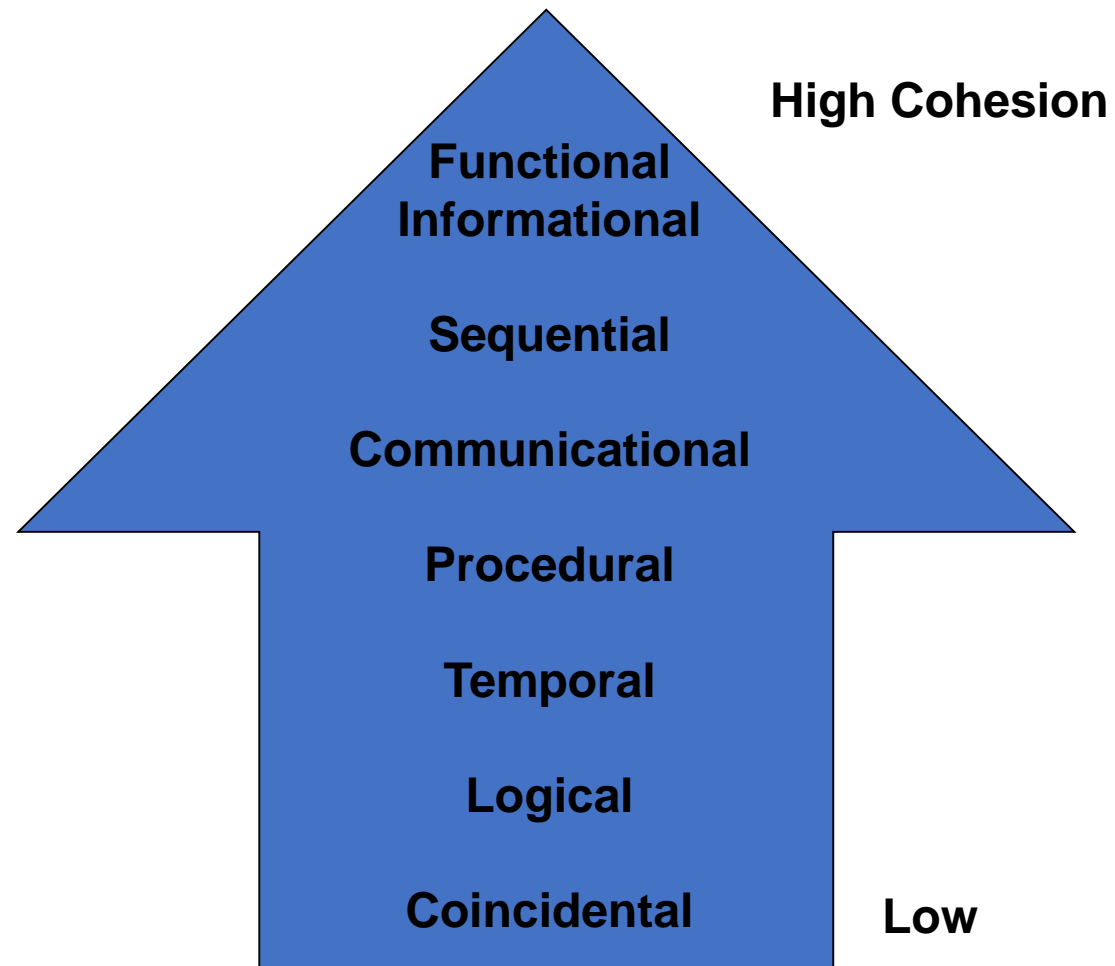
A good organization of s/w project

Low cohesion means the module is trying to do too many things.

High coupling means modules are tightly interconnected via many complex interfaces and information flows.

# Types of cohesion

Parts of a module that collectively perform a single well-defined function is the strongest type of cohesion.



# Coincidental cohesion

Utility class is a class that defines a set of methods that perform common, often re-used functions.

No conceptual relationship between elements. It is accidental and the worst form of cohesion. This one is very common in utility classes. For example, in the class below, there are unrelated methods together.

```
from abc import ABC, abstractmethod

class Utility(ABC):

    @abstractmethod
    def check_currency(self):
        pass

    @abstractmethod
    def parse_user_entry(self):
        pass

    @abstractmethod
    def convert_list_to_string(self):
        pass
```

# Coincidental Cohesion

- Def.
  - module performs multiple, completely unrelated actions
- Example
  - module prints next line, reverses the characters of the 2<sup>nd</sup> argument, adds 7 to 3<sup>rd</sup> argument
- How could this happen?
  - hard organizational rules about module size
- Why is this bad?
  - degrades maintainability & modules are not reusable
- Easy to fix. How?
  - break into separate modules each performing one task

# Logical cohesion:

They are classes that are brought together by thinking that they are about the same concept, even though they are different in reality.

Better than coincidental, there is at least a low level of semantic relation.

All the methods in the below class are related to reading, but there is no semantic unity between them. Which object reads both the article and the database?

```
@abstractmethod
def read_paper(self):
    pass
```

```
@abstractmethod
def read_from_database(self):
    pass
```

```
@abstractmethod
def read_article(self):
    pass
```

```
@abstractmethod
def read_exams(self):
```

# Logical Cohesion

- Def.
  - module performs series of related actions, one of which is selected by calling module
- Why is this bad?
  - interface difficult to understand
  - code for more than one action may be intertwined
  - difficult to reuse



# Temporal cohesion

The elements are related by their timing involved. In a module connected with temporal cohesion, all the tasks must be executed in the same period. For example, there may be a situation where a method trying to perform a calculation catches the exception thrown when it fails to perform the calculation for some reason, logs the situation, generates the necessary warnings, and continue to calculate in other ways.

```
class Calculation(ABC):

    @abstractmethod
    def final_result(self):
        pass

    @abstractmethod
    def temperature_failure(self):
        pass

    @abstractmethod
    def log_temperature_failure(self):
        pass

    @abstractmethod
    def throw_exception_temperature_failure(self):
        pass

    @abstractmethod
    def warn_user_temperature_failure(self):
        pass

    @abstractmethod
    def final_result_temperature_failure_case(self):
        pass
```

# Temporal Cohesion

- Def.
  - module performs series of actions related in time
- Initialization example

```
open old db, new db, transaction db, print db, initialize
sales district table, read first transaction record, read
first old db record
```
- Why is this bad?
  - actions weakly related to one another, but strongly related to actions in other modules
  - code spread out -> not maintainable or reusable
- Initialization example fix
  - define these initializers in the proper modules & then have an initialization module call each

# Procedural cohesion:

---

It is the functional separation of works related to a subject from top to bottom and bringing them all together in a class.

```
def read_input(self):  
    pass
```

```
@abstractmethod  
def validate(self):  
    pass
```

```
@abstractmethod  
def convert_units(self):  
    pass
```

```
@abstractmethod  
def calculate_temperature(self):  
    pass
```

```
@abstractmethod  
def decide_state(self,temp):
```

# Procedural Cohesion

- Def.
  - module performs series of actions related by procedure to be followed by product
- Example
  - update part number and update repair record in master db
- Why is this bad?
  - actions are still weakly related to one another
  - not reusable
- Solution
  - break up!

# Sequential cohesion:

They are classes that combine functions that work in a way that the output of one feeds the other at the class level.

python

Copy code

```
def get_average(numbers):  
    total = 0  
    for num in numbers:  
        total += num  
    average = total / len(numbers)  
    return average
```

This function calculates the average of a list of numbers. It exhibits sequential cohesion because the statements are arranged in a logical sequence, with the initialization of the total variable performed first, followed by the looping and the calculation of the average. Each statement depends on the previous statement to produce a meaningful result.

```
@abstractmethod  
def read_file(self):  
    pass
```

```
@abstractmethod  
def parse(self):  
    pass
```

```
@abstractmethod  
def do_semantic_analysis(self):  
    pass
```

```
@abstractmethod  
def generate_intermediate_code(self):  
    pass
```

```
@abstractmethod  
def optimize_intermediate_code(self):  
    pass
```

```
@abstractmethod
```

# Functional cohesion:

**Ideal case. Every essential element for a single computation is contained in the component. A functional cohesion performs the task and functions. They are structures that are single, well-defined, and put together for as little as possible a task or responsibility.**

**As the details increase in the software development process, the classes in other types of cohesion tend to be larger and larger. In functional cohesion, it is tried to protect the focus by dividing, for example, when new add-on methods arrive, classes are divided by grouping again. Because as the detail increases, the definition of the job or responsibility changes, and each job is divided into different and smaller jobs.**

**In the example below, the Phone class is both connecting and sending data. We split this class into two to make it more cohesive. DataHandler handles data transfer and PhoneConnection is responsible for connections.**

```
class Phone(ABC):

    @abstractmethod
    def dial(self):
        pass

    @abstractmethod
    def hangup(self):
        pass

    @abstractmethod
    def send(self):
        pass

    @abstractmethod
    def receive(self):
        pass

class DataHandler(ABC):

    @abstractmethod
    def send(self):
        pass

    @abstractmethod
    def receive(self):
        pass

class PhoneConnection(ABC):

    @abstractmethod
    def dial(self):
        pass


    @abstractmethod
    def hangup(self):
        pass
```



# Communicational Cohesion

- Def.
  - module performs series of actions related by procedure to be followed by product, but in addition all the actions operate on same data
  - communicational cohesion is about how the parts of a module or function work together to achieve a common goal, while sequential cohesion is about the order in which those parts are executed to produce the desired result.
- Why is this bad?
  - still leads to less reusability -> break it up

python

 Copy code

```
def calculate_total_cost(price, quantity):  
    total_cost = price * quantity  
    return total_cost
```

This function calculates the total cost of a purchase given the price and quantity of the item being purchased. It exhibits communicational cohesion because its parameters (i.e., price and quantity) are related to each other and are used together to compute the total cost. The function does not perform any additional tasks beyond this calculation, but it communicates information about the purchase (i.e., the price and quantity) in order to do so.

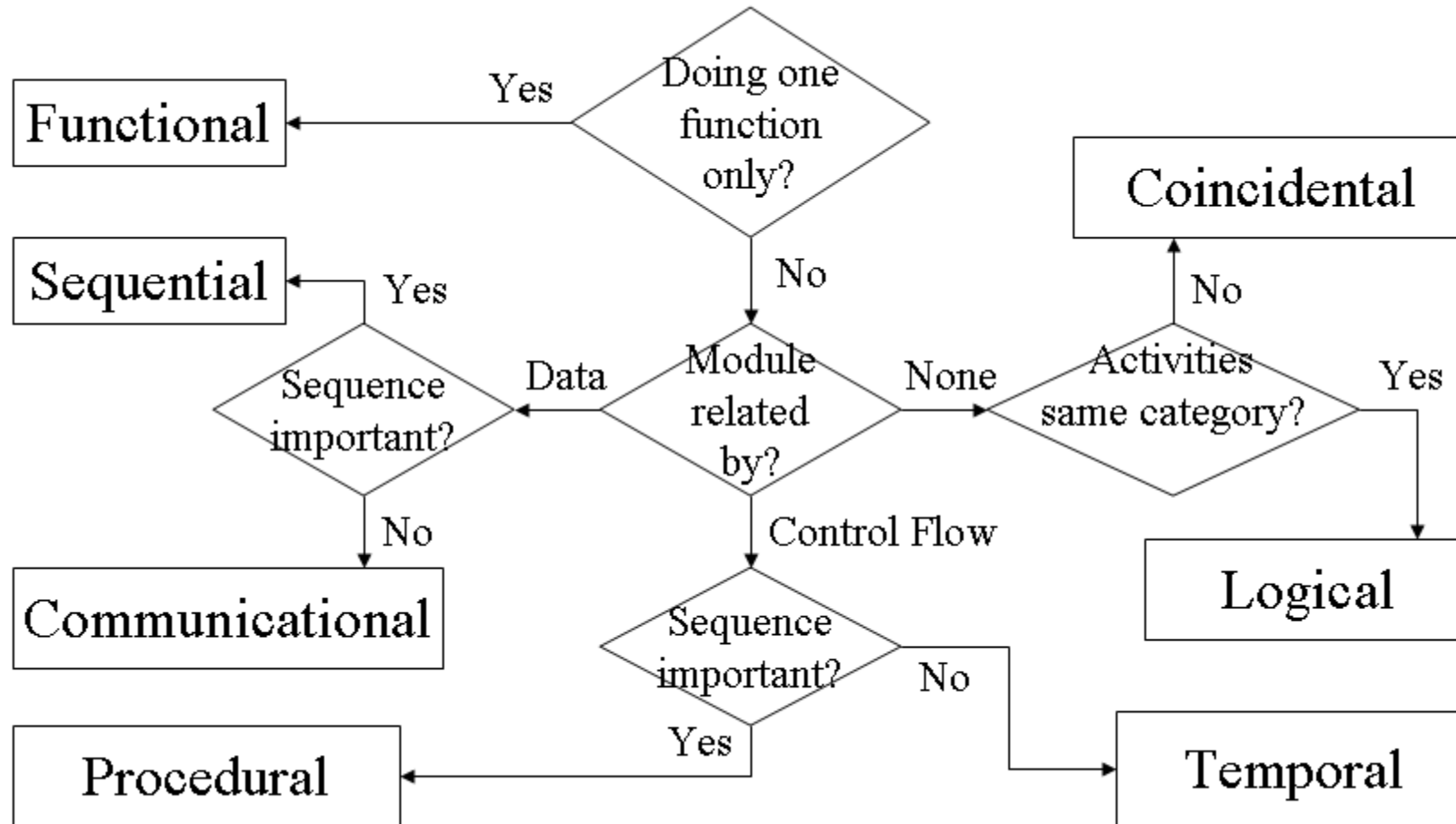
2/14/2001

# Functional Cohesion

- Def.
  - module performs exactly one action
- Examples
  - `get temperature of furnace`
  - `compute orbital of electron`
  - `calculate sales commission`
- Why is this good?
  - more reusable
  - corrective maintenance easier
    - fault isolation
    - reduced regression faults
  - easier to extend product



# Types of cohesion



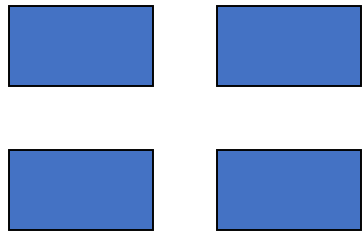
# Types of cohesion

- **Coincidental cohesion:** Parts are not related and just happen to be in the same module. Such a module is hard to understand, maintain or reuse.
- **Logical cohesion:** Parts relate to different entities though they perform similar logical functions. For example, mouse inputs and keyboard inputs are handled in the same class.
- **Temporal cohesion:** Parts that are executed when something happens, such as start-up or clean-up routines. Code changes may affect many modules.
- **Procedural cohesion:** Code that's called one after another.

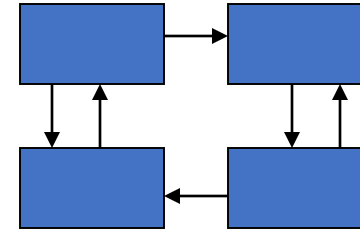
# Types of cohesion

- **Sequential cohesion:** Similar to procedural cohesion with the additional constraint that the execution sequence is important. For example, call `readFile()` before calling `processData()`.
- **Communicational cohesion:** Also called information cohesion. Parts that share or operate on the same data.
- **Functional cohesion:** The most desirable type of cohesion. Parts work together in fulfilling a single function or purpose.

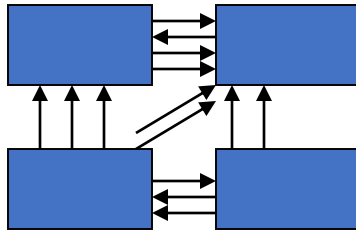
# Coupling: Degree of dependence among components



No dependencies



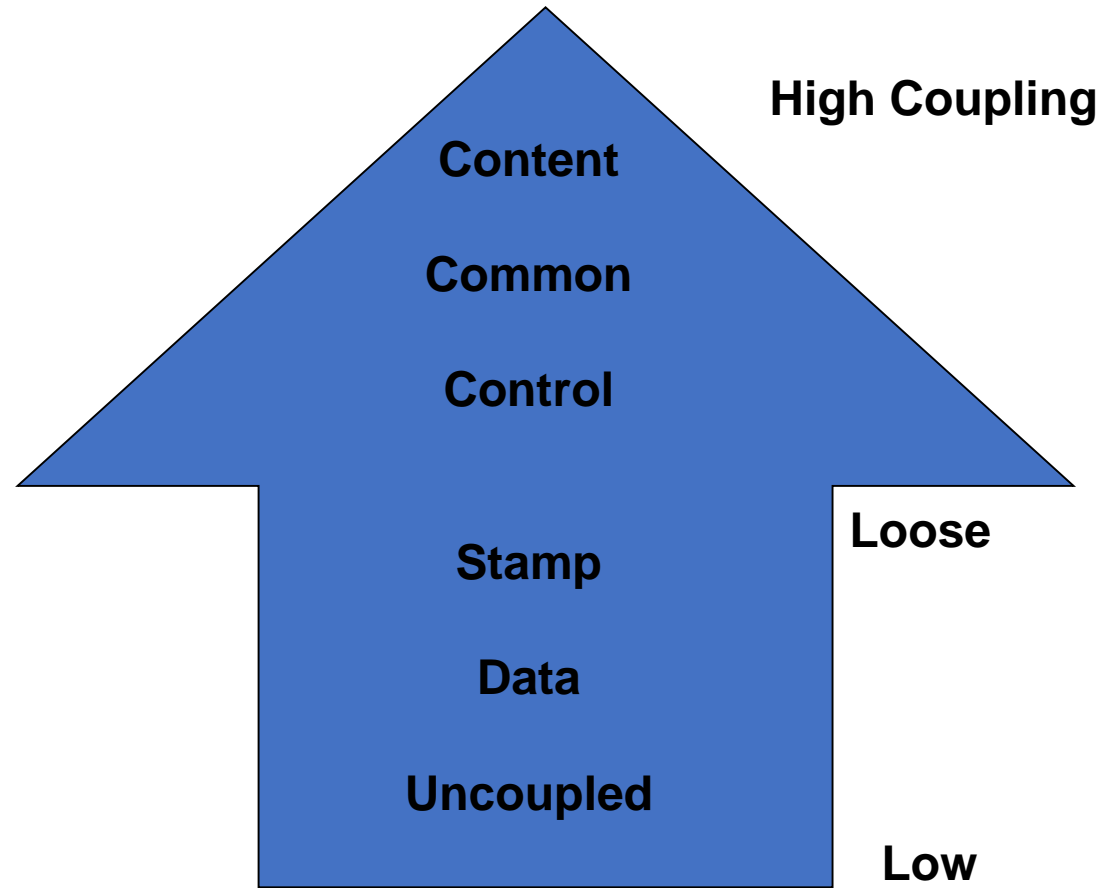
Loosely coupled-some dependencies



Highly coupled-many dependencies

High coupling makes modifying parts of the system difficult, e.g., modifying a component affects all the components to which the component is connected.

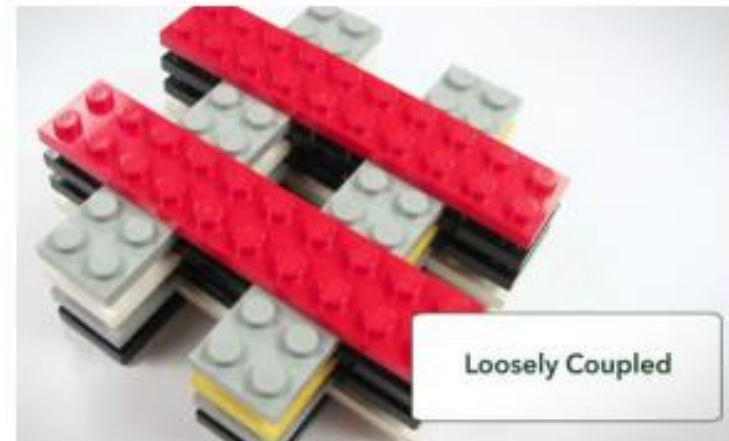
# Type of coupling



# Coupling

- Data and control are two types of information flow across modules.
- If only some data is exchanged, the amount of coupling is probably acceptable.
- Exchanging lots of data and control results in a high degree of coupling.

# Coupling

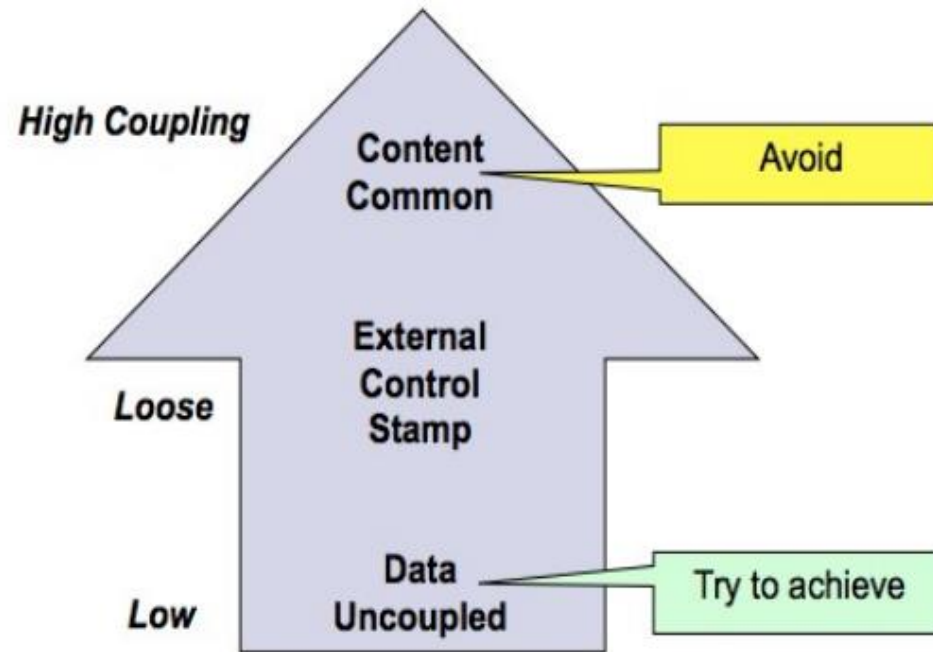


# How to evaluate coupling of a module?

- Degree :
  - Number of connections between the module and others
- Ease:
  - How easy is to connect to other modules.
- Flexibility
  - How interchangeable are other modules for this module.

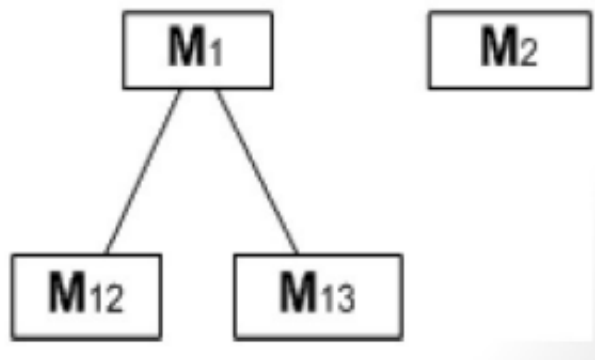


# Types of Coupling



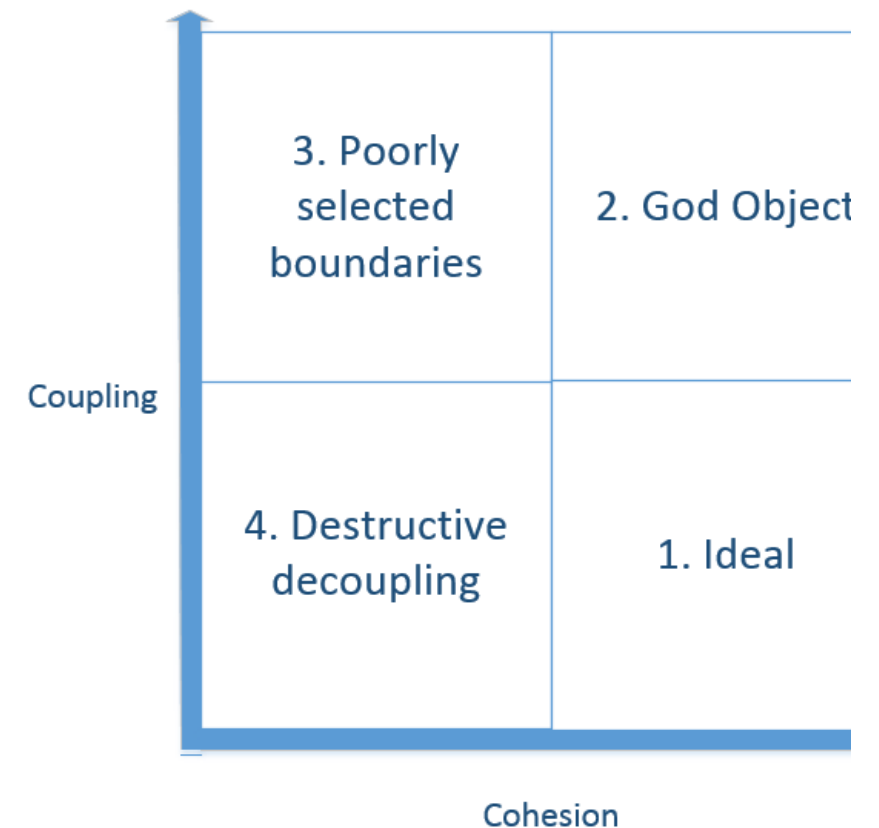
# Uncoupled

- Completely uncoupled systems are not systems
- Systems are made of **interacting components**



# Balanced cohesion and coupling

- Ideally, we should aim for **high cohesion and low coupling**.
- Related parts of code should be in the same module.
- Each module should be independent of others. Modules should interact in a minimal way.
- Apply the **single responsibility principle** for high cohesion
- Use **well-defined interfaces** and inversion of control for low-coupling
- An anti-pattern is when a piece of code does all the work.

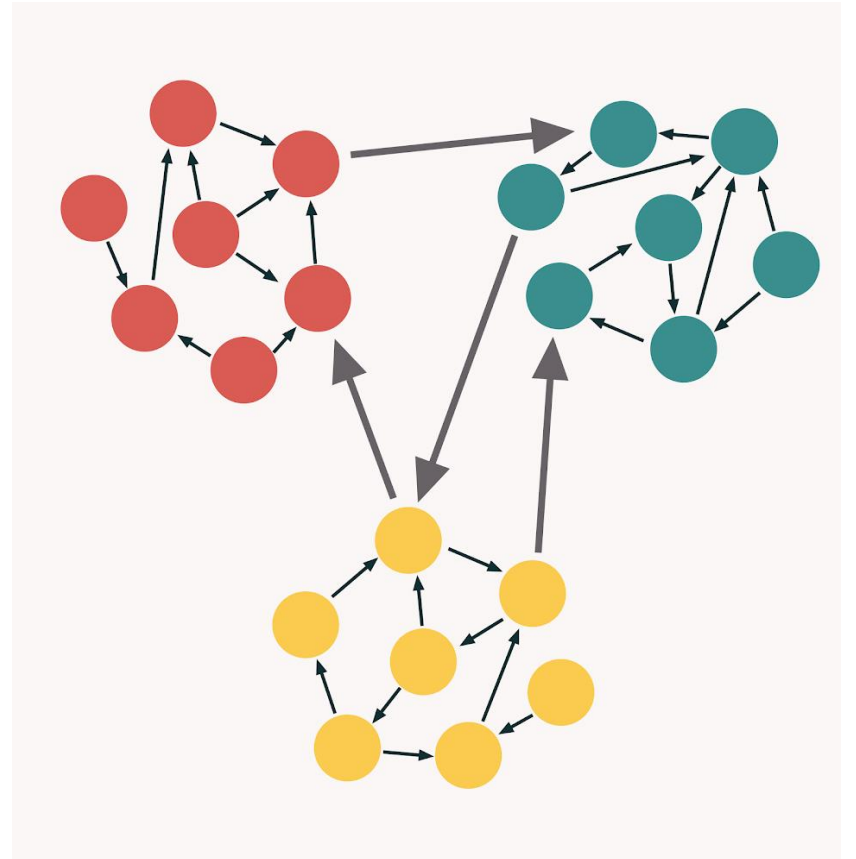


# Ideal

---

Ideal is the code that follows the guideline. It is loosely coupled and highly cohesive. We can illustrate such code with this picture:

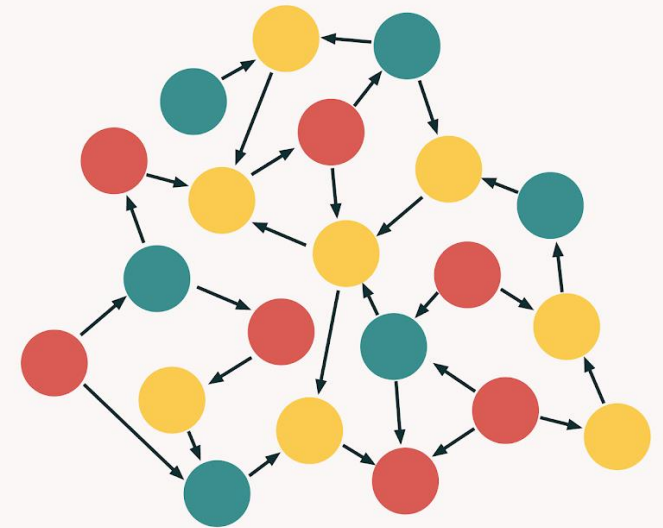
- Here, circles of the same color represent pieces of the code base related to each other.
- Example: A system that has a module for handling orders and a module for handling payments. The two modules are tightly coupled, with the order module calling the payment module to process payments. If the payment module is refactored without considering its dependencies on the order module, it may break the functionality of the order module, causing errors or reducing functionality.



# God Object

---

2God Object is a result of introducing high cohesion and high coupling. It is an anti-pattern and basically stands for a single piece of code that does all the work at once:

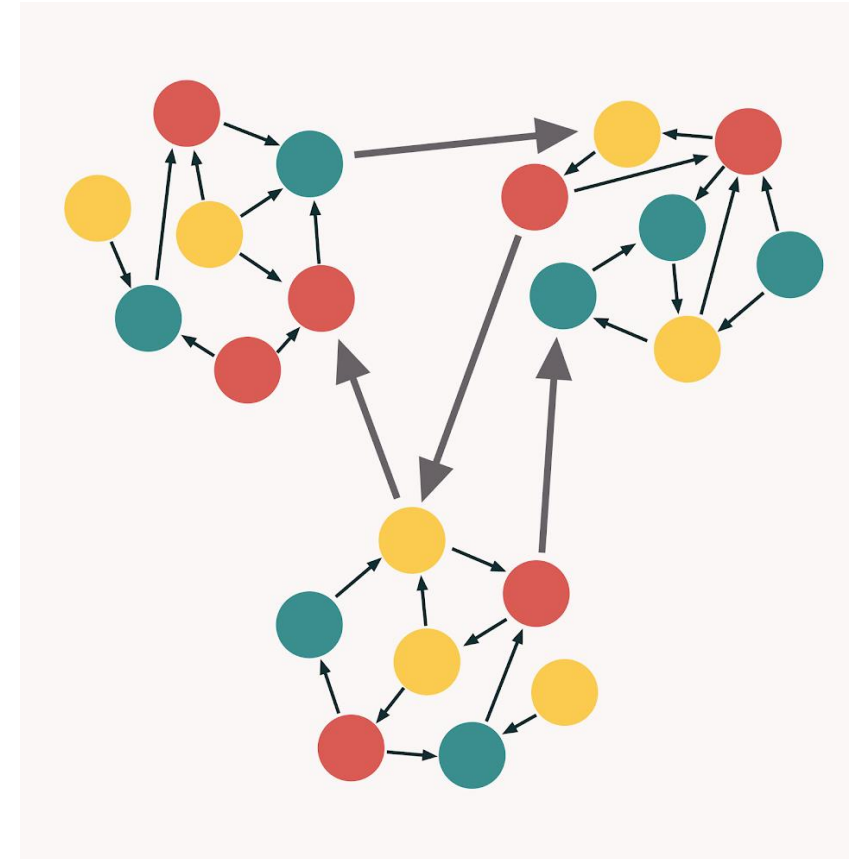


# Poorly Selected Boundaries:

---

The third type takes place when the boundaries between different classes or modules are selected poorly. Poorly selected boundaries occur when the boundaries between objects or modules are unclear or inconsistent. This can lead to confusion, errors, and a lack of clarity in the system design.

- Example: A system that has multiple modules for handling customer information, but each module uses a different database schema and API. This can lead to confusion and errors when modules try to access or modify customer information, and can make it difficult to maintain or modify the system.
- Unlike God Object, code of this type does have boundaries. The problem here is that they are selected improperly and often do not reflect the actual semantics of the domain. Such code quite often violates the Single Responsibility Principle.



# Destructive Decoupling

- 4. Destructive decoupling is the most interesting one. It sometimes occurs when a programmer tries to decouple a code base so much that the code completely loses its focus:
- Destructive decoupling occurs when two or more objects or modules that are tightly coupled are decoupled in a way that causes errors or reduces functionality. This can occur when modules or objects are refactored without considering the dependencies between them.
- Example: A system that has a module for handling orders and a module for handling payments. The two modules are tightly coupled, with the order module calling the payment module to process payments. If the payment module is refactored without considering its dependencies on the order module, it may break the functionality of the order module, causing errors or reducing functionality.

