COSC 310: SOFTWARE ARCHITECTURE.

Dr. Gema Rodriguez-Perez
University of British Columbia
gema.rodriguezperez@ubc.ca

ARCHITECTURAL DESIGN

SOFTWARE DESIGN

- The ***design process*** determines *how* to construct a software system to meet the system requirements.

- Design involves determining:
  - the system architecture
  - the modules, classes, methods, and interfaces
  - the data structures and algorithms used
  - the interaction protocols/interfaces with users and other systems
  - implementation language

ARCHITECTURAL DESIGN AND AGILITY

Architectural design is concerned with understanding how a software system should be organized and designing the overall structure of that system

- Critical link between design and requirements engineering, as it identifies the main structural components in a system and the relationships between them.
- The output of the architectural design process is an architectural model that describes how the system is organized as a set of communicating components.
- An early stage of agile processes is to design an overall systems architecture.

ARCHITECTURAL DESIGN ISSUES

While Individual components implement the functional system requirements, the dominant influence on the **non-functional system characteristic** is the system's architecture.

- *Users*: How many simultaneous users must the system support? Scalable?
- *Distribution*: Are the users, data, or system components physically distributed?
- *Performance*: Are there stringent real-time or interactive performance requirements that must be satisfied?
- *Maintainability*: How will the software be maintained?
- *Security*: Must the software need to handle private data?

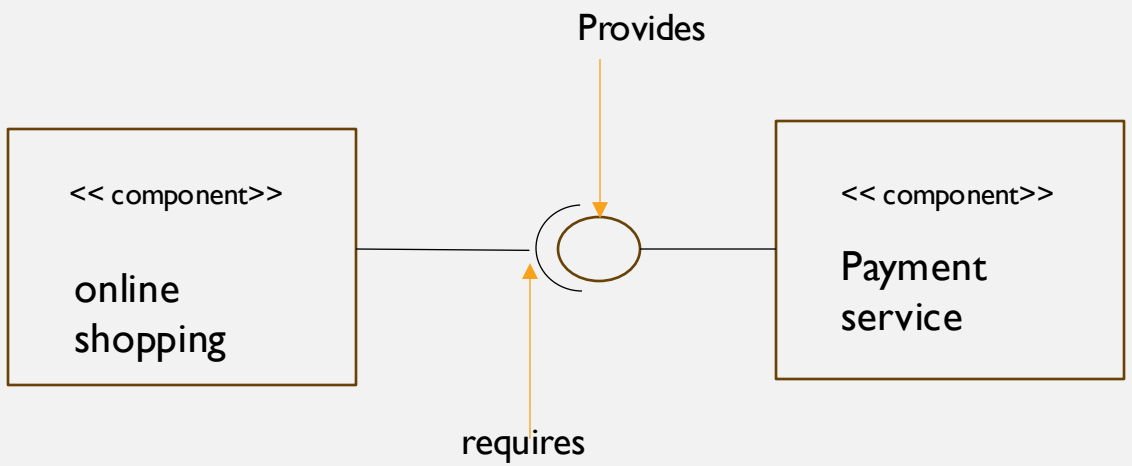
Where to start?? Systems in the same domain often have similar architectures that reflect domain concepts

HOW TO REPRESENT THE ARCHITECTURE?

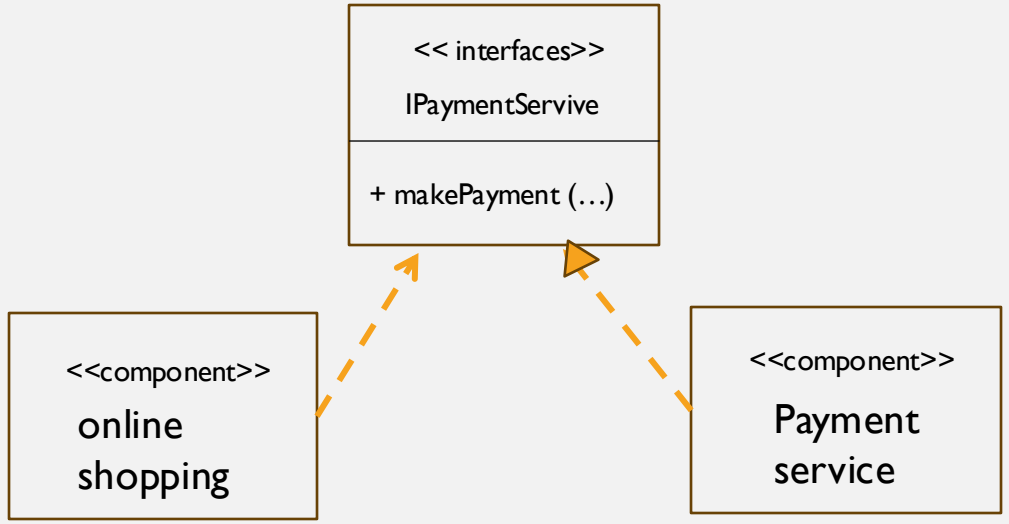
- UML Component diagram – It's a high level view of component and interfaces.
- What it is a component? – It's a piece of your system, a "unit of composition"
  - It contributes to the overall functionality of the system, essentially acting as an individual element within a larger structure
    - For example, in a web service, the online shopping is a component that depends on another component, the payment system. But both systems contributes to the overall functionality of the system.
  - A component can have explicit dependencies

UML COMPONENT DIAGRAM

- UML syntax



- Basic



- Detailed

ARCHITECTURAL PATTERNS

ARCHITECTURAL STYLES/PATTERNS

- They are good practices; they are generic styles that have been used a lot and have been proven to work.
- We know their benefits and limitations.
- The architecture of your system will have big implications on your quality requirements
- There is no perfect architecture, but some will help with safety, security, performance, availability, maintainability, etc.

ARCHITECTURE: MONOLITHIC

- **Monolithic design** an architecture in which all components of an application are tightly integrated and interdependent
  - Monolithic application is built as a single, unified unit, with a single codebase and a single deployment package
  - all components of the application are tightly coupled and communicate with each other directly
  - Any changes to one component of the application can potentially affect other components, making it more difficult to maintain and update the system
  - They can be useful for smaller applications or for teams that are just starting out and need to quickly build a prototype or proof-of-concept
    - However, as applications grow and become more complex, monolithic designs can become difficult to maintain and scale.

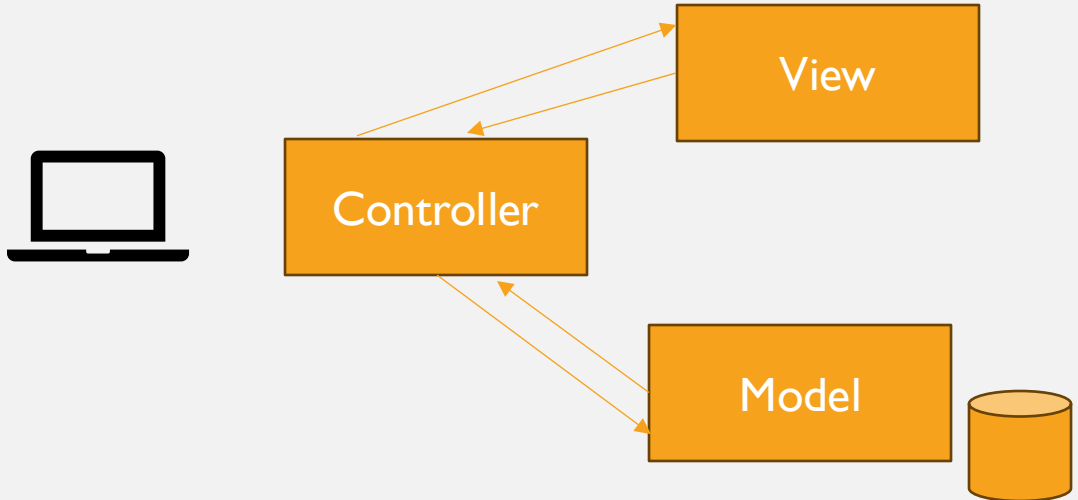
ARCHITECTURAL STYLES/PATTERNS

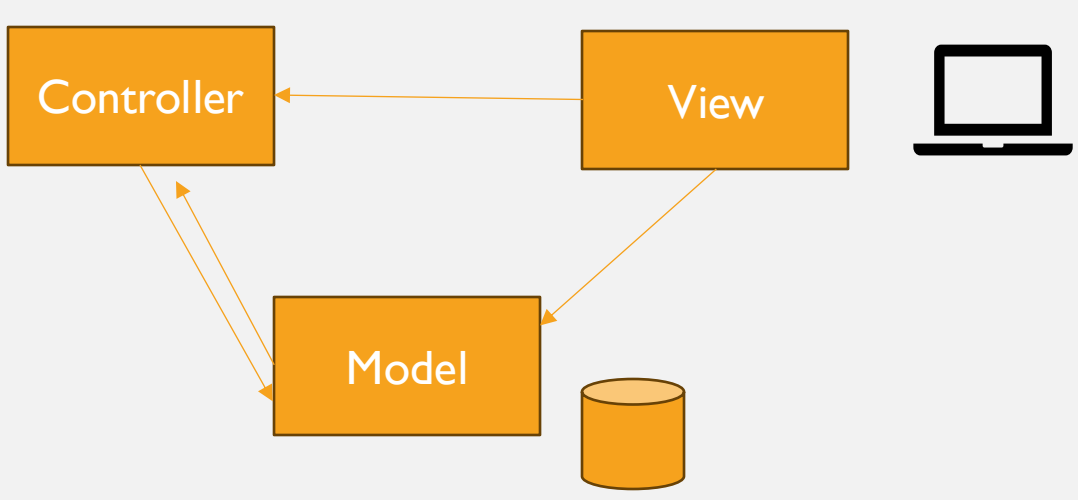
MVC (Model/View/Controller): The state of the data should be independent of the representation and the way to control it.

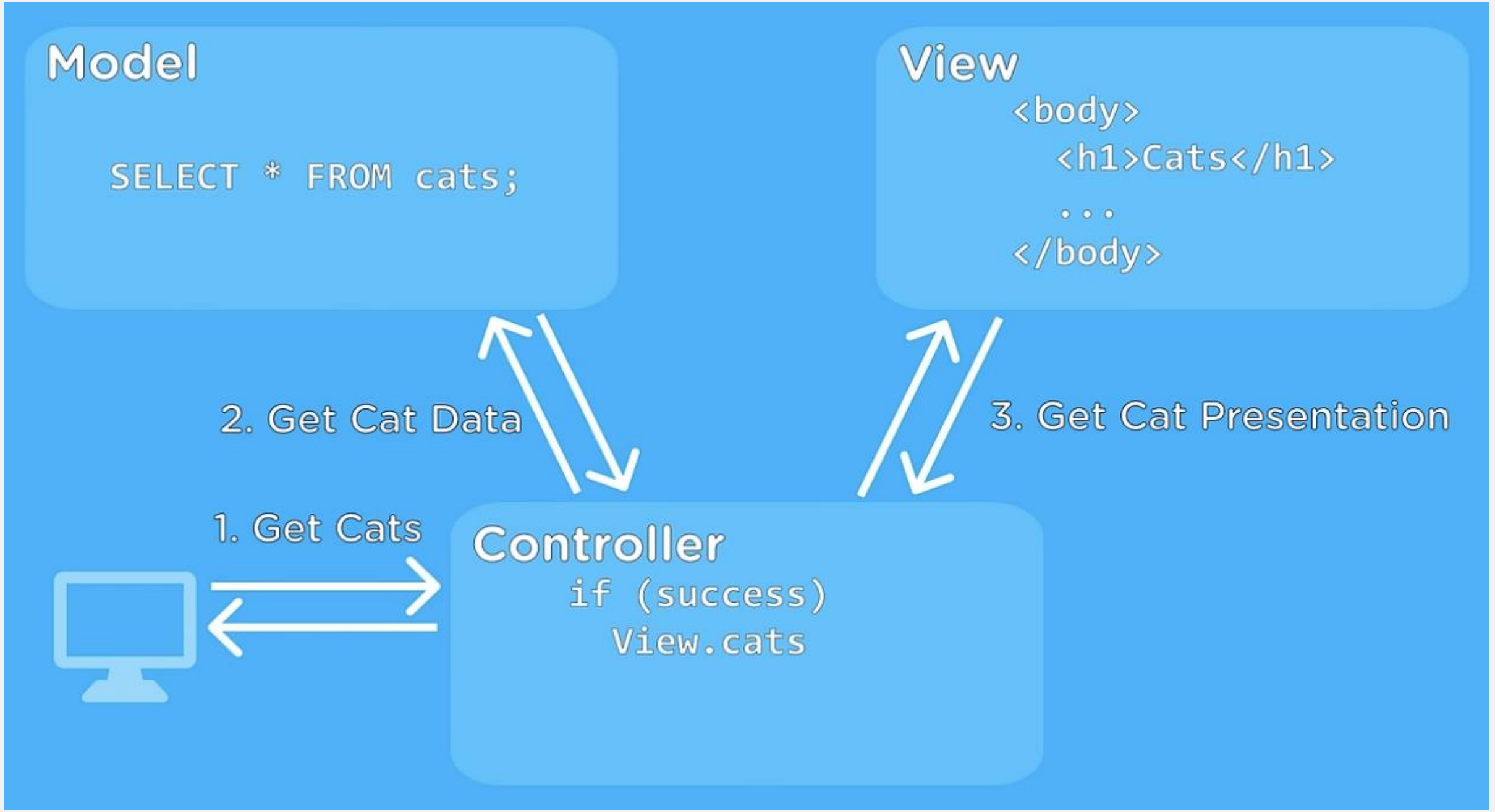
Model: Handle data logically so it basically deals with data.

View: Data Representation (UI)

Controller: Handles requests flow



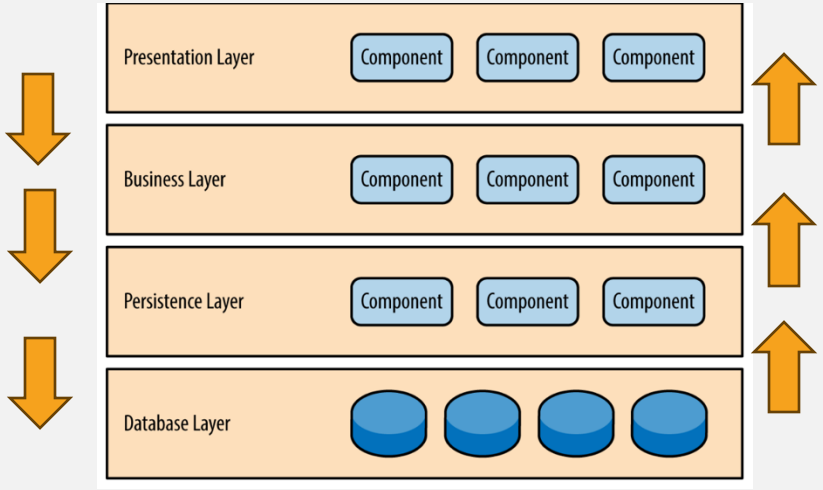



Model
SELECT * FROM cats;
View
<body>
<h1>Cats</h1>
...
</body>
2. Get Cat Data
3. Get Cat Presentation
1. Get Cats
Controller
if (success)
View.cats

ARCHITECTURAL STYLES/PATTERNS

Layered architecture: We have multiple layers, like Presentation (UI), Business logic, etc…

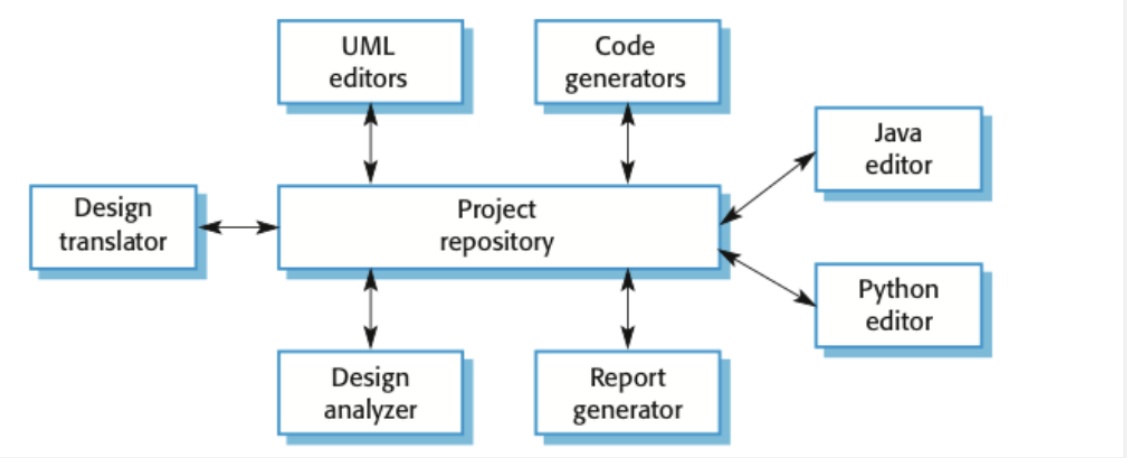
- The upper layers only talked to the layer below them. And the lower layers never called up, only reply. They are only reacting.
- Common why to organize teams.
- Advantages:
  - Security at several layers
  - Build on top
  - Replacement
- Limitations:
  - Performance
  - Difficult to implement in practice



ARCHITECTURAL STYLES/PATTERNS

Repository style: You have a central component (repository) and a number of component that access the repository.

- The repository contains all the data
- The other components do not talk to each other, they only access the main repo.
- Advantage:
  - Component do not have interfaces with other components
- Disadvantage
  - Central component have many interfaces
  - Failure of the central component



ARCHITECTURAL STYLES/PATTERNS

Client/Service architecture: You have some network and clients talk to the network and then servers send what the clients wants from the service.

It is distributed

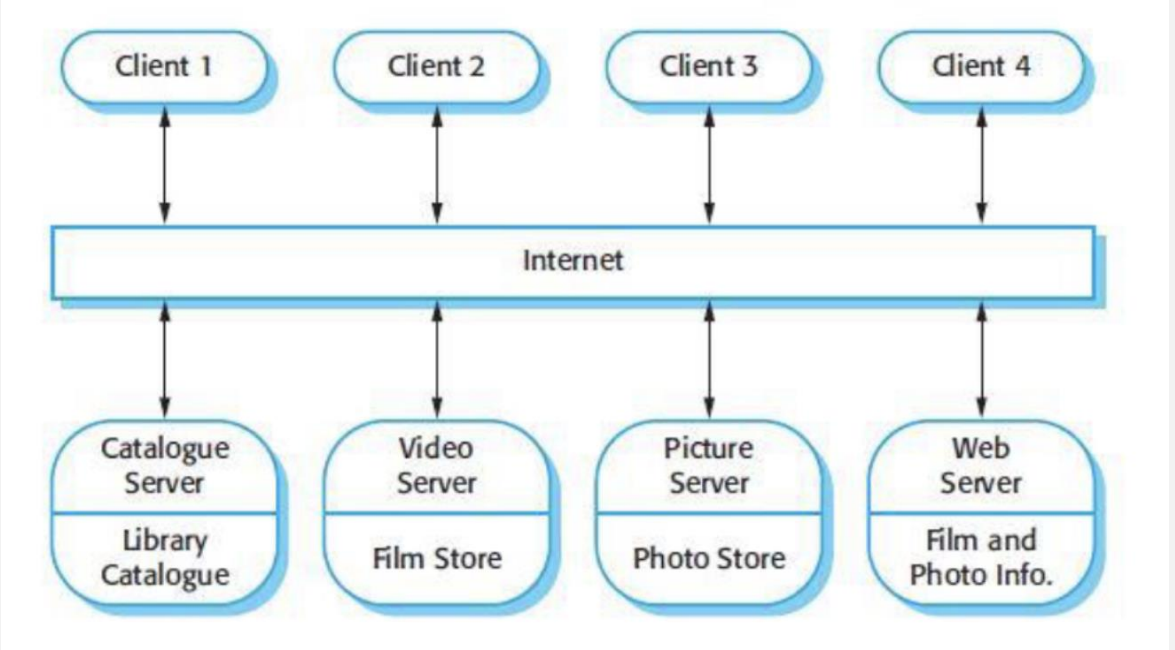
Advantages:

- Performance

Disadvantages:

- Prediction of performance is hard

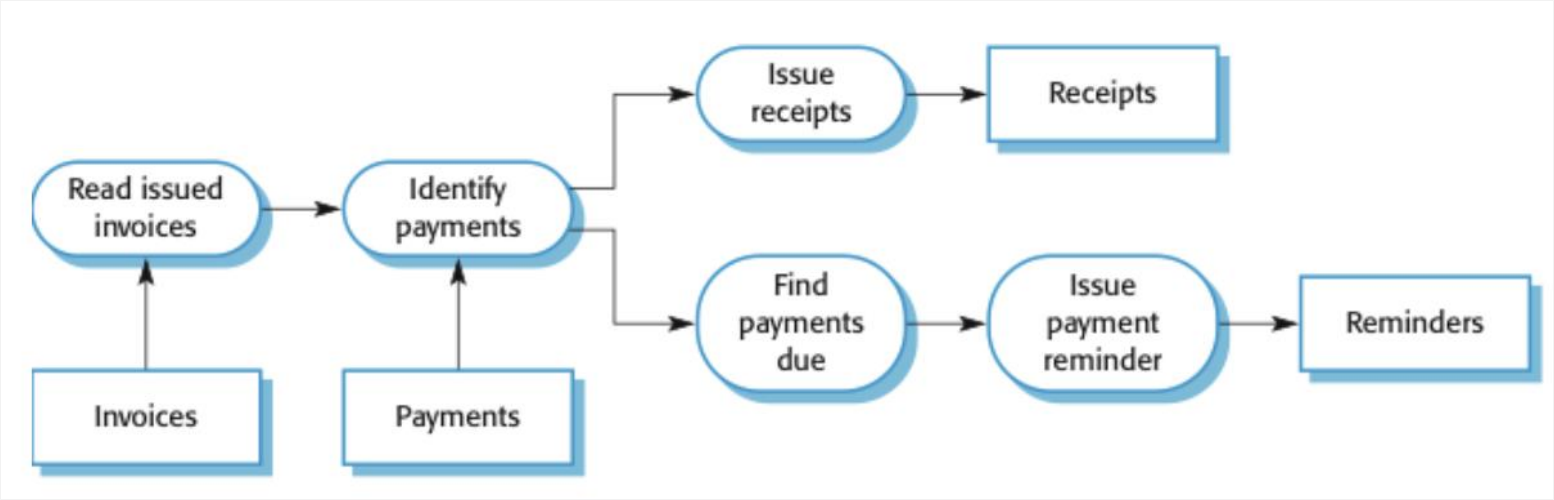
- Single point of failure



ARCHITECTURAL STYLES/PATTERNS

Pipe and Filter: Common for data processing applications (i.e, graphic processing, invoicing)

- There can be concurrent types
- There are transformation steps and the data flows
- Advantages:
  - Reuse
  - Workflows
  - Concurrency
  - Modification
- Disadvantages:
  - Agreed on the pipe



GENERIC APPLICATION ARCHITECTURES

- There are common architectures for similar business
- ERP (enterprise resource planning) system: easy to tailor to different business context. For example, Oracle, SAP
- Data processing applications
- Transactions processing
- Event processing applications
- Language processing applications

SERVICE-ORIENTED ARCHITECTURES (SOA)

- SOA are based around the notion of externally provided services.
  - Focuses on delivery of a given service using a communications protocol over the network
- A service has four properties according to one of many definitions of SOA
  - It logically represents a repeatable business activity with a specified outcome
  - It is self-contained
  - It is a black box for its consumers; consumer does not have to be aware of the service's inner workings
  - It may be composed of other services

ARCHITECTURE TYPES: SOA/MICROSERVICES

- **Microservices architecture:** modern approach to software architecture where the application is broken down into smaller, independent services that communicate with each other via lightweight mechanisms such as APIs
  - Each service is designed to perform a specific business function and can be developed, tested, and deployed independently of the others.
  - Each service is responsible for a specific task (i.e. authentication, payment processing)
    - Can be developed using different technologies and can be deployed independently, allowing teams to work on different parts of the application simultaneously
- Can introduce some complexities, such as managing communication between services, ensuring consistency across services, and monitoring and debugging distributed systems.
  - Choice to use microservices should be made based on the specific needs of the application and the organization.

ARCHITECTURE TYPES: SOA/MICROSERVICES - BENEFITS

- **Scalability** Microservices can be scaled independently of each other, allowing for greater flexibility and the ability to handle changes in traffic or workload.
- **Resilience** If one service fails, it does not necessarily affect the entire system, as other services can continue to function.
- **Flexibility** Teams can work on different parts of the application independently, allowing for faster development and deployment of new features.
- **Technology diversity** Different services can be developed using different programming languages or technologies, allowing teams to choose the best tool for the job.

CHALLENGES OF THE DESIGN PHASE

- The design team should not do too much.
  - The detailed design should not become code.
- The design team should not do too little.
  - It is essential for the design team to produce a complete detailed design.
- Good design requires experience in addition to education.
  - Experience is only learned through practice.
  - Best to learn good design practice from experienced designers while working in teams.

CONCLUSION

- A ***software architecture*** is the fundamental framework for structuring a system. It is critical during design to determine the correct architecture to satisfy the system requirements.
- Architectures have different performance, scalability, security, and availability. Complex architectures are not always better.

COSC 310: DESIGN PRINCIPLES

Dr. Gema Rodriguez-Perez
University of British Columbia
gema.rodriguezperez@ubc.ca

OBJECTIVES

- Understand how to create modular, maintainable and testable code using the SOLID design principles
- Explain the SOLID principles
- Understand when to apply the SOLID design principles
- Define and compare metrics for design: Cohesion and Coupling
- Understand the different levels of coupling and cohesion.

SOLID PRINCIPLES

PRINCIPLES AND HEURISTICS FOR MODULAR DESIGN

S O L I D (one nice combo of principles)

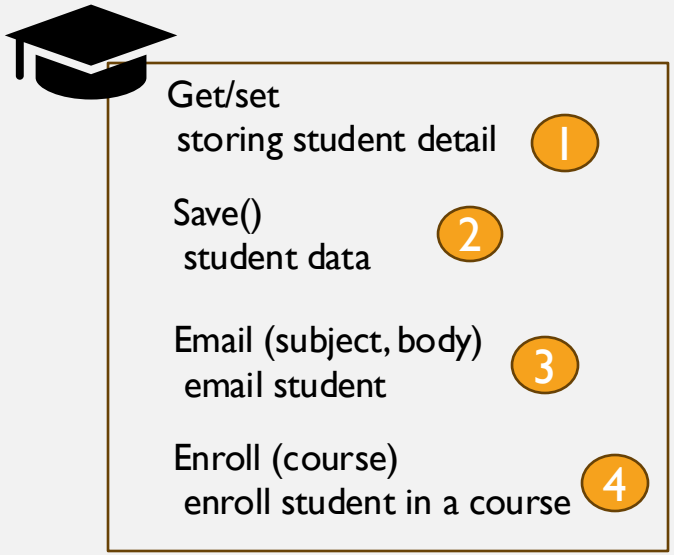
- Single Responsibility Principle (High Cohesion, Low Coupling)
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- A class should have only a single responsibility (i.e. only one potential change in the software's specification should be able to affect the specification of the class)
- "One class doing the work of two" 

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

- We are developing an application like Canvas. Where we have a student class.



Multiple reasons for changing the file

Potential merging conflicts if all dev are working on this file

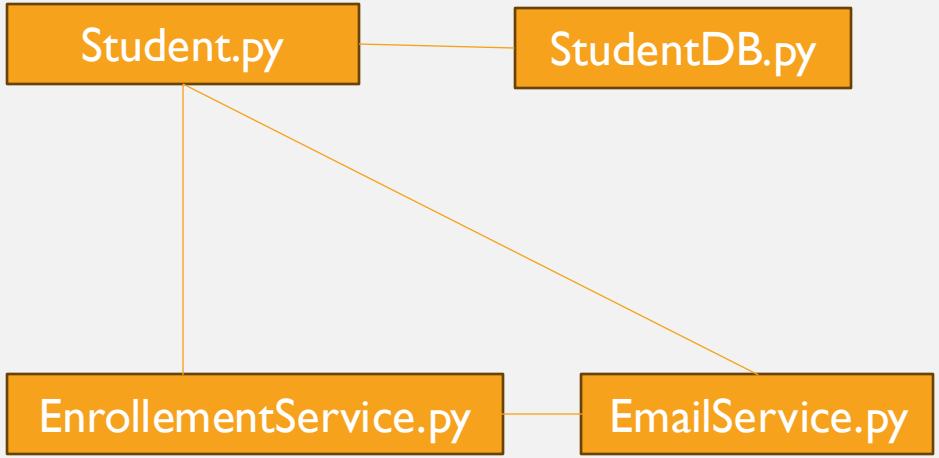
SINGLE RESPONSIBILITY PRINCIPLE (SRP)

Get/set
storing student detail

Save()
student data

Email (subject, body)
email student

Enroll (course)
enroll student in a course



SINGLE RESPONSIBILITY PRINCIPLE (SRP)

Single reasonability ≠ Single Job

Everything a class does should be closely related

SINGLE RESPONSIBILITY PRINCIPLE (SRP)

```python
# file_manager_srp.py

from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

1

2

```python
# file_manager_srp.py

from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)
```

```python
class ZipFileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

OPEN/CLOSED PRINCIPLE (OCP)

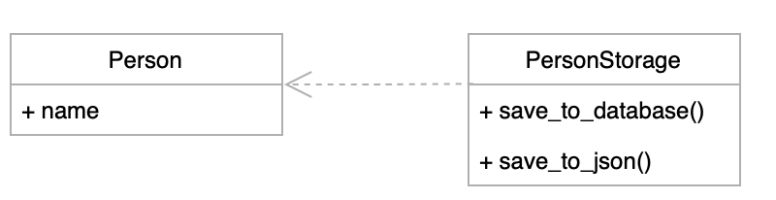
- A class must be closed for modifications, but must be open for extensions
- When designing classes, do not plan for brand new functionality to be added by modifying the core of the class.
- Remember that you have added unit tests to the core class functionality so modifying it will imply to modify all unit tests
- Instead, design your class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

OPEN/CLOSED PRINCIPLE (OCP)

How are we going to add new functionality without touching the class?

- Use the Decorator pattern (we will see this in the design patterns).
- Extension methods
- Inheritance or attributes

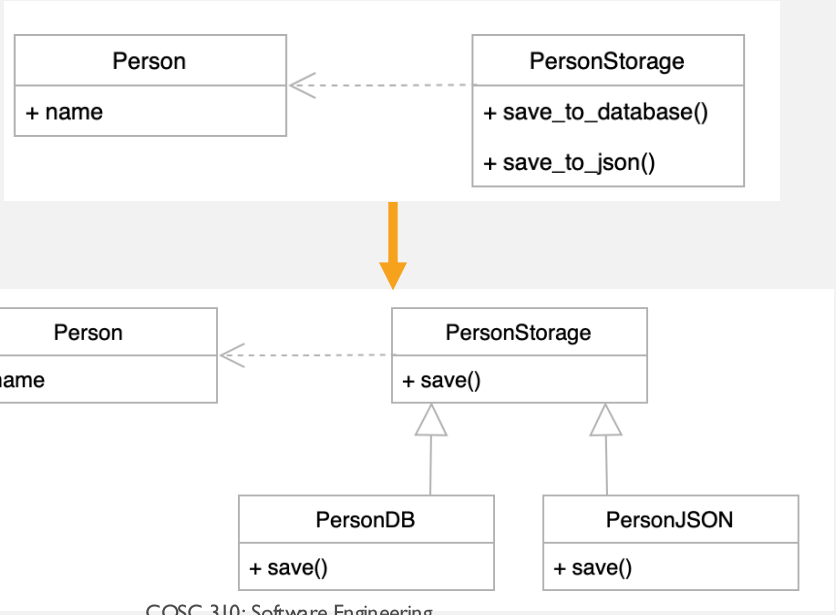
OPEN/CLOSED PRINCIPLE (OCP)



+ save_to_xml ()

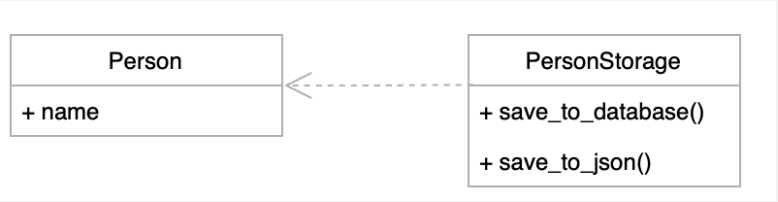
Person storage class is not open for extension, but modification

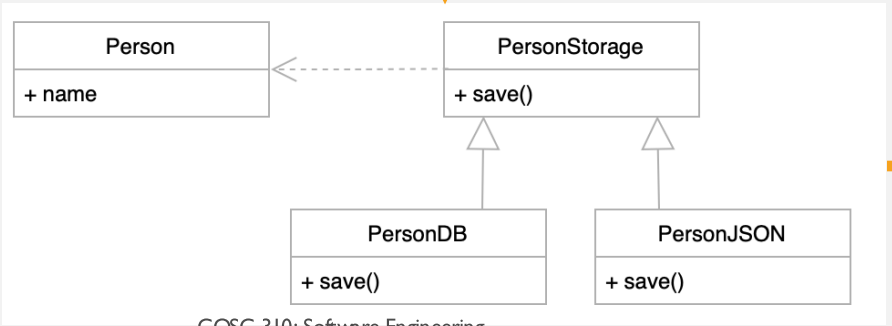
OPEN/CLOSED PRINCIPLE (OCP)



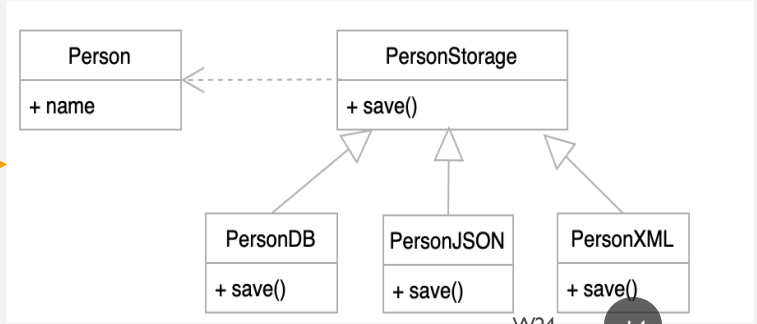
To make PersonStorage class to conform with the open-closed principle, we can use inheritance

OPEN/CLOSED PRINCIPLE (OCP)





Person storage class now conforms with the open-closed principle. I do not need to modify it when adding new functionality



LISKOV SUBSTITUTION PRINCIPLE (LSP)

Turing Award 2008

Introduced by Barbara Liskov at a conference in 1987. Since then, this principle has been a fundamental part of object-oriented programming, it says that:

- Subtypes must be substitutable for their base types.

- if S is a subtype of T, then objects of type T (supertype) in a program may be replaced with objects of type S (subtype) without altering any of the desirable properties of that program

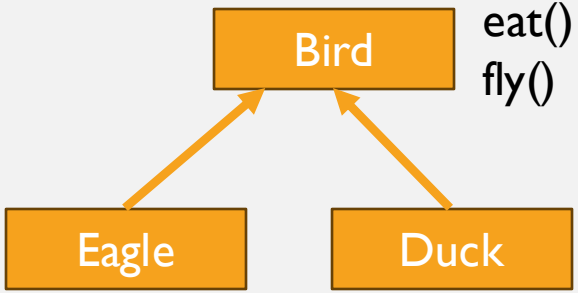
Your code can work but conceptually might be wrong!

LISKOV SUBSTITUTION PRINCIPLE (LSP)

- In other words, this principle states that a child class should be able to do everything that a parent class can.
- This is the expected behavior, because when we use inheritance we assume that the child class inherits everything that the superclass has. The child class extends the behavior but never narrows it down.
- Liskov's principle is easy to understand but hard to detect in code.

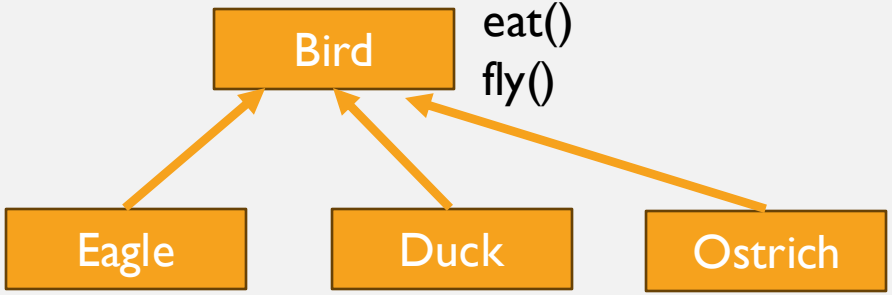
LISKOV SUBSTITUTION PRINCIPLE (LSP)

- A subclass should not break the expectations set by its superclass!



LISKOV SUBSTITUTION PRINCIPLE (LSP)

- A subclass should not break the expectations set by its superclass!

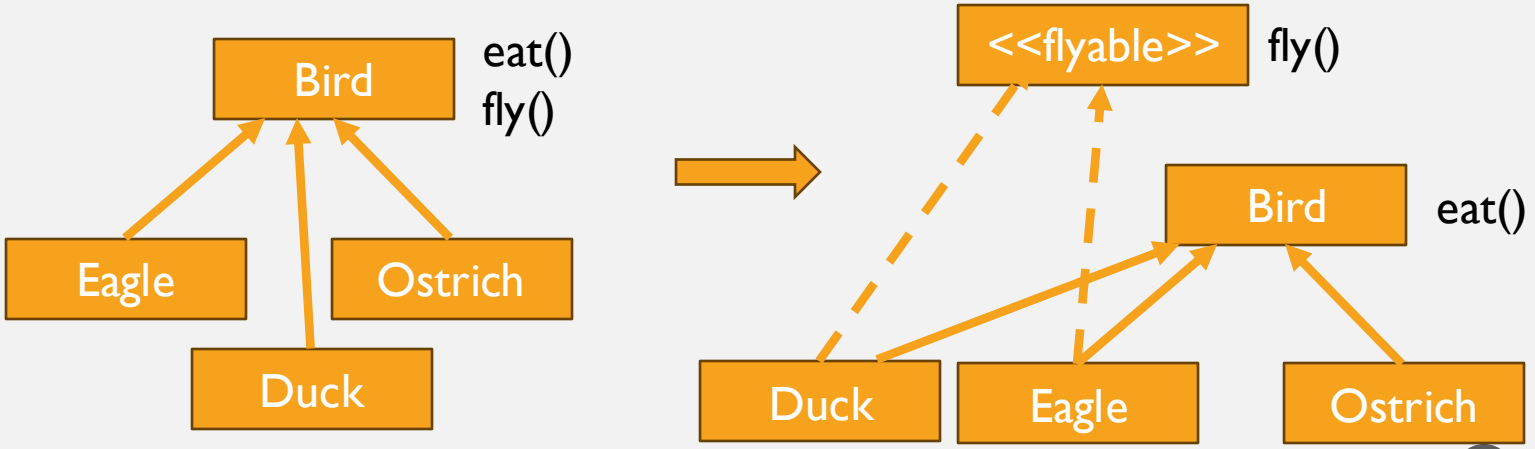



```
def makefly(bird):
    bird.fly()
```

?

LISKOV SUBSTITUTION PRINCIPLE (LSP)

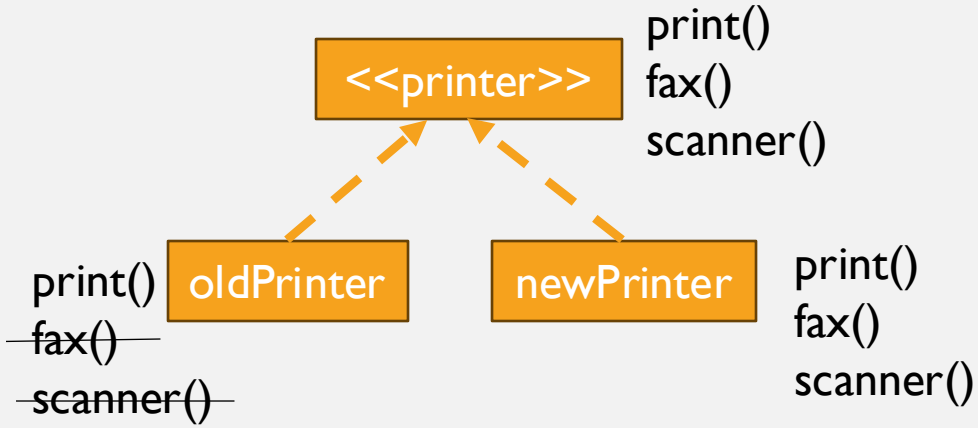
- We can solve it with an interface.



INTERFACE SEGREGATION PRINCIPLE (ISP)

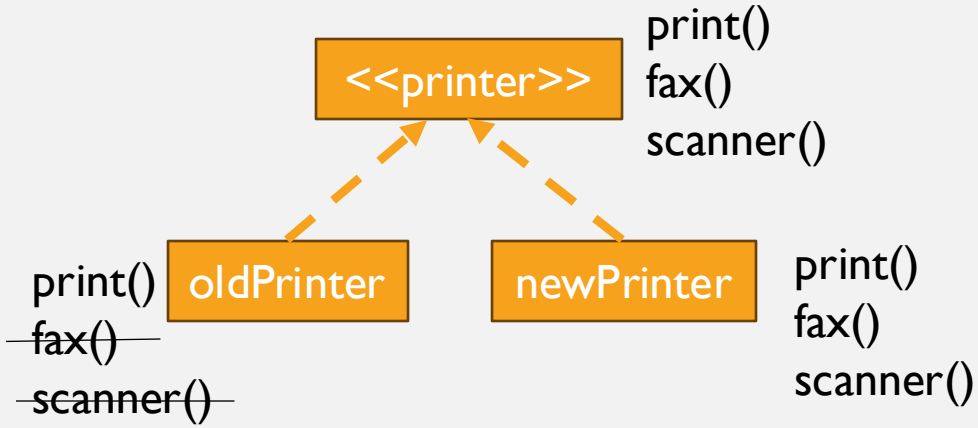
- Clients should not be forced to depend upon methods that they do not use. Interfaces belong to clients, not to hierarchies.
- If a class doesn't use particular methods or attributes, then those methods and attributes should be segregated into more specific classes.
- Many client-specific interfaces are better than one general-purpose interface.

INTERFACE SEGREGATION PRINCIPLE (ISP)



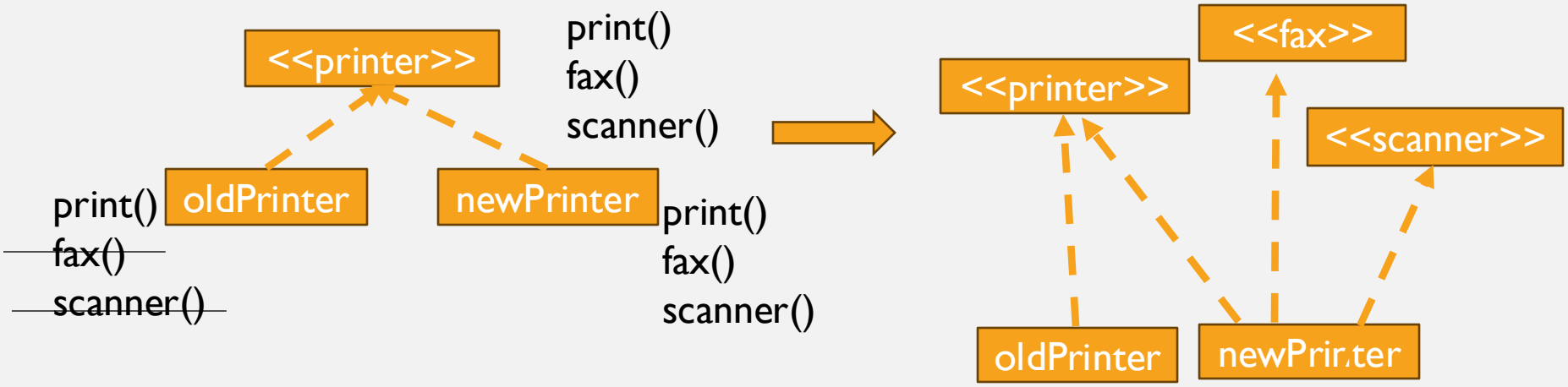
This implementation forces oldPrinter to expose an interface that the class doesn't implement or need.

INTERFACE SEGREGATION PRINCIPLE (ISP)



If oldPrinter implement the method that does not need and throw a notimplemented exception, we will be breaking the LSP

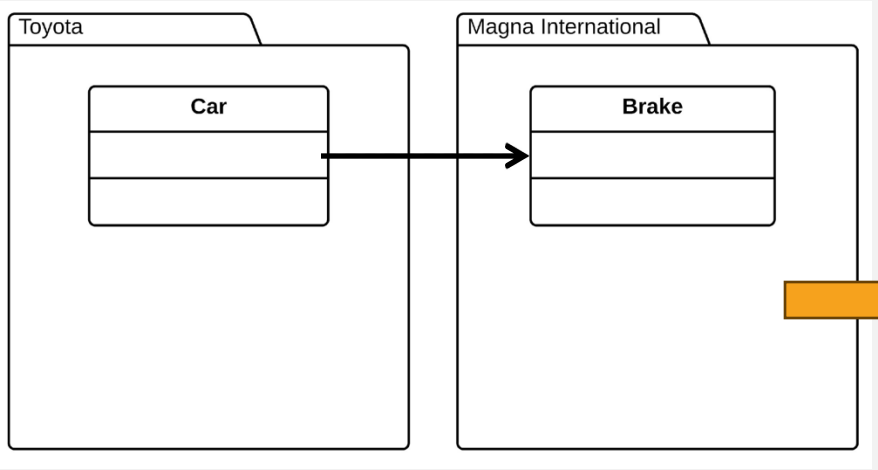
INTERFACE SEGREGATION PRINCIPLE (ISP)

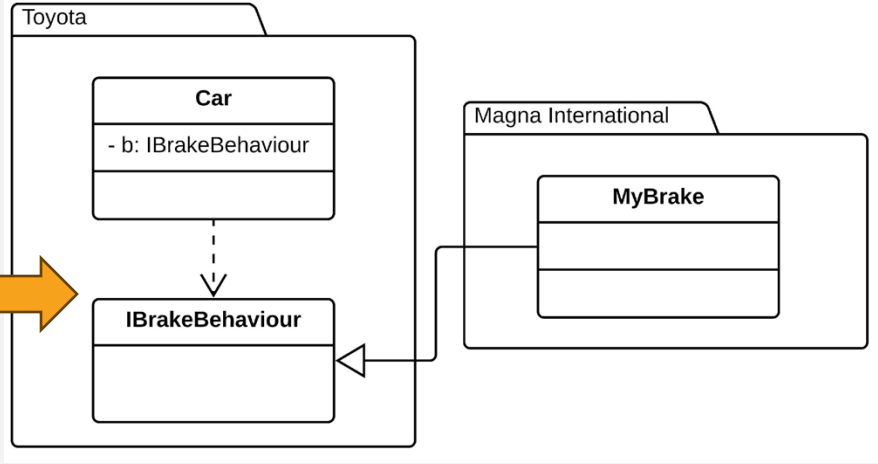


DEPENDENCY INVERSION PRINCIPLE (DIP)

- High-level modules should not depend on low-level modules; instead, they should depend on abstractions
- one should: "Depend upon Abstractions. Do not depend upon implementations."

DEPENDENCY INVERSION PRINCIPLE (DIP)

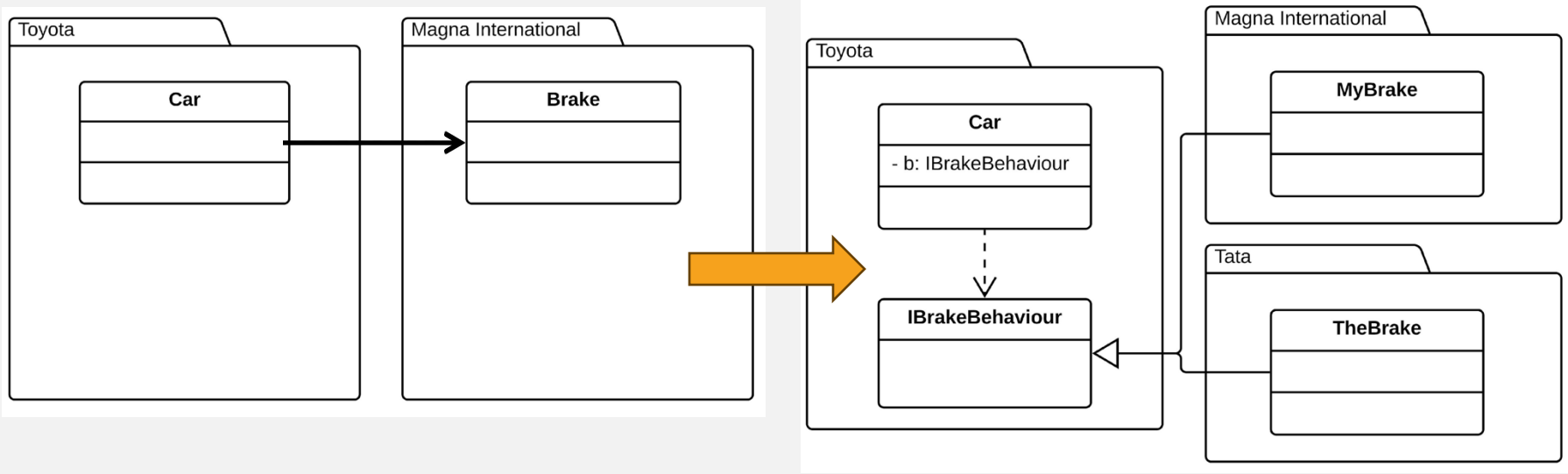




Car is hooked up to just one concrete service(Brake). Any changes to the concrete service will propagate to the client, meaning it will be harder to change or switch out

Car only knows about the interface, and changes in the concrete service will not propagate to the client

DEPENDENCY INVERSION PRINCIPLE (DIP)



SOLID PRINCIPLES

- These are just principles and not rules,
- They purpose is to help you write better code.
- Be carful with taking some design principle too far.
- A lot of the design principles rely in interfaces.
  - Interfaces make everything a lot easier to test as you can just inject a mock whenever you are doing your testing

COUPLING AND COHESION

COUPLING & COHESION INTUITION

Cohesion is the relationship between elements within a module

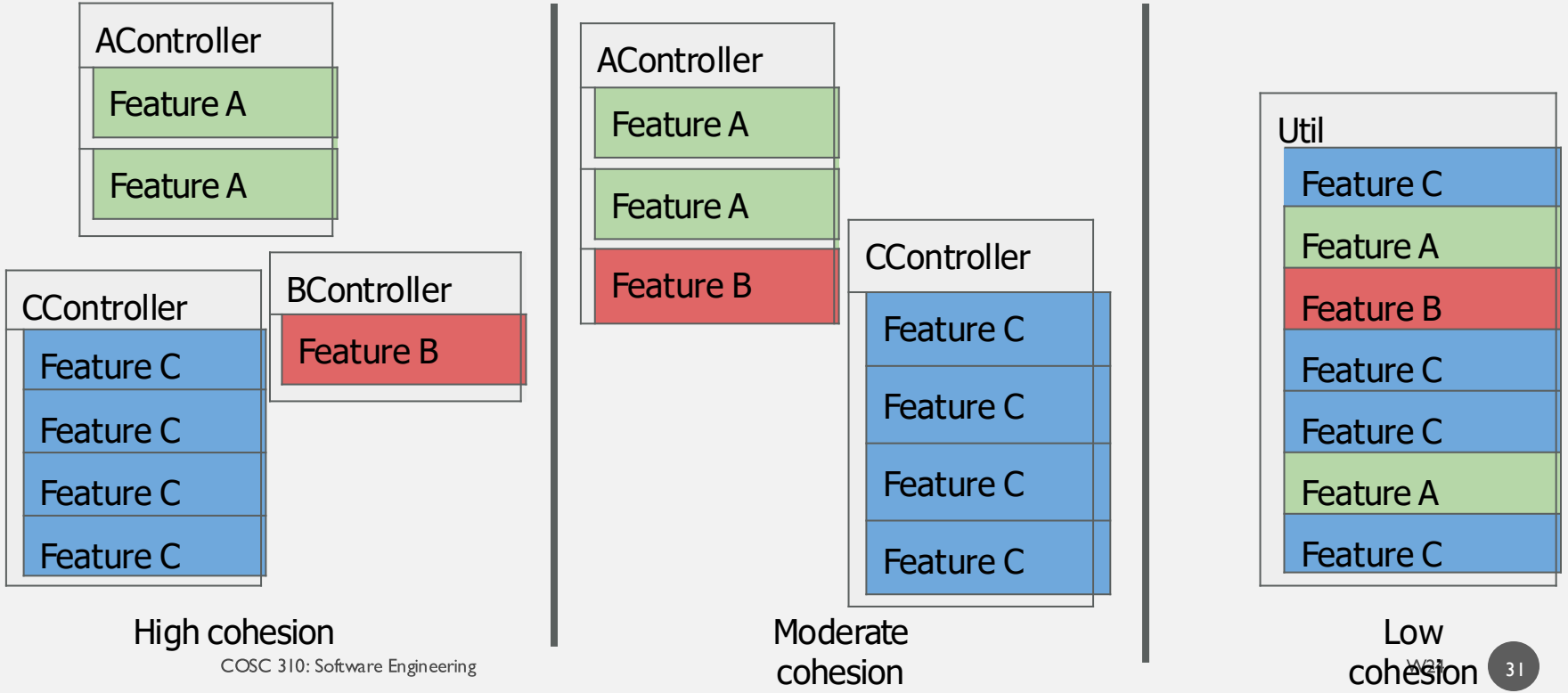
- Modules should only contain functions that belong together.
- Often observable by the data modified by the functions.
- Commonly violated over time as code evolves.

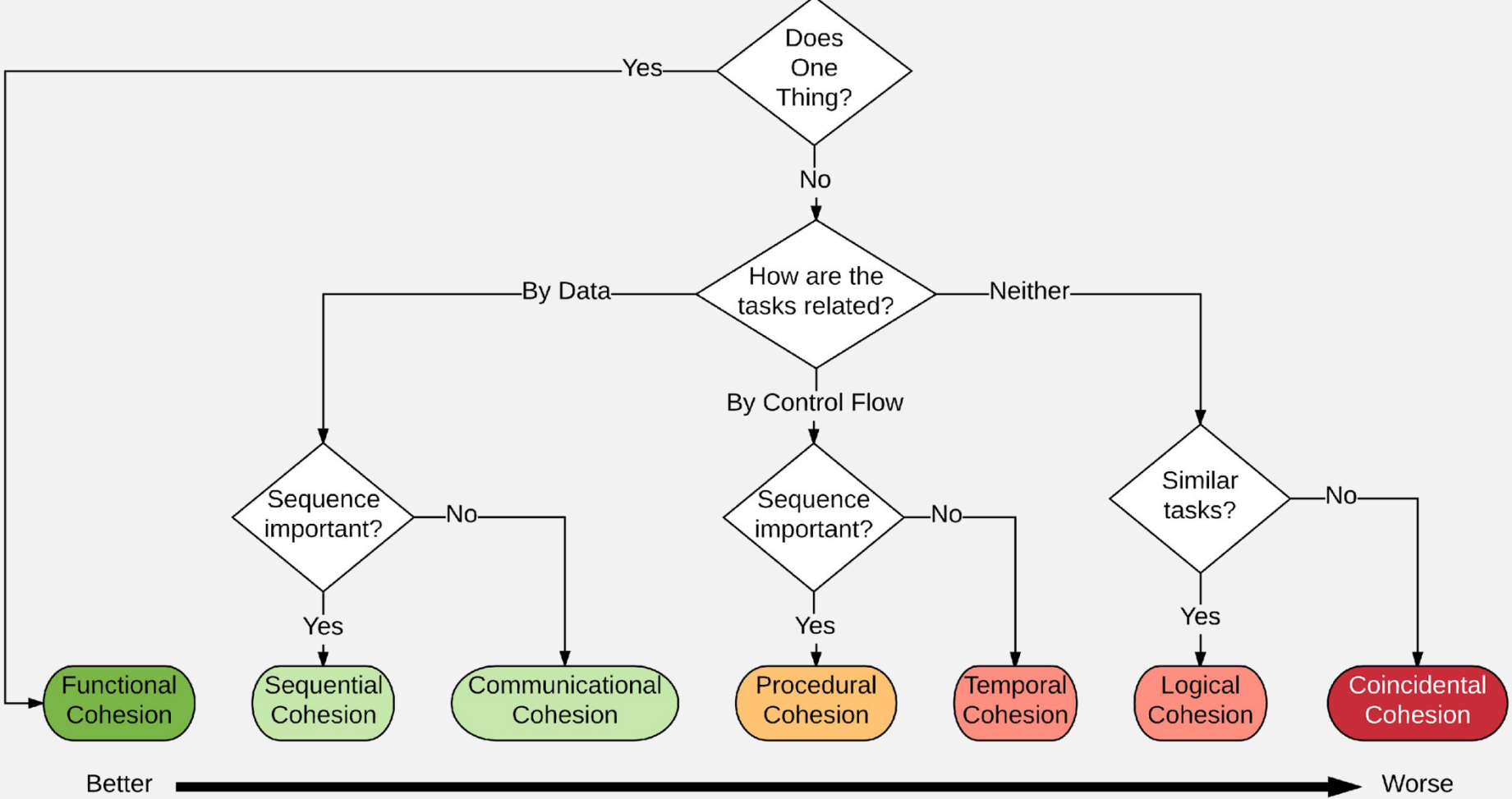
COUPLING & COHESION INTUITION

Coupling is the relationship between modules

- If we make a change in one module, what will the impact be on other modules within the system?
- Modules should not depend on each other's internals.

COHESION


AController
Feature A
Feature A
CController
Feature C
Feature C
Feature C
Feature C
BController
Feature B
High cohesion
AController
Feature A
Feature A
Feature B
CController
Feature C
Feature C
Feature C
Feature C
Moderate cohesion
Util
Feature C
Feature A
Feature B
Feature C
Feature C
Feature A
Feature C
Low cohesion


Does One Thing?
Yes
No
How are the tasks related?
By Data
Neither
By Control Flow
Sequence important?
No
Yes
Sequence important?
No
Yes
Similar tasks?
No
Yes
Functional Cohesion
Sequential Cohesion
Communicational Cohesion
Procedural Cohesion
Temporal Cohesion
Logical Cohesion
Coincidental Cohesion
Better
Worse

COINCIDENTAL COHESION (WORST)

A module has ***coincidental cohesion*** if it performs multiple, completely unrelated actions

- These modules are not maintainable or reusable.
- Processing elements are grouped arbitrarily and have no significant relationship
- The problem is easy to fix by breaking the module into separate modules, each performing one task.

```
myFunc (param1, param2)
{
  read input line
  connect to database
  output param1
  sort param2
}
```

LOGICAL COHESION

- A module has ***logical cohesion*** when it performs a series of related actions, one of which is selected by the calling module.
- Example:

```
op_code = 7;
myFunc (op_code, dummy_1, dummy_2, dummy_3);
// dummy_1, dummy_2, and dummy_3 are dummy variables,
// not used if op_code code is equal to 7
```

- Issues:
  - Difficult to understand and reuse
  - Code for more than one action may be intertwined

TEMPORAL COHESION

- A module has ***temporal cohesion*** when 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

- Issues:
  - Actions are weakly related in time.
  - Hard to reuse.

```
myFunc()
{
   calculate student GPA
   print student GPA
   calculate cumulative GPA
   print cumulative GPA
}
```

PROCEDURAL COHESION

- A module has ***procedural cohesion*** if it performs a series of actions related by the procedure to be followed by the product.

```
myFunc()
{
   read part number
   find record with part number
   find and update inventory for part in database
}
```

- Issues:
  - Actions are related to each other but still would be better as separate methods.
    - E.g.: validate user, process a payment, trigger stock inventory

COMMUNICATIONAL COHESION

- A module has ***communicational cohesion*** if two elements operate on the same input data or contribute towards the same output data.

```
myFunc()
{
   find record with part number
   update part record inventory
}
```

- Issue:
  - Still have issue with reusability as method does more than one thing.
    - E.g.: processing elements that would take a shopping basket and calculate discounts, promotions,

SEQUENTIAL COHESION

- A module has ***sequential cohesion*** if its processing elements are grouped so that the output of one processing element can be used as input for another processing element.
- E.g.
  - *raw_data = read_data()*
  - *parsed_data = parse_data(raw_data)*
  - *structured_data = convert_data(parsed_data)*

FUNCTIONAL COHESION (BEST)

A module with ***functional cohesion*** performs exactly one action (SRP).

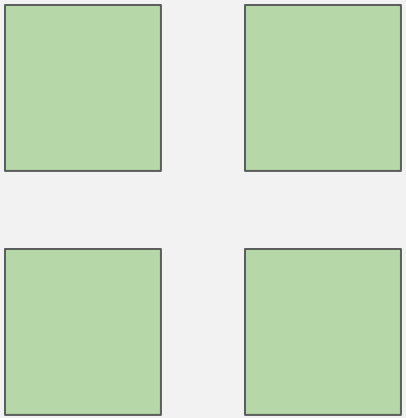
- Simple example is get/set methods.

- Benefits:
  - More reusable
  - Corrective maintenance is easier
    - Fault isolation
    - Fewer regression faults
  - Easier to extend a product

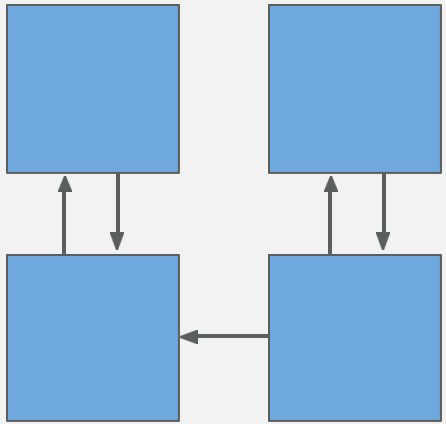
QUESTION

- Which of the following is an example of low cohesion?
  A. A class that has a single, well-defined responsibility
  B. A class that has several unrelated responsibilities
  C. A class that is easily testable
  D. A class that has a lot of public methods

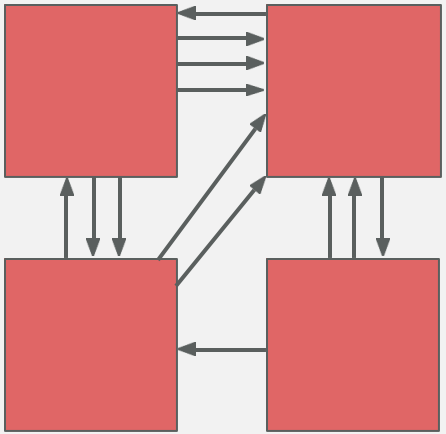
COUPLING



No dependencies



Loosely coupled

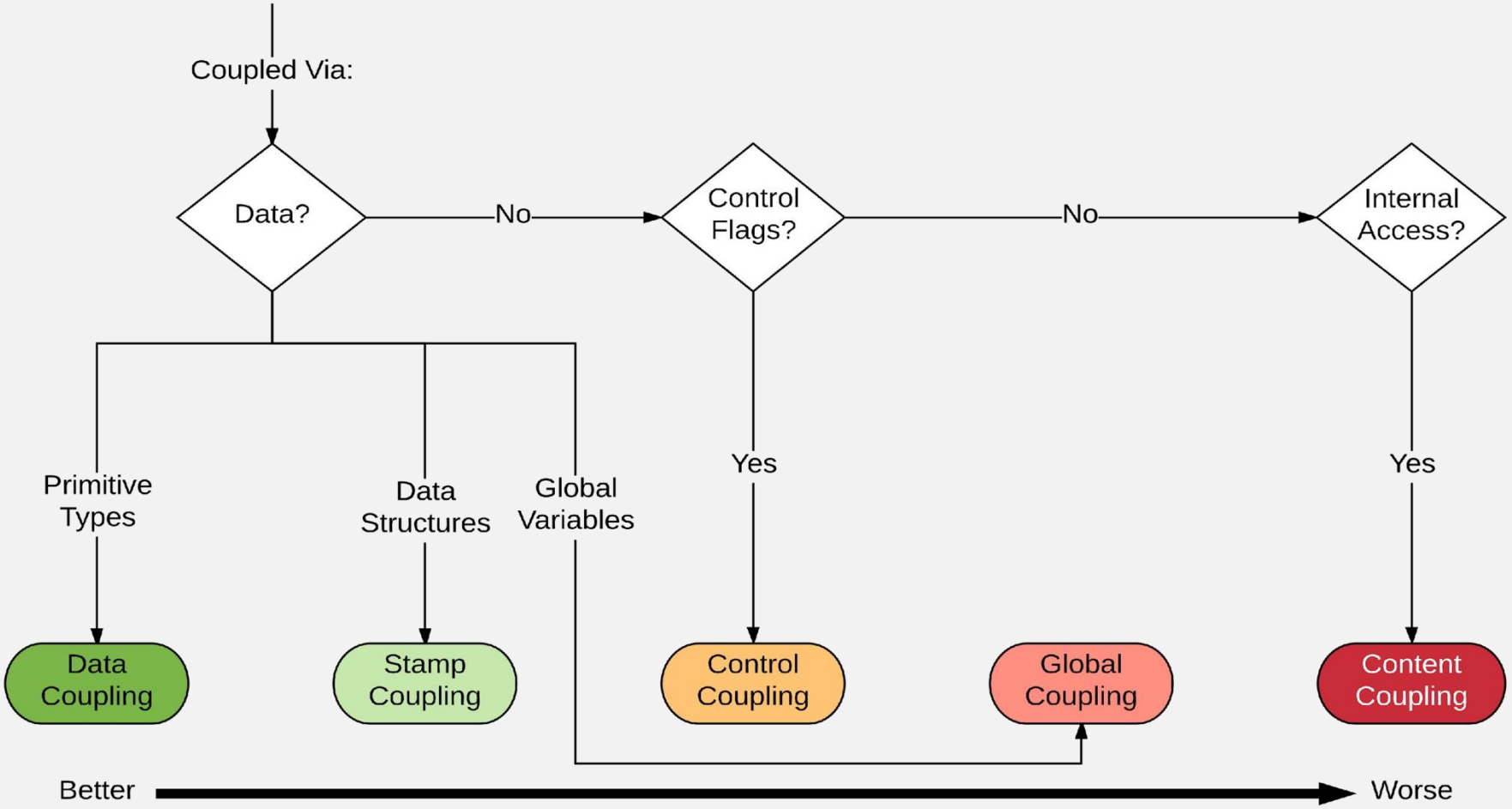


Highly coupled

COUPLING

```java
// Java program to illustrate
// tight coupling concept
class Subject {
    Topic t = new Topic();
    public void startReading()
    {
        t.understand();
    }
}
class Topic {
    public void understand()
    {
        System.out.println("Tight coupling concept");
    }
}
```

```java
// Java program to illustrate
// loose coupling concept
public interface Topic
{
    void understand();
}
class Topic1 implements Topic {
public void understand()
    {
        System.out.println("Got it");
    }
} class Topic2 implements Topic {
public void understand()
    {
        System.out.println("understand");
    }
} public class Subject {
public static void main(String[] args)
    {
        Topic t = new Topic1();
        t.understand();
    }
}
```


Coupled Via:
Data?
No
Control Flags?
No
Internal Access?
Primitive Types
Data Structures
Global Variables
Yes
Yes
Data Coupling
Stamp Coupling
Control Coupling
Global Coupling
Content Coupling
Better
Worse

CONTENT COUPLING (WORST)

- Two modules are ***content coupled*** if one directly access and modified the data of the other. THIS SHOULD BE AVOIDED!!!
  - E.g:
    - Class A has a private instance member int age;
    - Class B sets the value of age directly by a.age = 15

GLOBAL COUPLING

- Two modules are ***global coupled*** if they have write access to global data.
  - That is, global variables are bad!
- Issues:
  - Contradicts the spirit of structured programming by making it hard to trace methods.
  - Side-effects on method execution based on other methods (often undocumented).
  - A change during maintenance of a global variable in one module necessitates corresponding changes in other modules.
  - Common-coupled modules are difficult to reuse and mantain

CONTROL COUPLING

- Two modules are ***control coupled*** if one passes an element of control to the other. Examples:
  - An operation code is passed to a module with logical cohesion.
  - A control switch passed as an argument.
    - process_payment(order_id, payment_mode)
- Issue:
  - Modules are not independent as must know each other's structure.
  - Usually related to modules with logical cohesion.

STAMP COUPLING

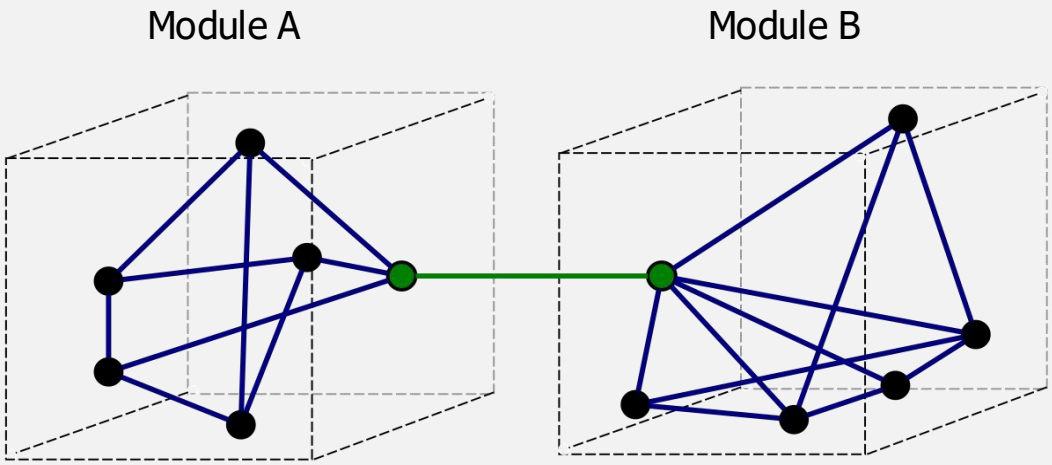
- Two modules are ***stamp coupled*** if a data structure is passed as a parameter, but the called module operates on some but not all of the individual components of the data structure. Example:
  - process_billing(patient).
- Issues:
  - It is not clear, without reading the entire module, which fields of a record are accessed or changed.
  - Difficult to understand and unlikely to be reusable.
  - More data than necessary is passed.
  - However, it is okay to pass a data structure as a parameter, provided that *all* of its fields are used.

DATA COUPLING (BEST)

- Two modules are **data coupled** If the dependency between the modules is based on the fact that they communicate by passing only data. Examples:
  - *display_time_of_arrival (flight_number);*
  - *compute_product (first_number, second_number);*
  - *get_job_with_highest_priority (job_queue);*
- Benefits:
  - The difficulties of content, common, control, and stamp coupling are not present.
  - Maintenance is easier.

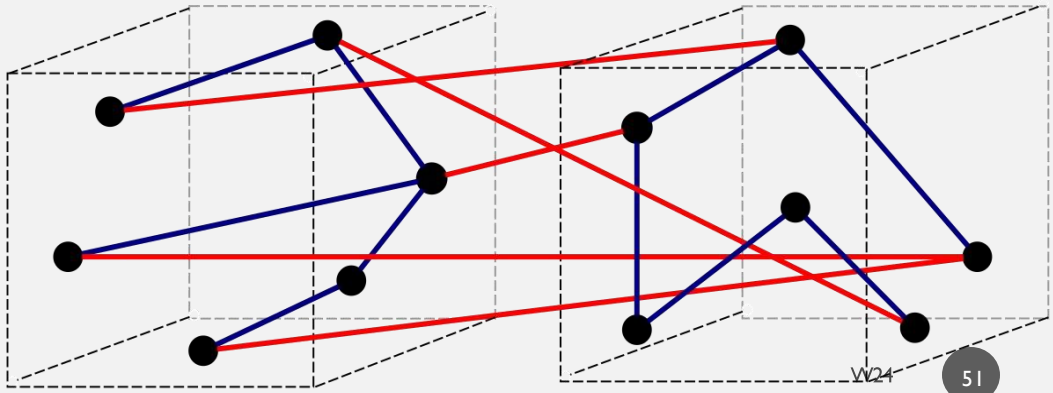
▼Coupling ▲Cohesion

- Lots of within-module collaboration.
- Controlled between-module collaboration.



▼Cohesion ▲Coupling

- Little within-module collaboration.
- Common between-module collaboration.



CONCLUSION

- To create modular maintainable and testable code we use the SOLID principles
- Design quality can be measured by
  - ***cohesion*** (iinteraction within a module) and
  - ***coupling*** (interaction between modules).

Each module should have high cohesion and low coupling.

COSC 310: DESIGN PATTERNS AND ANTIPATTERNS

Dr. Gema Rodriguez-Perez
University of British Columbia
gema.rodriguezperez@ubc.ca

RECAP

- **S O L I D** (one nice combo of principles)
  - **S**ingle Responsibility Principle (High Cohesion, Low Coupling)
  - **O**pen/Closed Principle
  - **L**iskov Substitution Principle
  - **I**nterface Segregation Principle
  - **D**ependency Inversion Principle

https://en.wikipedia.org/wiki/SOLID

OPEN CLOSED PRINCIPLE

```java
class Animal {
    private String name;

    public Animal(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    Run | Debug
    public static void main(String[] args) {
        List<Animal> animals = new ArrayList<>();
        animals.add(new Animal(name:"lion"));
        animals.add(new Animal(name:"mouse"));

        animalSound(animals);
    }
```

- How does this class violate OCP?
- How will this design cause issues in the future?
- How do we make it (the animal_sound) conform to OCP?

```java
    public static void animalSound(List<Animal> animals) {
        for (Animal animal : animals) {
            if (animal.getName().equals("lion")) {
                System.out.println(animal.getName() + " -> roar");
            } else if (animal.getName().equals("mouse")) {
                System.out.println(animal.getName() + " -> squeak");
            }
        }
    }
}
```

OPEN CLOSED PRINCIPLE

```java
public class Animal{

    Run | Debug
    public static void main(String[] args) {
        List<AnimalInterface> animals = new ArrayList<>();
        animals.add(new Lion());
        animals.add(new Mouse());

        animalSound(animals);
    }

    public static void animalSound(List<AnimalInterface> animals) {
        for (AnimalInterface animal : animals) {
            animal.makeSound();
        }
    }
}
```

```java
public interface AnimalInterface {
    String getName();
    void makeSound();
}
```

```java
public class Lion implements AnimalInterface{

    private String name;

    public Lion() {
        this.name = "lion";
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void makeSound() {
        System.out.println(getName() + " -> roar");
    }
}
```

```java
public class Mouse implements AnimalInterface{
    private String name;

    public Mouse() {
        this.name = "mouse";
    }

    @Override
    public String getName() {
        return name;
    }

    @Override
    public void makeSound() {
        System.out.println(getName() + " -> squeak");
    }
}
```

Snake,
Bear,
…..

DEPENDENCY INVERSE PRINCIPLE

```java
public class UserProfile {
    private String name;

    public UserProfile(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    //User Profile class is dependent on CacheStore class
    public void addKeytoCache(CacheStore cacheStore, String key, String value) {
        cacheStore.addKey(key,value);

    }
}
```

```java
public class CacheStore {

    public void addKey(String key, String value) {
        System.out.println("Adding key: " + key + " with value: " + value);
    }

    public void removeKey(String key) {
        System.out.println("Removing key: " + key);
    }

    public void EvictKey(String key) {
        System.out.println("Evicting key: " + key);
    }
}
```

- How does this class violate DIP?
- How will this design cause issues in the future?
- How do we make it conform to DIP?

```java
public class App {
    public static void main(String[] args) throws Exception {
        UserProfile user = new UserProfile("Gema");
        CacheStore cacheStore = new CacheStore();

        //Dependency injection
        user.addKeytoCache(cacheStore,"thisIsmyKey", "123456");

    }
}
```

DIP

```java
public class App {
    public static void main(String[] args) throws Exception {
        UserProfile user = new UserProfile("Gema");
        CacheStoreInterface cacheStore = new CacheStore();

        // UserProfile Class does not need to know the implementation of CacheStore class DIP
        // USerProfile relies on the CacheStoreInterface abstraction, not on the CacheStore implementation
        user.addKeytoCache(cacheStore,"thisIsmyKey", "123456");

    }
}
```

```java
public interface CacheStoreInterface {
    public void addKey(String key, String value);
    public void removeKey(String key);
    public void evictKey(String key);
}
```

```java
public class UserProfile {
    private String name;


    public UserProfile(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    //User Profile class is dependent on CacheStoreInterface abstraction
    public void addKeytoCache(CacheStoreInterface cacheStore, String key, String value) {
        cacheStore.addKey(key,value);

    }
}
```

OBJECTIVES

- Understanding the concept of **design patterns** and how they can be used to solve occurring problems in software design
- Familiarity with the three types of design patterns: **creational**, **structural**, and **behavioral** patterns, and the ability to recognize and apply the most commonly used patterns in each category.
- Knowledge of the key elements of a software design pattern, including its name, problem, solution, and consequences.
- Awareness of common anti-patterns in software design: **BBoM**, **god object**, **golden hammer** and **stove-pipe**
- The ability to recognize how to apply design patterns and avoid anti-patterns in software design to create flexible, maintainable, and high-quality software systems

PATTERNS AND ANTI-PATTERNS

We will cover the basics of **Design Patterns** and **Anti-patterns**, the types, and how they can be used in software design.

- We will also discuss some examples of commonly occurring patterns and anti-patterns in software design, and how to avoid them.

The goal is to be able to recognize common design patterns and anti-patterns in software design, understand their implications, and apply best practices to create high-quality and maintainable software systems.

INTRODUCTION TO DESIGN PATTERNS

In software development, while it may seem that there are a number of ways to solve or implement a solution, there are in fact a limited number of ways (given the constraints of the real world).

Design Patterns describe a general repeatable solution to a commonly occurring problem.

A design pattern isn't a finished design that can be transformed directly into code but describes the characteristics of a solution.

DESIGN PATTERNS

- **We should look to see if a design pattern already exists** instead of re-inventing the wheel.
  - If a pattern exists, use the pattern as is or adapt it
  - Allows for familiarity of use
- Probably the most influential book is: **Design Patterns: Elements of Reusable Object-Oriented Software** by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
  - Available at https://go.exlibris.link/BGXGdvYV. Introduces 23 classic software design patters as well as pitfalls of OO programming
- This is a complex and deep topic. Thus, we will just look at an introduction of what a design pattern is with some of the more basic patterns

ANTI-PATTERNS

ANTI-PATTERNS

So what's an anti-pattern?

- A common solution to a recurring problem that initially appears to be effective **BUT** ultimately results in negative consequences or problems that outweigh any benefits
- It is the opposite of a design pattern, which is a proven solution to a recurring problem that has been demonstrated to be effective in practice

- **Anti-patterns** can occur at any level of a software development project, from coding practices to system architecture, and can cause issues such as **reduced code quality, poor performance, and increased maintenance costs.**
  - The identification and avoidance of anti-patterns is an important aspect of software engineering, as it can help ensure the overall success and quality of a project.
  - Recognize that you have a problem and avoid it.

ANTI-PATTERNS

There are two key elements to an **anti-pattern** that distinguish it from a **bad habit, bad practice, or bad idea**:

1. The anti-pattern is a commonly-used process, structure or pattern of action that, despite initially appearing to be an appropriate and effective response to a problem, has more bad consequences than good ones
2. Another solution exists to the problem the anti-pattern is attempting to address. This solution is documented, repeatable, and proven to be effective where the anti-pattern is not

For somethings to recognized as an anti-pattern it must pass the **Rule of Three**

- Has it been witnessed to occur at least 3 times.

•Neill, Colin J.; Laplante, Philip A.; DeFranco, Joanna F. (2011). Antipatterns: Managing Software Organizations and People. Applied Software Engineering Series (second ed.). CRC Press. ISBN 9781439862162.

ANTI-PATTERNS

Some common anti-patterns:

- BBOM (big ball of mud) – Programming anti-pattern
- God object – Programming anti-pattern
- Golden hammer – Process anti-pattern

The question to consider is how they lead to poor design or implementation

BIG BALL OF MUD (BBOM)

- Aka:
  - Spaghetti code
  - Duct-tape and baling-wire code
- Key features:
  - Unregulated growth
  - Repeated sections of code
  - Almost impossible to maintain
- Structure of system may never have been planned or well defined (lack of everything we've talked about to date)

https://sourcemaking.com/antipatterns/spaghetti-code

BIG BALL OF MUD (BBOM)

```python
def process_data(data):
    if data:
        for item in data:
            if item:
                if 'price' in item:
                    price = item['price']
                    if 'discount' in item:
                        discount = item['discount']
                        if discount > 0:
                            final_price = price * (1 - discount / 100)
                            if 'quantity' in item:
                                quantity = item['quantity']
                                if quantity > 0:
                                    total_cost = final_price * quantity
                                    print("Total Cost:", total_cost)
```

Spaghetti code: **Excessive Nesting**

```python
def calculate_numbers():
    i = 0
    while i < 10:
        print("Value of i:", i)
        i += 1
        if i >= 10:
            break
        else:
            continue
```

Spaghetti code: **Go To statements**

GOD-CLASS OBJECT

Refers to an object within a software system **that knows or does too much**

- An object that is the central point of control for a significant portion of the system
- Majority of the system's functionality is contained within a single class or module
  - High **coupling** (dependence) and low **cohesion** (responsibilities are not well-defined/isolated)
- Violates the principles of good software design such as:
  - **Single Responsibility Principle (SRP)** (Each class or module should have only one responsibility)
- Makes testing and maintenance really, really challenging and hard
- Increases the complexity of the system and creates dependencies between different parts of the code
- can lead to performance issues due to complexity (failing to use sub-routines/methods, too many global variables)

Use modular and well-structured design patterns to achieve quality, maintainability and performance.

GOD-CLASS OBJECT

- An AdminController class with all available actions regarding the admin of an application can be considered as a god object.
- On an e-commerce application, if we find methods about checkout, orders, trackings , shipments and customers in the same class, then it can be considered as a god object.
- If a class or module has hundreds or thousands of lines of code, dozens of methods, fields, and parameters, and many nested if-else or switch statements, it is likely a god object.

GOLDEN HAMMER

- AKA Maslow's Hammer, Law of the Instrument:
  - *If the only tool you have is a hammer, it is tempting to treat everything as if it were a nail.*
- When an individual or team has gained a high level of competence/expertise in a given area/technology they attempt to use it for every new product or development effort
  - Ignores a mismatch between technology and problem in favor of minimal effort
- Some issues
  - Existing products dictate design and system architecture
  - New development relies heavily on a specific vendor product or technology
  - Architectures starts to be described by a particular produce or tool set

https://sourcemaking.com/antipatterns/golden-hammer

GOLDEN HAMMER

- Imagine you've spent a decade mastering databases—designing, optimizing, and integrating them seamlessly with websites. Now, in your new role, you're tasked with building a **static website** where the content remains unchanged.
- If you find yourself spending days setting up a database—**not because the project requires it, but simply because databases are your expertise**—you've fallen into an **anti-pattern**.

This is a classic case of **overengineering**, where familiarity with a tool leads to unnecessary complexity rather than an optimal solution.

QUESTION

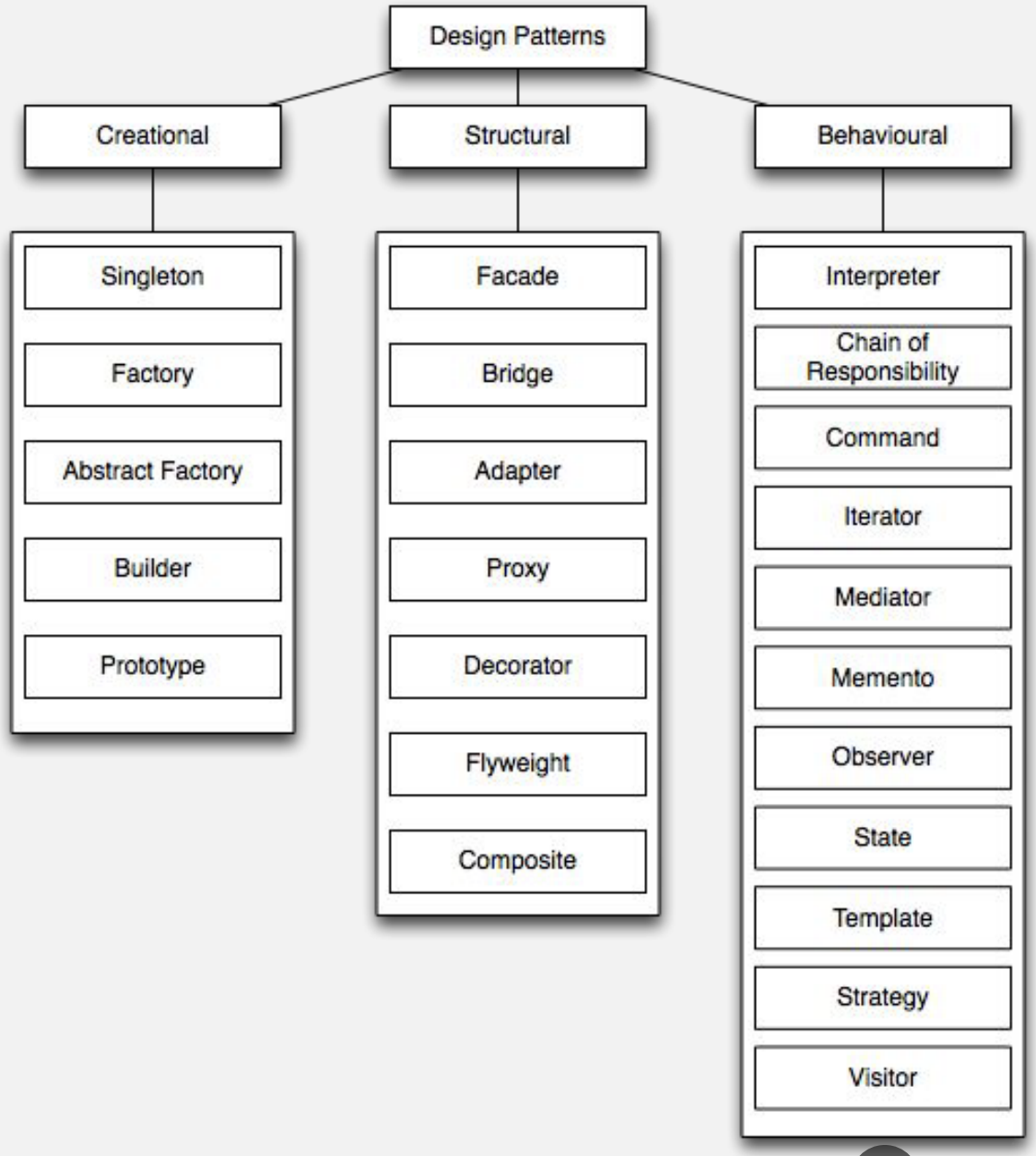
What is the result of using the God Object anti-pattern in software engineering?

A. Improved code readability
B. Increased flexibility
C. Increased code maintainability
D. Increased coupling and decreased cohesion
E. Improved software performance

DESIGN PATTERNS

THERE ARE LOTS OF OO DESIGN PATTERNS!

this is just some of them!



CATEGORIES OF DESIGN PATTERNS

Creational

- Deal with the process of object creation, providing mechanisms for creating objects in a way that is suitable for a given situation.
- Vary object creation

Structural

- Deal with the composition of classes and objects, providing ways to form larger structures from individual parts
- Vary object structure

CATEGORIES OF DESIGN PATTERNS

Behavioral

- Deal with the communication between objects and the delegation of responsibility among them, providing ways for objects to interact and fulfill their tasks.
- Vary the behavior you want

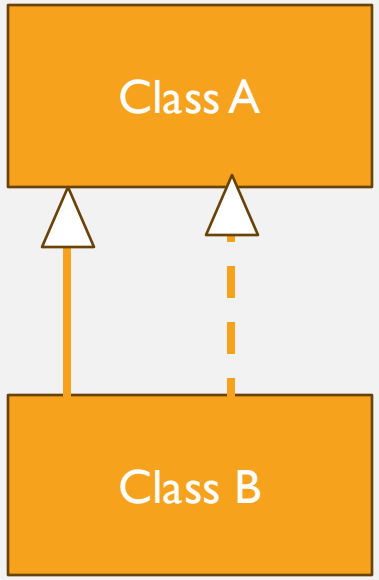
There is a lot, so we will just examine a few.

RECAP ESSENTIALS OOP

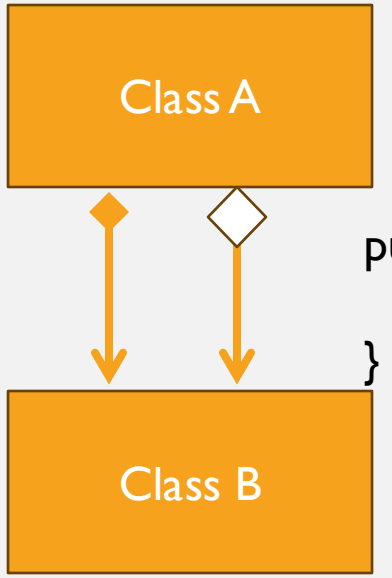
- **Interface**: A contract that specifies the capabilities that a class should provide
- **Encapsulation**: A class's variables are hidden from other classes and can only be accessed by the methods of the class in which they are found
- **Abstraction**: Reduce complexity by hiding unnecessary details in a class.
- **Inheritance**: Mechanism that allows us to reuse code across our classes.
- **Polymorphism**: Ability of an object to take many forms.

UML RECAP

public class B
extends/implements A {}

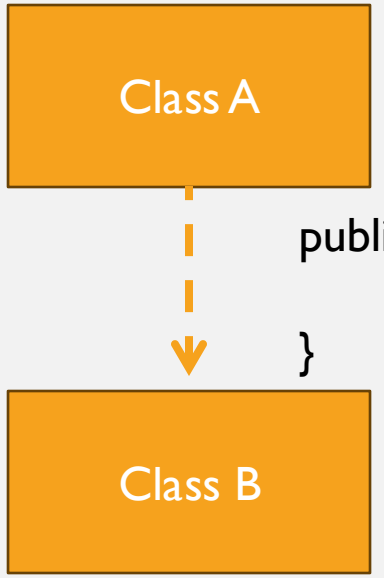


Inheritance/Interface relationship




```
public class A {
    private B b;
}
```

Composition/Aggregation relationship




```
public class A {
    public void methodA (B b);
}
```

Dependency relationship

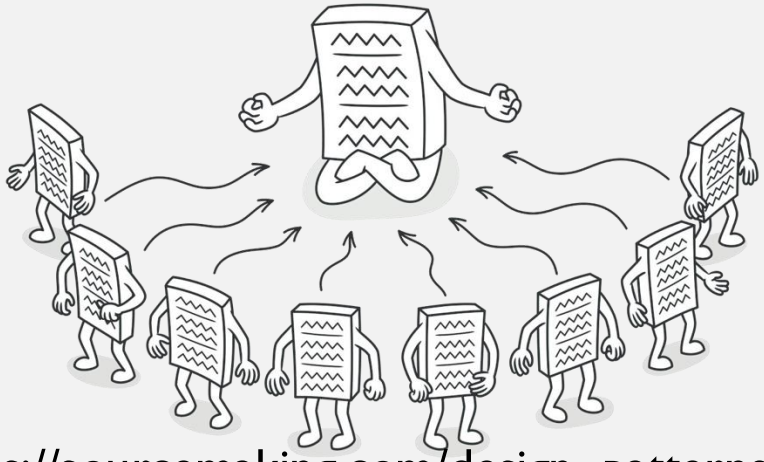
SINGLETON

(Creational design pattern)

Ensures that a class has only one instance, providing a global point of access to it

THE SINGLETON PATTERN

- The singleton pattern is a design pattern that restricts the instantiation of a class to one object. Additionally, lazy initialization and global access are necessary.
- This is useful when:
  - exactly one object is needed to coordinate actions across the system.
  - you want to ensure that there is only one instance of a class in the system



https://sourcemaking.com/design_patterns/singleton

SINGLETON: BENEFITS

- **Ensures only one instance** is created.
- **Reduces memory and resource usage.**
- **Centralized access** to shared resources.
- **Prevents inconsistent states** across multiple instances.

SCENARIO

You are creating a calendar application for the prime minister and different department staff need to access to it for booking meetings between the department and the prime minister . You need to ensure there is only one calendar at any point in time.

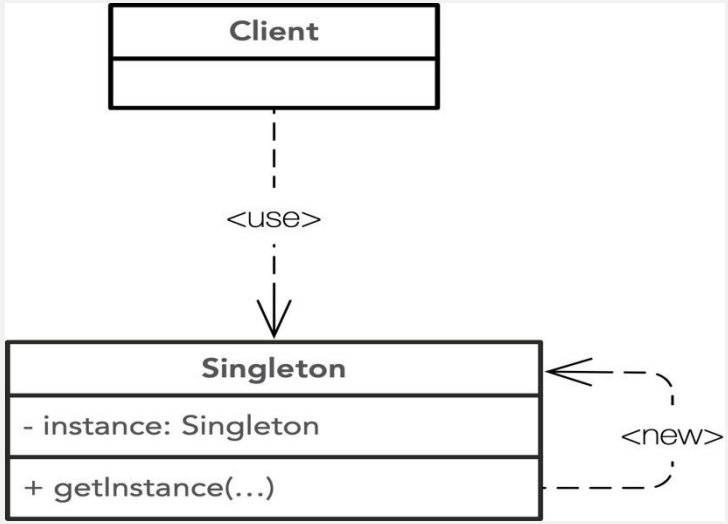
- In the real world, the prime minister has one and only one official calendar.

DISCUSSION

How can we solve it?

SUGGESTED IMPLEMENTATION

1. Make the class of the single instance object responsible for creation, initialization, and access.
2. Declare the instance as a private static instance of the class.
3. Provide a public static member function that encapsulates all initialization code, and provides access to the instance/object.
4. The client calls the accessor function whenever a reference to the single instance is required.



EXERCISE

- Implement a singleton Class and test the implementation:

```
public class App {
    Run | Debug
    public static void main(String[] args) {
        Calendar singleton = Calendar.getInstance();
        Calendar singleton2 = Calendar.getInstance();

        if (singleton == singleton2) {
            System.out.println("Singleton works");
            singleton.showMeetingsToday();
        } else {
            System.out.println("Singleton failed");
        }
    }
}
```

```
Singleton works
Meetings today
```

```java
public class Calendar {

    // Private static variable of the same class that is the only instance of the class
    private static Calendar singleCalendar = null;

    // Private constructor to restrict instantiation of the class from other classes
    private Calendar() {
        // Initialization code here
    }

    // Public static method that returns the instance of the class, this is the global access point
    public static Calendar getInstance() {
        if (singleCalendar == null) {
            singleCalendar = new Calendar();
        }
        return singleCalendar;
    }

    public void showMeetingsToday() {
        System.out.println("Meetings today");
    }
}
```

SINGLETON PATTERN PROBLEMS

- Requires special treatment in multi-threaded environments (where multiple threads need to access the single instance)
- Introduces a new **global** with dependencies spread throughout the codebase -> (implicit coupling)
- Singleton classes often create tight coupling between different parts of the system, making it difficult to substitute dependencies for testing.

FACTORY METHOD

(Creational design pattern)

Use of factory methods to deal with the problem of creating objects without having to specify their exact classes

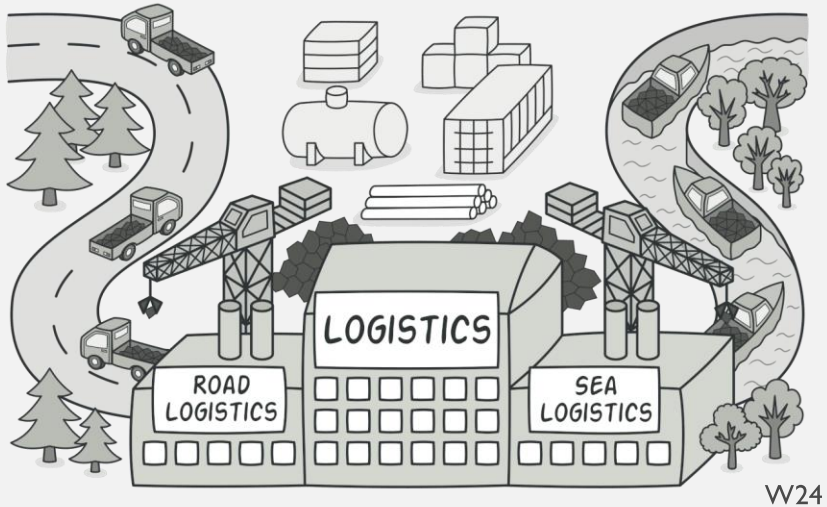
THE FACTORY PATTERN

- Allows for the creation of objects without specifying the exact class of object that will be created
  - Greater flexibility in the system and makes it easier to add new types of objects to the system
  - Use when you don't know beforehand the exact types and dependencies of the objects your code should work with.
- Advantages
  - Decoupling the creation of objects from the calling code; more flexible and maintainable
  - Encapsulating object creation logic in a single location; easier to modify and extend
  - Hiding the implementation details of object creation from the calling code

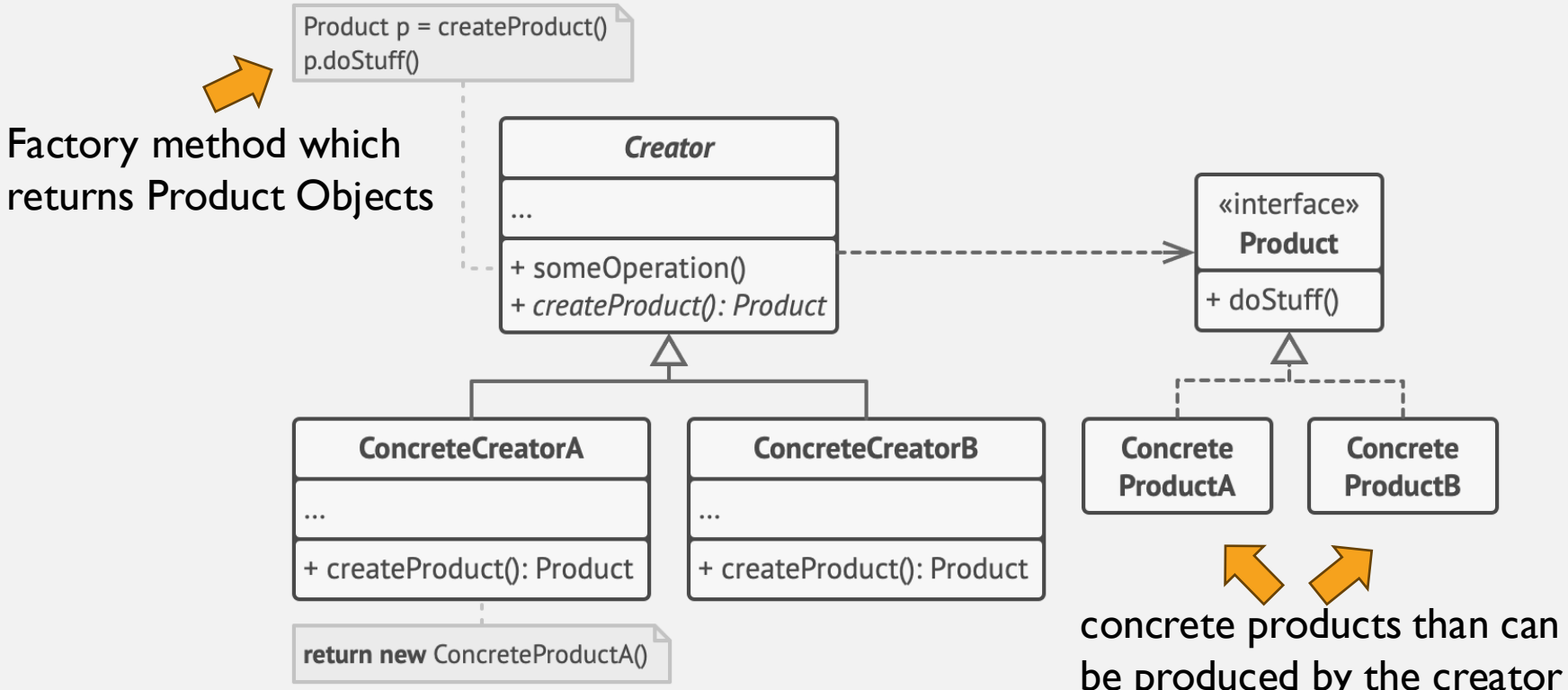
SCENARIO

You are developing a **logistics management application** that helps companies manage their transportation needs. Initially, the app is designed to handle only truck-based deliveries. As your app gains popularity, several **sea transportation companies** start requesting support for **cargo ships**. You realize that adding ship-based logistics directly into the existing code would lead to **tight coupling**. To solve this, you decide to **refactor the application using the Factory Method pattern**

https://refactoring.guru/images/patterns/diagrams/factory-method



THE FACTORY PATTERN



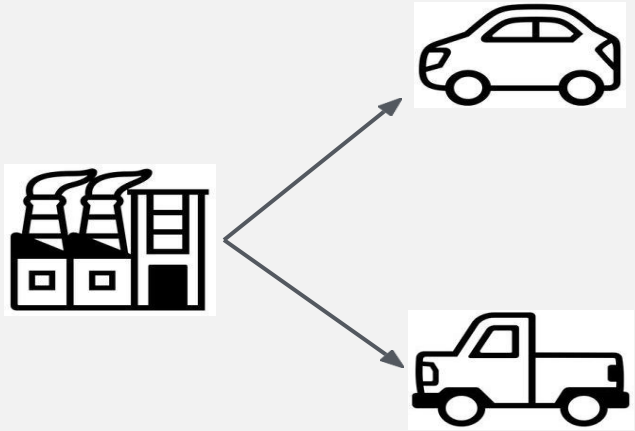
https://refactoring.guru/images/patterns/diagrams/factory-method/structure-2x.png

SUGGESTED IMPLEMENTATION

1. Define a common product interface.
2. Make all concrete products follow that interface.
3. Add a factory method inside the creator class (Factory). The return type of the method should match the common product interface.
4. Create a set of creator subclasses for each type of product listed in the factory method, these subclasses will handle the creation of the concrete products
5. The client code uses the factory method to create the correct transport.

EXERCISE

You are working on an implementation for managing Cars. These can have different categories (Big and Small). Much logic is involved to create your car objects and you **don't always know** what type of car should be created. Use the factory method to build small and big cars



EXERCISE

We are building some kind of game where the player has to avoid some obstacles. The obstacles have different attributes (i.e., speed, position, size, etc). The game has different levels. In each level there might be new types of obstacles and the speed of the obstacles increase in each level.

Use the factory method pattern and draw the UML class diagram for creating the objects.

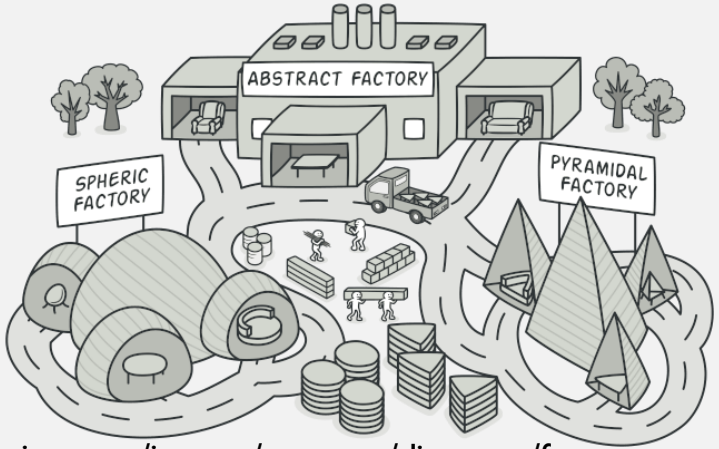
ABSTRACT FACTORY

(Creational design pattern)

Allows to produce families of related objects without specifying their concrete classes

ABSTRACT FACTORY PATTERN

- Provides an interface for creating families of related or dependent objects without specifying their concrete classes and implementation.
- It provides a set of rules or instructions that let you create different types of things without knowing exactly what those things are



https://refactoring.guru/images/patterns/diagrams/factory-method/structure-2x.png

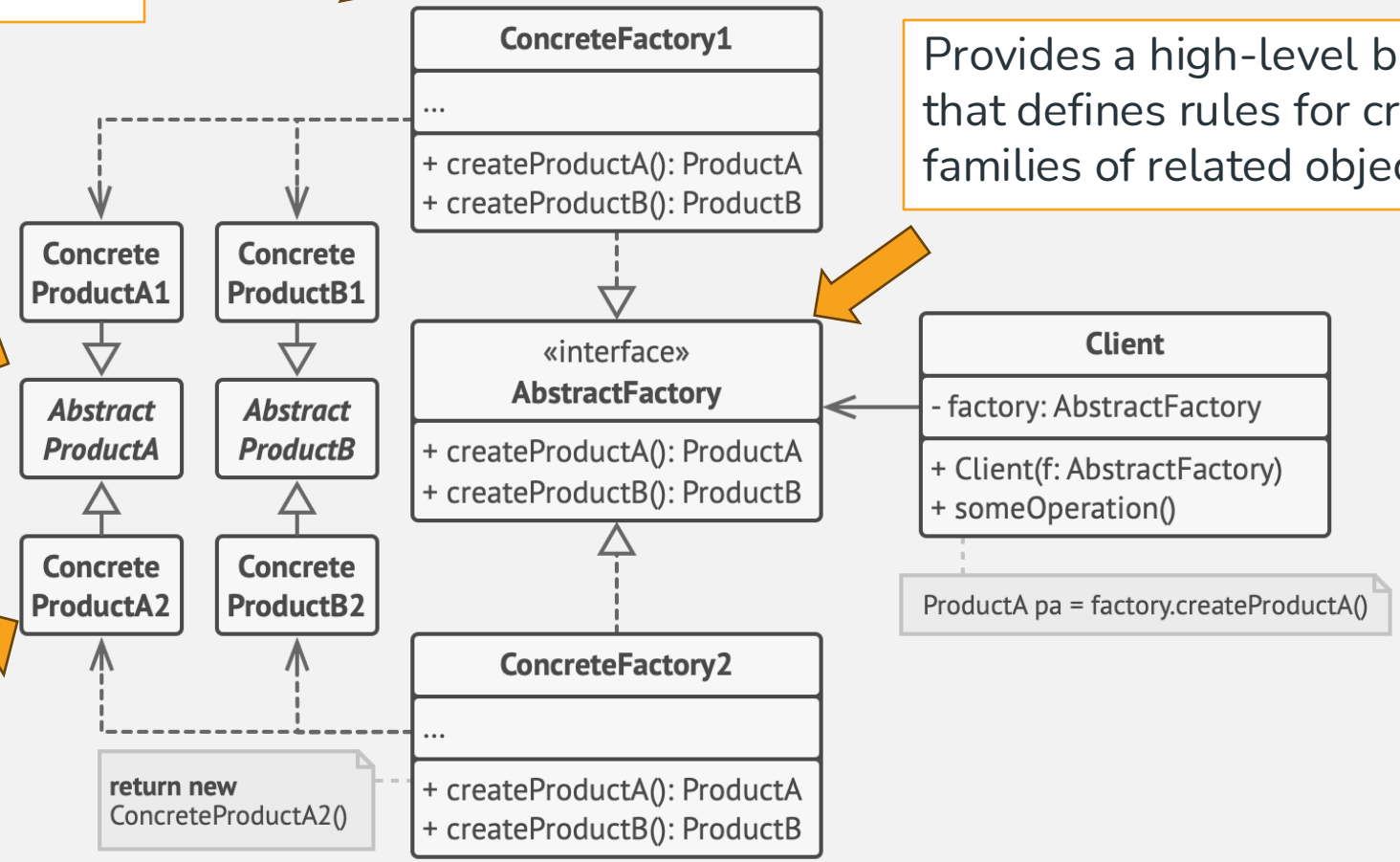
SOLUTION

1. **Abstract Factory** interface declares a set of methods for creating each of the abstract products.
2. **Concrete Factories** implement creation methods of the abstract factory. Each concrete factory corresponds to a specific variant of products and creates only those product variants.
3. **Abstract Products**: declare interfaces for a set of distinct but related products which make up a product family. Although concrete factories instantiate concrete products, signatures of their creation methods must return corresponding abstract products. This way the client code that uses a factory doesn't get coupled to the specific variant of the product it gets from a factory
4. **Concrete Products:** are various implementations of abstract products, grouped by variants. Each abstract product must be implemented in all given variants

Implement the rules specified by the abstract factory. Contain the logic for creating specific instances of objects within a family.

represents a family of related objects by defining a set of common methods or properties

They are the actual instances of objects created by concrete factories.



Provides a high-level blueprint that defines rules for creating families of related object

EXERCISE

Imagine you're developing a cross-platform application that must render UI (button, dialogs, and status Bars) components differently based on the operating system (Windows, macOS, Linux). Instead of writing OS-specific logic throughout the codebase, you use the **Abstract Factory Pattern** to delegate object creation to platform-specific factories.

Draw the UML class diagram to solve this problem.

STRUCTURAL

Adapter Pattern:

- Converts the interface of a class into another interface clients expect, allowing classes to work together that otherwise could not
- Often used to integrate legacy code with modern systems.

Decorator Pattern:

- Adds additional behavior to an object dynamically, without changing its original structure
- Allows you to extend the behavior of an object at runtime, and can be used to add new features to existing classes without modifying them directly

STRUCTURAL

Bridge Pattern:

- Decouples an abstraction from its implementation, allowing them to vary independently
- Reduces coupling between a class and implementation; more flexible, easier to maintain
- Hides implementation from client

There are more….

ADAPTER

STRUCTURAL PATTERN

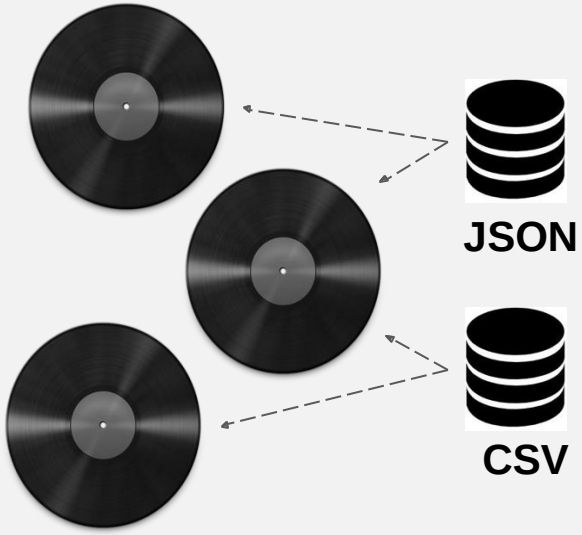
Allows objects with incompatible interfaces to collaborate

THE ADAPTER PATTERN

- It is a structural design pattern that allows objects with incompatible interfaces to collaborate.
- It is a special object that converts the interface of one object so that another object can understand it.
- An adapter wraps one of the objects to hide the complexity of conversion happening behind the scenes. The wrapped object isn't even aware of the adapter.
- Use the Adapter class when you want to use some existing class, but its interface isn't compatible with the rest of your code.
- Use the pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the superclass.

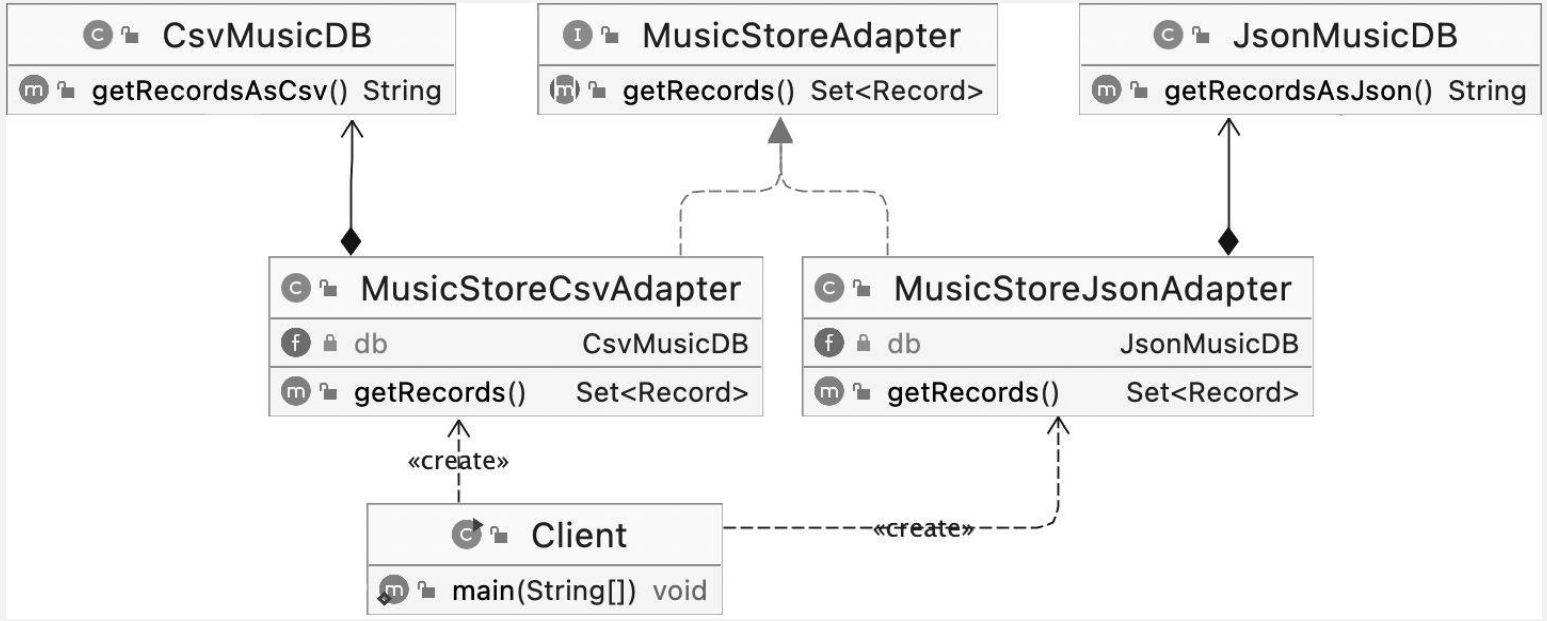
SCENARIO

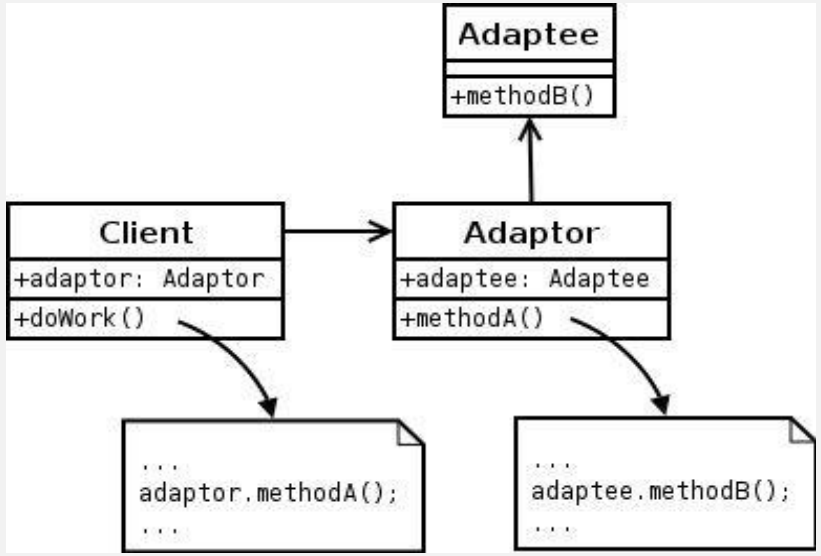
You are developing a web application for a music store that manages a catalog of music records. Your application already defines a standard interface for retrieving music **record objects.** However, you now need to integrate data from multiple external sources, where records are provided in **JSON and CSV** formats.



ADAPTER SUGGESTED IMPLEMENTATION

Convert the interface of a class into another interface that clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces. Wrap an existing class with a new interface.





ADAPTER SUGGESTED IMPLEMENTATION

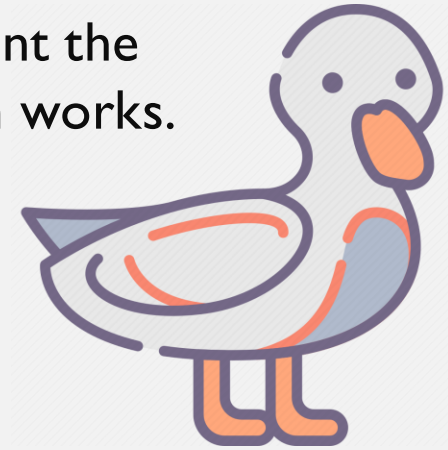
- Make sure that you have at least two classes with incompatible interfaces
- Declare the client interface and describe how clients communicate with the service
- Create the adapter class and make it follow the client interface.
- Add a field to the adapter class to store a reference to the service object.
- One by one, implement all methods of the client interface in the adapter class. The adapter should delegate most of the real work to the service object.
- Clients should use the adapter via the client interface.

EXERCISE

A Turkey amongst DUCKS: If it walks like a duck and quacks like a duck, then it must be a duck!

Or …. It **might** be a **turkey wrapped with a duck adapter**!

Checkout the "Adapter" available on Canvas and implement the TurkeyAdapter class to make the DuckSimulator program works.



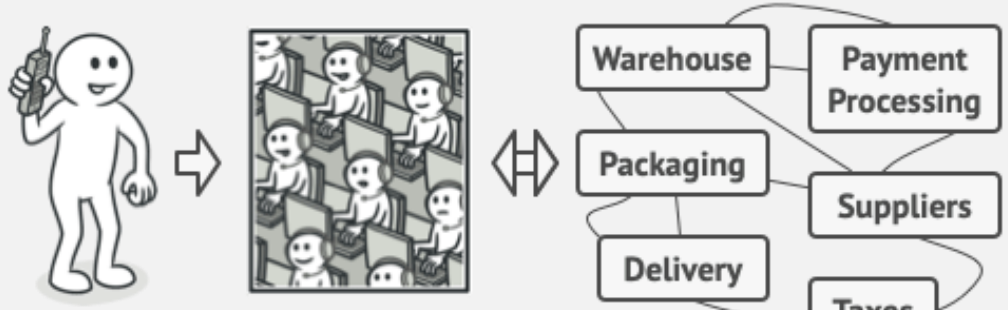
FACADE

STRUCTURAL PATTERN

Provides a simplified interface to a library, a framework, or any other complex set of classes.

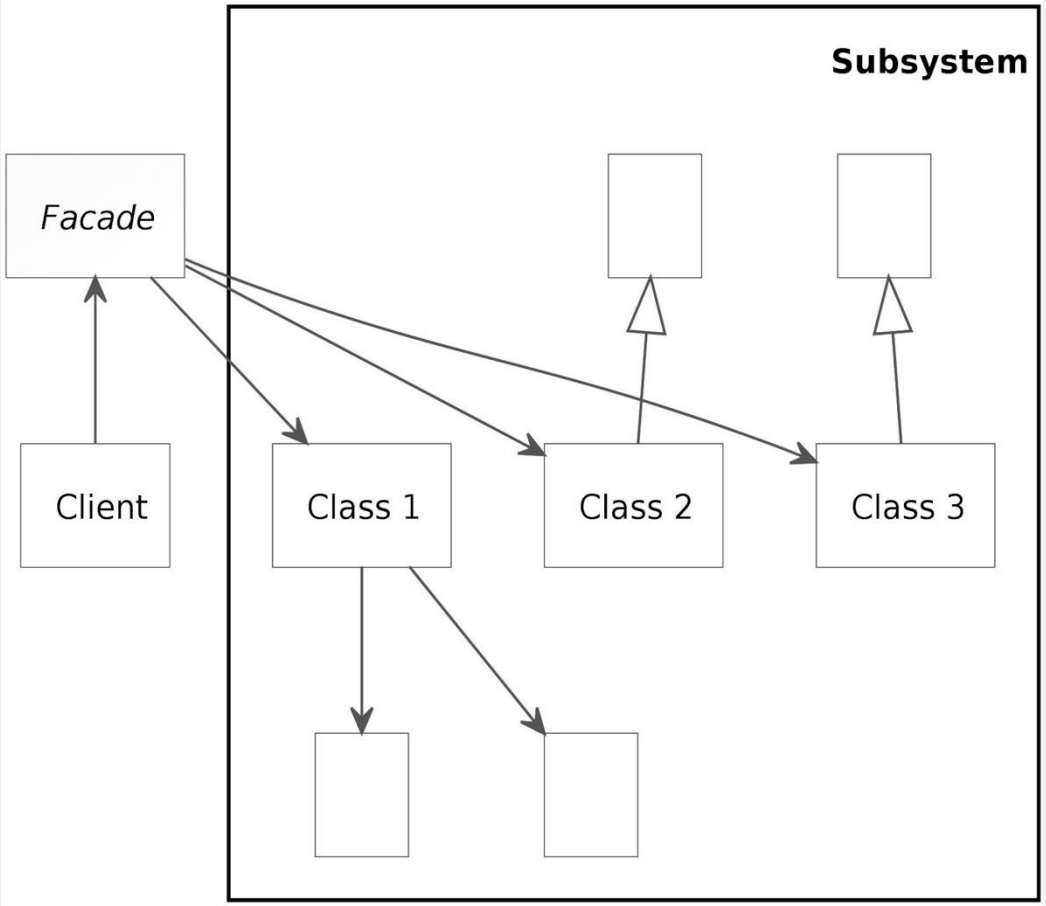
FACADE PATTERN

- Provides a simplified interface to a library, a framework, or any other complex set of classes
- Useful when you need a simple interface to a complex subsystem
- You want to avoid tight coupling of your implementation to the complex subsystem



https://refactoring.guru/design-patterns/facade

FACADE PATTERN



- provides a simplified interface to a library, a framework, or any other complex set of classes
- Useful when you need a simple interface to a complex subsystem
- You want to avoid tight coupling of your implementation to the complex subsystem

SCENARIO

You are implementing a query engine that lets clients add a dataset from an archive, remove a dataset, and perform complex queries on the dataset. Your code base has grown significantly and you want to provide a simplified interface to clients.

FACADE SUGGESTED IMPLEMENTATION

- Check whether it's possible to provide a simpler interface than what an existing subsystem already provides.
- Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem.
- The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.
- To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade.

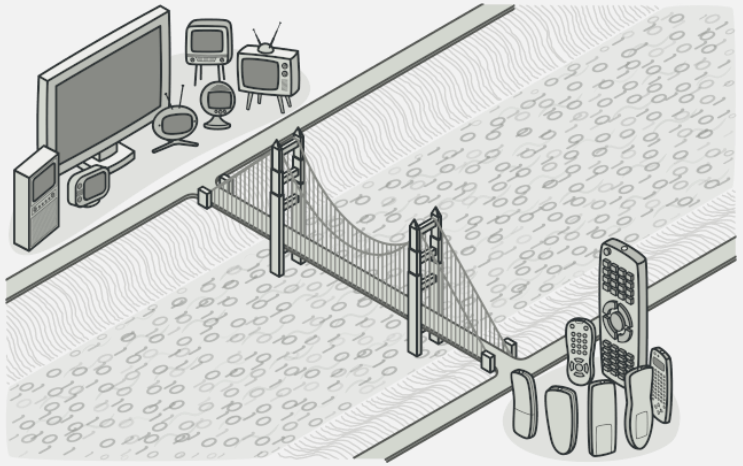
BRIDGE

STRUCTURAL PATTERN

Decouple an abstraction from its implementation so that the two can vary independently.

THE BRIDGE PATTERN

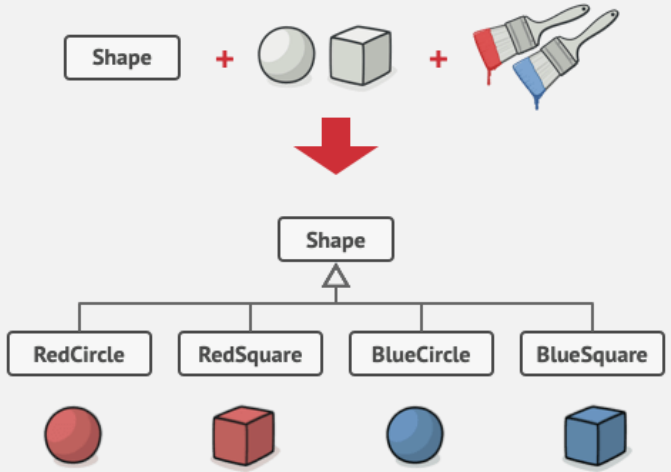
- It helps you **separate abstraction from implementation**, making code **more flexible and maintainable**.
- Think of a bridge in real life—it connects two parts of land while remaining independent of the type of vehicles crossing it. Similarly, the **Bridge pattern** separates the **high-level logic (abstraction)** from the **low-level implementation details**, allowing them to evolve **independently**.



https://refactoring.guru/design-patterns/bridge

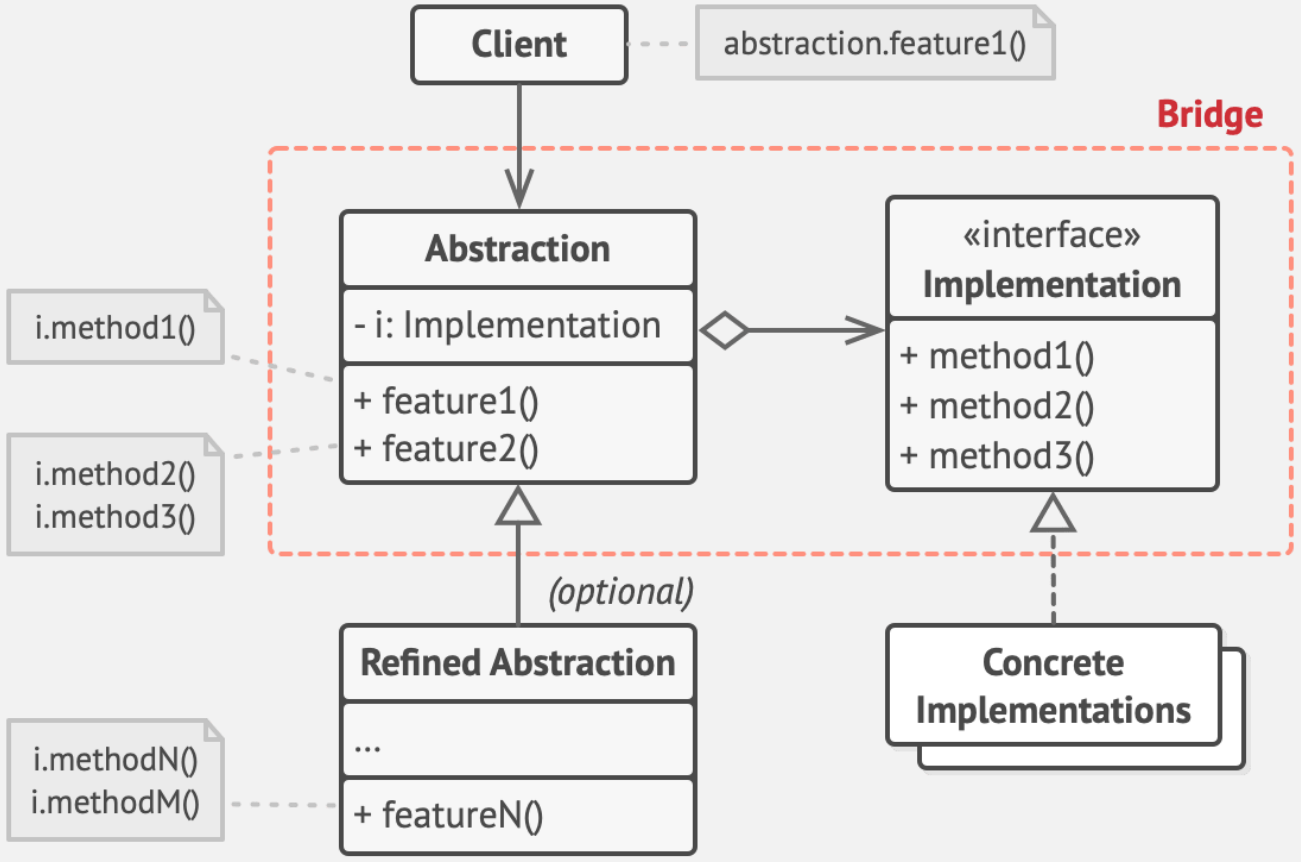
SCENARIO

- Suppose you are designing a **Shape** class that needs to support **different colors**. Instead of hardcoding color choices in each shape class, we **separate** shape logic from color logic using the Bridge pattern.
- This pattern is widely used in **GUI frameworks, file systems, and driver development** where multiple variations of a component exist.



https://refactoring.guru/design-patterns/bridge

BRIDGE IMPLEMENTATION



BEHAVIOURAL PATTERNS

Observer Pattern:

- used to establish a one-to-many dependency between objects.
- When the subject changes state, it automatically notifies all its observers keeping them updated without tightly coupling them.

Iterator Pattern:

- Provides a way to access the elements of a collection sequentially, without exposing its underlying representation
- Simplifies the traversal of complex data structures, and to provide a standardized way of iterating over collections

OBSERVER PATTERN

BEHAVIOURAL PATTERN

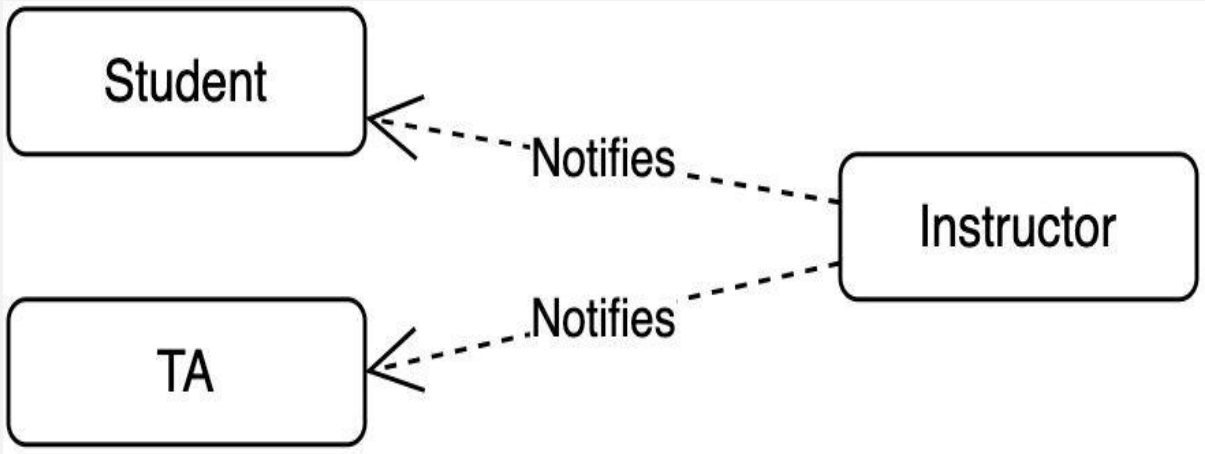
Provides a defines a one-to-many dependency between objects.

OBSERVER PATTERN

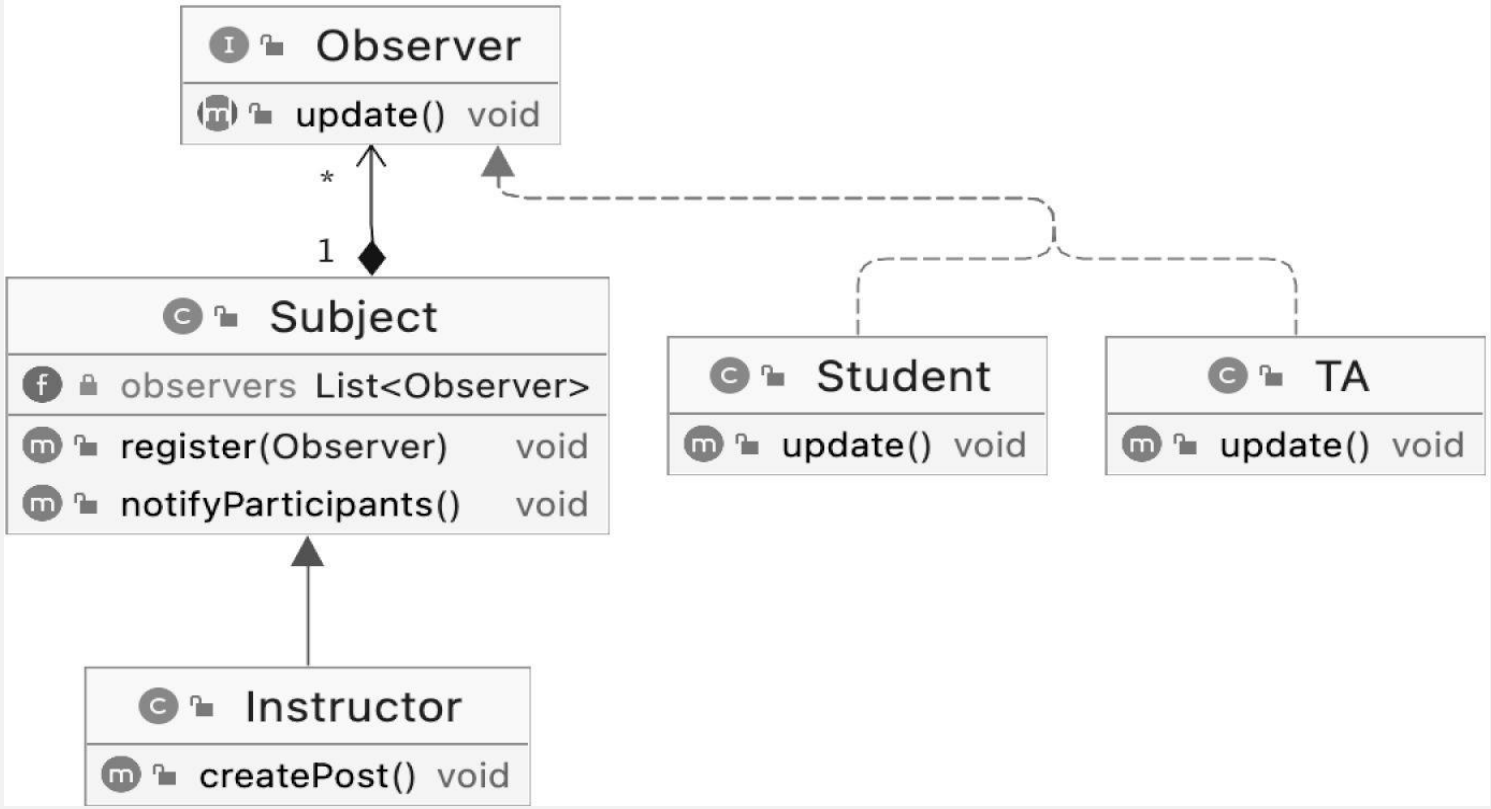
- The Observer pattern involves defining an abstract Subject class and an abstract Observer class
  - Subject maintains a list of Observers and provides methods to add/remove Observers
  - Observer class defines an update() method called by the Subject when its state changes
- When a Subject's state changes, it notifies all its Observers by calling their update() method
  - Observers can then query the Subject for any information they need to maintain consistency with the Subject's state
- Advantages
  - Loosely coupling objects, makes the system more flexible and maintainable
  - Providing a consistent way for objects to be notified of state changes
  - Allowing for dynamic addition and removal of Observers at runtime

SCENARIO

You are working on a university discussion forum where instructors create posts. Students and TAs want to get notified about any new post.



SUGGESTED OBSERVER IMPLEMENTATION



SUGGESTED IMPLEMENTATION

- Declare the observer interface. At a bare minimum, it should declare a single update method
- Declare the subject interface and describe a pair of methods for adding a observer object to and removing it from the list. Remember that subjects must work with observers only via the observer interface.
- Decide where to put the actual observers' list and the implementation of observer's methods.

SUGGESTED IMPLEMENTATION

- Create concrete subject classes. Each time something important happens inside a subject, it must notify all its observers.
- Implement the update notification methods in concrete observers classes.
- The client must create all necessary observers and register them with proper subject.

EXERCISE

- Let's imagine you have a weather station which has some sensors, the sensors measure the temperature, wind, humidity, etc. and update their state. When the weather station changes we want to notify that change to different displays (smartphone and a physical display)

- Implement the behavior of the weather station using the Observer pattern

```
def main():

    # The client code.

    subject = WeatherStation()

    phone = Phone_display()
    subject.attach(phone)

    physical = Physical_display()
    subject.attach(physical)

    subject.notify()

    subject.detach(physical)

    subject.notify()
```

```
Subject: New update ----> Notifying observers...
Phone_display: Reacted to the event
Physical_display: Reacted to the event
Subject: New update ----> Notifying observers...
Phone_display: Reacted to the event
```

CONCLUSION

- **Design patterns** are reusable solutions to commonly occurring problems in software design.
- **Anti-patterns** are common mistakes or poor design decisions that can lead to low-quality and difficult-to-maintain software systems
- There are three types of design patterns: **creational**, **structural**, and **behavioural**
- The key elements of a software design pattern are the **name**, **problem**, **solution**, and **consequences**.
- Common anti-patterns are the **BBoM**, **god object**, **golden hammer** and **stove-pipe**
- Anti-patterns can lead to issues such as high coupling, low cohesion, and poor maintainability
- Design patterns can be powerful tools for creating high-quality and flexible software systems, while anti-patterns can be a major obstacle to achieving these goals.

SUMMARY

DESIGN PATTERNS SUMMARY

- **Singleton**:
  - **Use case**: When you need exactly one instance of a class that is accessible globally
  - **Problem it solves**: Ensures a class has only one instance and provides a global point of access to it
  - **Advantages**:
    - Controlled access to a sole instance
    - Prevents inconsistent states across various instances
  - **Disadvantages**:
    - Introduces a global state
    - Might have problems in multi-threads scenarios

DESIGN PATTERNS SUMMARY

- **Factory Method**:
  - **Use case**: When a class cannot anticipate the type of objects it needs to create. These objects are created based on conditions at runtime
  - **Problem it solves**: Defines an interface (factory) for creating an object, (product) but lets the subclasses decide which concrete class of the object to instantiate.
  - **Advantages**:
    - Avoid tight coupling between the factory and the concrete products
    - Follows the SRP and OCP
  - **Disadvantages**:
    - Can lead to many subclasses
    - The code may become more complex

DESIGN PATTERNS SUMMARY

- **Abstract Factory**:
  - **Use case**: When you want to create different objects of related types and you want to ensure they are compatible.
  - **Problem it solves**: Provides an interface (factory) for creating families (concrete factories) of related object (products).
  - **Advantages**:
    - Isolates concrete classes and promotes consistency among products
    - Follows OCP
  - **Disadvantages**:
    - Can introduce unnecessary complexity
    - The code may become more complex to understand due to abstractions

DESIGN PATTERNS SUMMARY

- **Adapter**:
  - **Use case**: When you want to use an existing class whose interface isn't compatible with the rest of your code.
  - **Problem it solves**: Converts the interface of a class into another interface that the client expect.
  - **Advantages**:
    - Allows classes with incompatible interfaces to work together
    - Promotes reusability
  - **Disadvantages**:
    - Can introduce unnecessary complexity by adding interfaces
    - Not always as efficient as refactoring the original code

DESIGN PATTERNS SUMMARY

- **Facade**:
  - **Use case**: When you want to use provide a simple interface to a complex subsystem.
  - **Problem it solves**: Provides a simple entry point to a set of interfaces/classes in a subsystem.
  - **Advantages**:
    - Reduces dependency on outside (subsystem) code
    - Promotes loose coupling
  - **Disadvantages**:
    - Can became a good object coupled to all classes
    - Can hide useful lower level functionality

DESIGN PATTERNS SUMMARY

- **Bridge**:
  - **Use case**: When you want to split a large class or a set of closely related classes into two separate hierarchies
  - **Problem it solves**: Provides object composition so that we prevent the explosion of a class hierarchy by transforming it into several related hierarchies.
  - **Advantages**:
    - Client code works with high level abstractions
    - Follows the OCP, SRP
  - **Disadvantages**:
    - You might make the code more complicated

DESIGN PATTERNS SUMMARY

- **Observer**:
  - **Use case**: When you need many objects to receive updates when another object changes.
  - **Problem it solves**: Defines a one-to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically
  - **Advantages**:
    - Allows sending data to many objects in a clean way
    - Promotes loose coupling
  - **Disadvantages**:
    - Unexpected updates can cascade thought the system
    - Can lead to complex update logic

IN CLASS ACTIVITY

Which one of the design patterns would you use in the following scenarios?

1. A drawing application which draw different shapes (circle, rectangle, ….) based on the user input.
2. My e-commerce application wants to implement an e-commerce checkout system providing a single "place order" method. This system is quite complex as it needs to interact with other systems (inventory, payment, shipping, etc).
3. My application generate different UI components (buttons, text boxes, menus) based on a selected theme (light, dark, high contrast, 80's) by the user.
4. A chat room application that notifies all users when a new message is posted
5. An application which needs to export different documents (Invoices, Employees, products …) into different formats (CSV, JSON, XML, …).