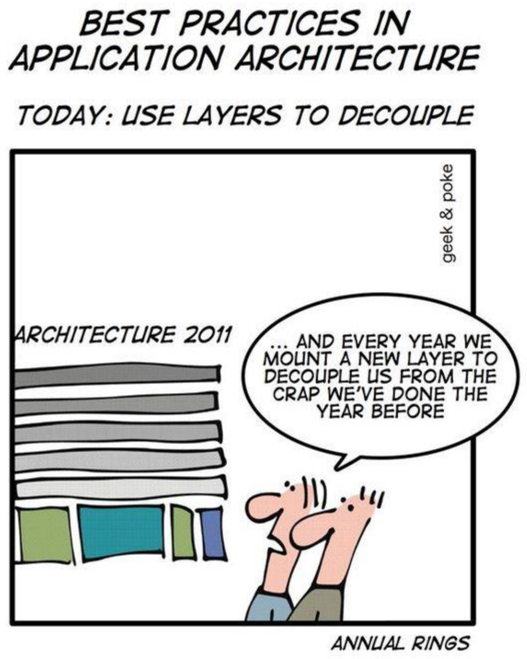
University
Mohammed VI
Polytechnic


BEST PRACTICES IN APPLICATION ARCHITECTURE
TODAY: USE LAYERS TO DECOUPLE
geek & poke
ARCHITECTURE 2011
... AND EVERY YEAR WE MOUNT A NEW LAYER TO DECOUPLE US FROM THE CRAP WE'VE DONE THE YEAR BEFORE
ANNUAL RINGS

Software Design

Pr. Imane Fouad

What is software design?

In the general sense, **design** can be viewed as a form of a problem solving process.

In the case of software the input of the design process is the requirements.

What are the basic steps of the design process?

Architectural design (also referred to as high level design and top-level design) describes how software is organized into components.

Detailed design describes the desired behavior of these components.

What is the outcome of the design process?

The output of these two processes is a set of models and artifacts that record the major decisions that have been taken, along with an explanation of the rationale for each nontrivial decision.

Pattern Language

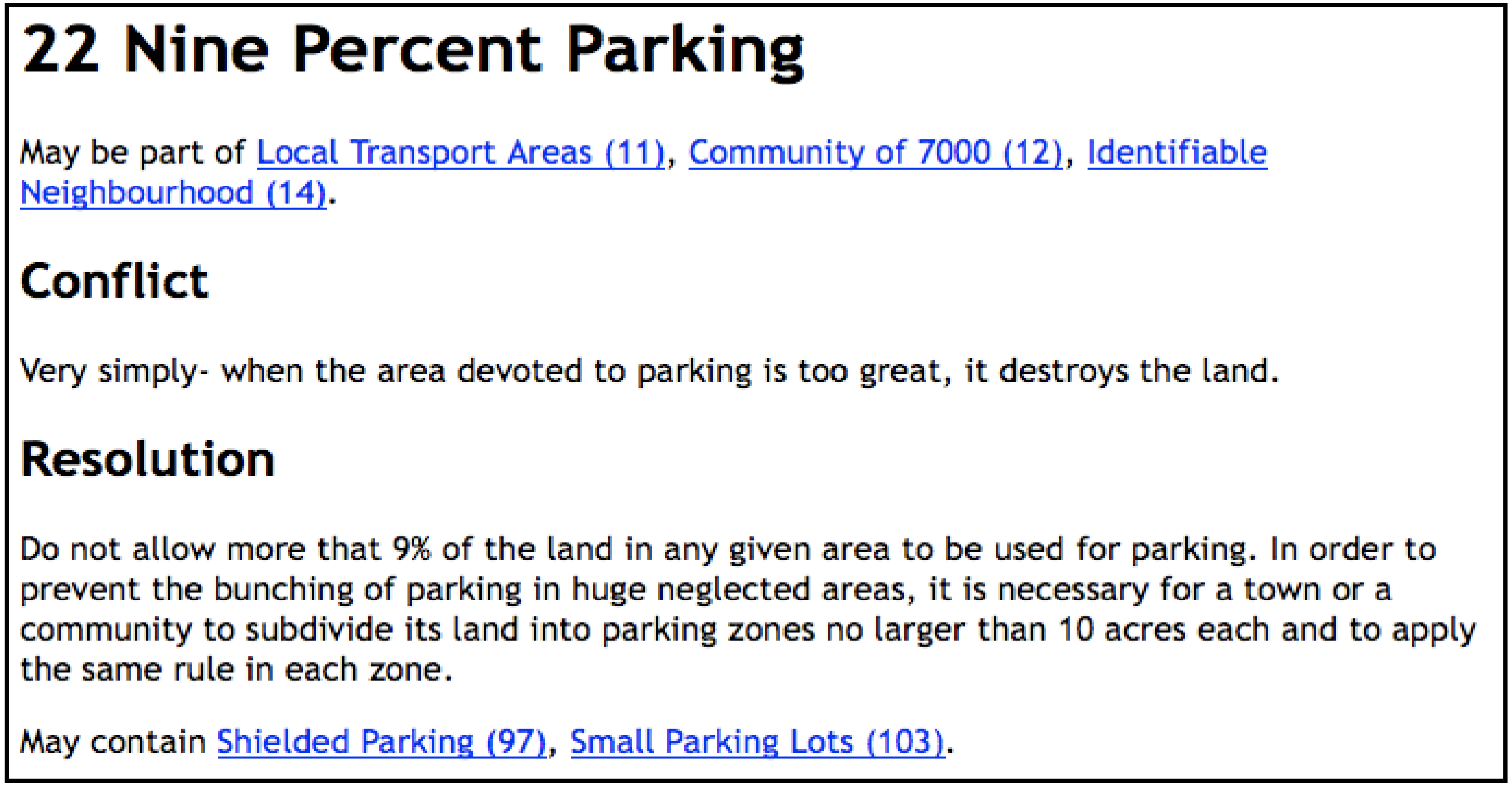
Christopher Alexander says,

"Each **pattern** describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"

Design patterns - History

- Concept originated with **Christopher Alexander**, an architect.
- Initially applied to **architecture**: buildings, towns.
- “A Pattern Language: Towns, Buildings, Construction”, Christopher Alexander, 1977
- Later adapted to **software development** to solve recurring programming challenges.

“A Pattern Language” examples



22 Nine Percent Parking

May be part of Local Transport Areas (11), Community of 7000 (12), Identifiable Neighbourhood (14).

Conflict

Very simply- when the area devoted to parking is too great, it destroys the land.

Resolution

Do not allow more that 9% of the land in any given area to be used for parking. In order to prevent the bunching of parking in huge neglected areas, it is necessary for a town or a community to subdivide its land into parking zones no larger than 10 acres each and to apply the same rule in each zone.

May contain Shielded Parking (97), Small Parking Lots (103).

Architecture Patterns

- High-level patterns: view of entire system
- Multiple overlapping views possible

Examples:

- Model-View-Controller
- Client-Server architecture
- Layered architecture
- Repository pattern

The Model-View-Controller Pattern

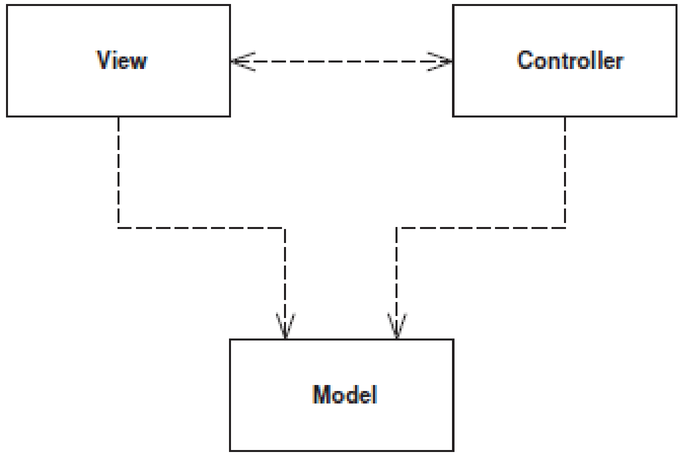
- Sometimes the same data may have to be accessed **under different contexts**.

- Each context may require a different interface.

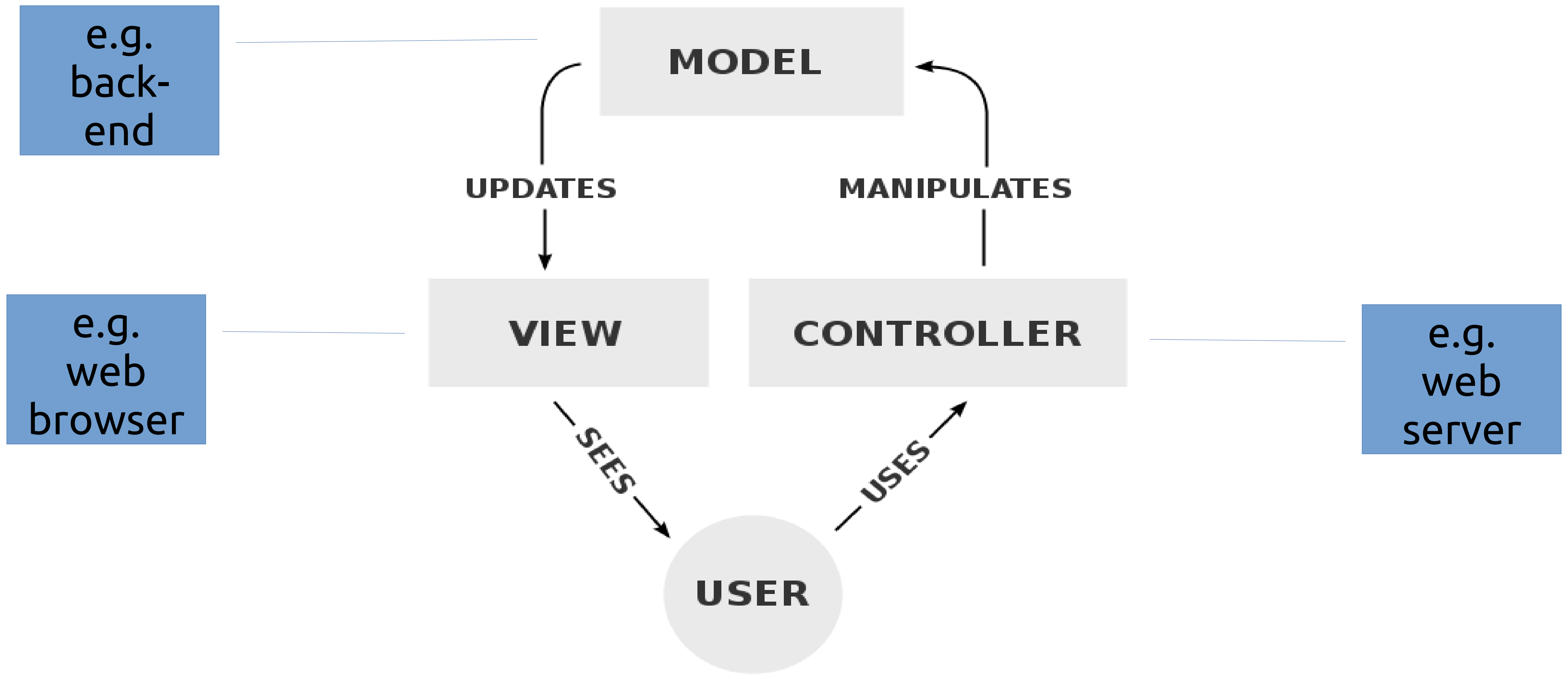
- Even within an interface there may be a desire to see different views of the data.

Model-View-Controller (MVC)

- Suitable for UI applications
- 3 logical components:
  - Model: manages data & associated operations
  - View: defines/manages presentation to user
  - Controller: manages user interaction
- Advantages:
  Data can change independently from presentation
- Disadvantages:
  Can cause extra complexity for simple data models



Model-View-Controller



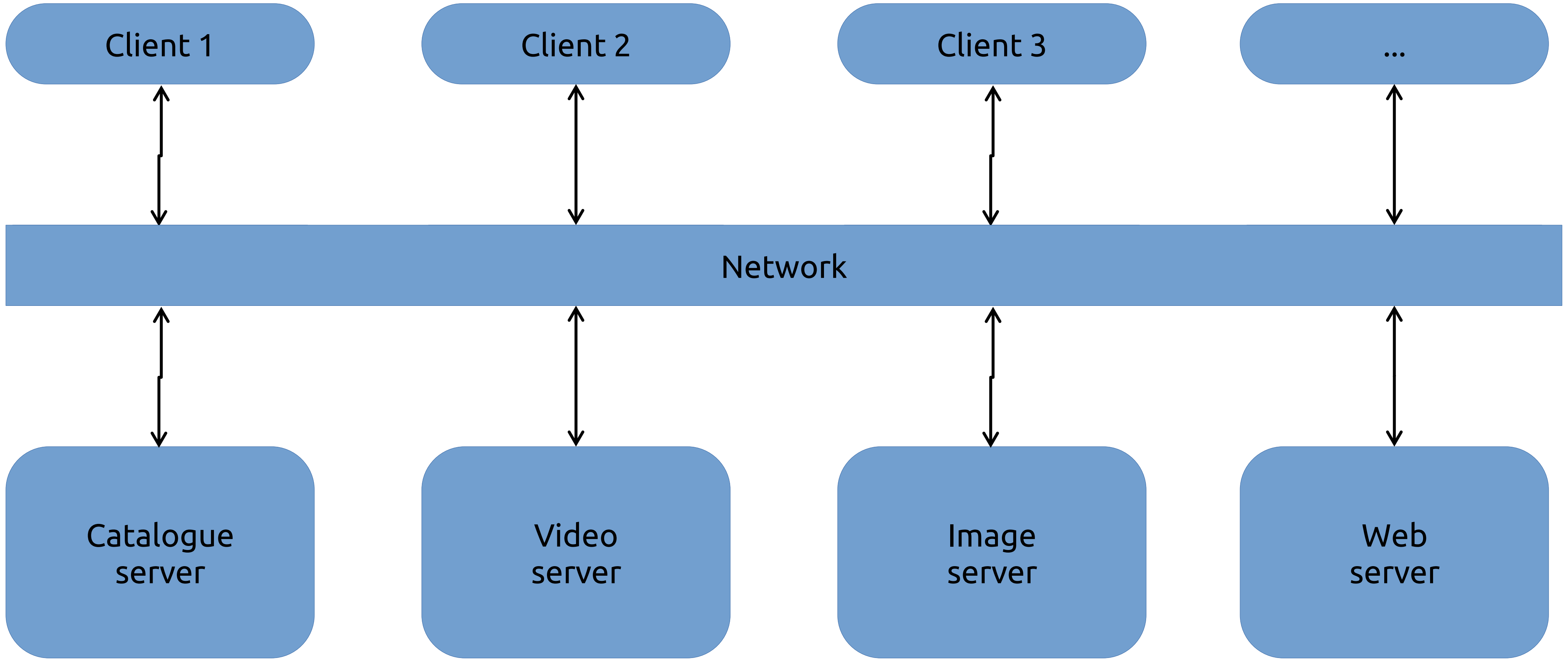
Model-View-Controller Pattern (3)

- Implementing different parts that are decoupled (minimized dependencies) provides many benefits:
    - Parts can be changed independently (e.g., UI vs. data)
    - Different experts can work on different parts
    - Single data source supports multiple views

Client-server architecture

- Functionality split into services
- Every service provided by one (logical) server
  - Accessing shared data from multiple locations
  - Load balancing → replicate servers
- Advantages:
  - Distributed, network-transparent architecture
- Disadvantages:
  - Every server → single point of failure

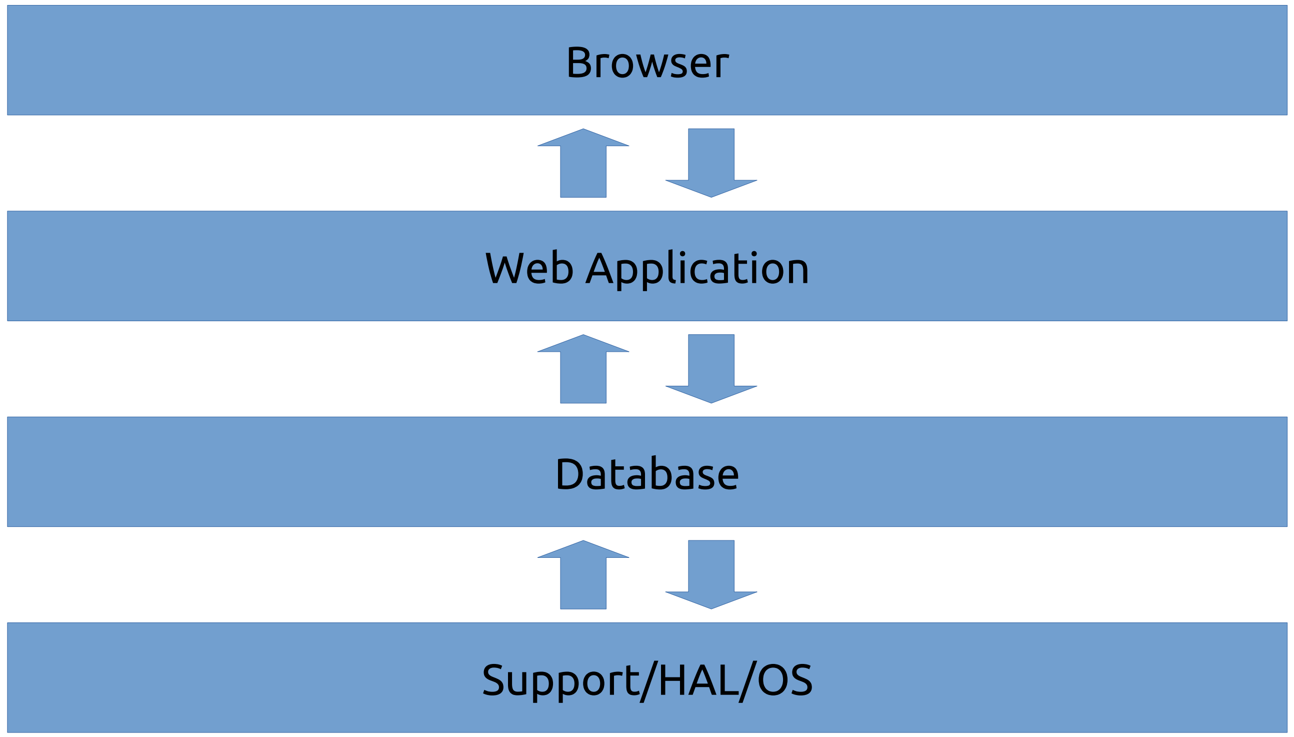
Client-server architecture



Layered architecture

- System as a stack of interconnected layers
  - Layers only communicate with neighbours
- Often used for network protocols
- Advantages:
  - Individual layers can be replaced
- Disadvantages:
  - Clean separation can be difficult

Example: layered UI architecture



Design patterns

What is a Design Pattern?

A problem that someone has **already solved**

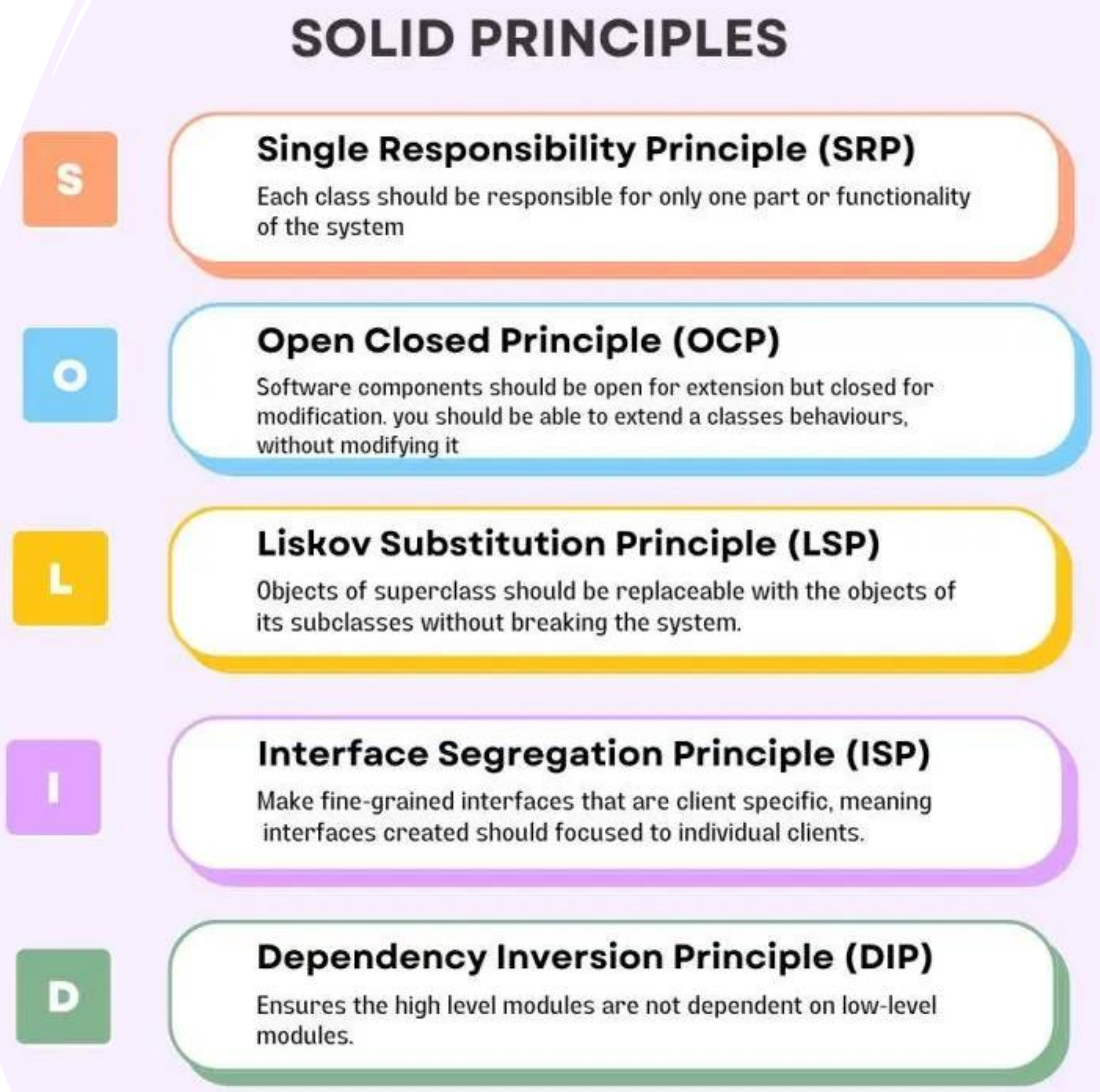
A **model or design** we can use as a guide

Formally: **“A proven solution to a common problem in a specific context”**

Why study Design Patterns

- **Improve Code Reusability –** Avoid reinventing the wheel, use tried-and-tested solutions.
- **Enhance Maintainability –** Makes code **easier to read, update, and debug**.
- **Promote Best Practices –** Encourages clean coding principles (SOLID, modularity).
- **Increase Team Collaboration –** Common language among developers (e.g., "Use Factory" = instantly understood).
- **Improve Flexibility and Scalability –** Helps you write code that adapts to future changes.

What makes a good software



SOLID PRINCIPLES

S — **Single Responsibility Principle (SRP)**
Each class should be responsible for only one part or functionality of the system

O — **Open Closed Principle (OCP)**
Software components should be open for extension but closed for modification. you should be able to extend a classes behaviours, without modifying it

L — **Liskov Substitution Principle (LSP)**
Objects of superclass should be replaceable with the objects of its subclasses without breaking the system.

I — **Interface Segregation Principle (ISP)**
Make fine-grained interfaces that are client specific, meaning interfaces created should focused to individual clients.

D — **Dependency Inversion Principle (DIP)**
Ensures the high level modules are not dependent on low-level modules.

Single Responsibility Principle

- **Definition:**
  Each class should only have one job or responsibility. It should only focus on one thing.

- **Why use it?**
  When a class has only one task, it's easier to manage. You can make changes to one part of the code without affecting other parts.

Open/Closed Principle (OCP)

- **Definition:**
  Your code should be open for extension but closed for modification.

- **Why use it?**
  If you modify existing code often, you might break something that was working. Instead, it’s better to extend it when adding new features.

Liskov Substitution Principle (LSP)

- **Definition:**
  Subclasses should be able to replace their parent classes without breaking the code.

- **Why use it?**
  It ensures that subclasses can be used anywhere the parent class is used, without causing bugs or errors.

Interface Segregation Principle (ISP)

- **Definition:**
  A class should not be forced to implement methods it doesn’t use. Instead of having one large interface, split it into smaller, more specific interfaces.

- **Why use it?**
  If a class has to implement methods it doesn’t need, the code becomes bloated and harder to maintain.

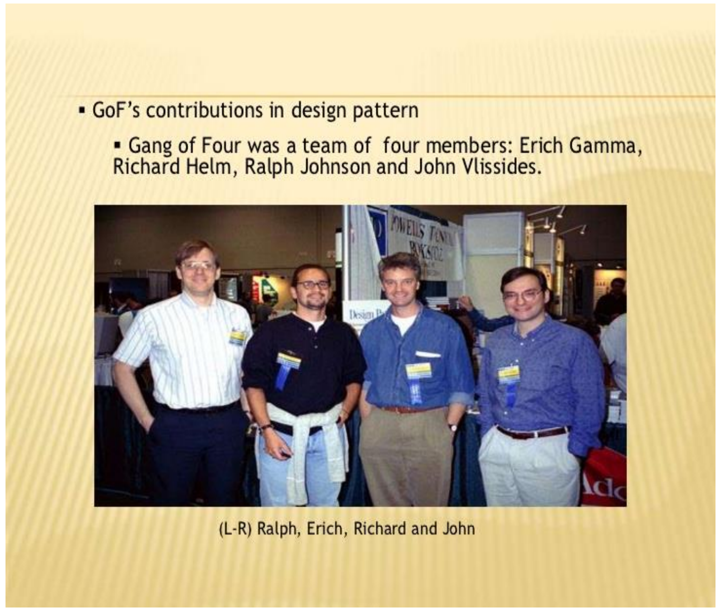
Dependency Inversion Principle (DIP)

- **Definition:**
  High-level classes should not depend on low-level classes. Both should depend on abstractions (like interfaces), not on concrete implementations.

- **Why use it?**
  When you directly depend on specific classes, it’s harder to swap them out if you need to make changes. Using abstractions makes your code more flexible and easier to maintain.

Gang of Four (GoF) patterns for OO design

GoF Patterns



- GoF's contributions in design pattern
  - Gang of Four was a team of four members: Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides.

(L-R) Ralph, Erich, Richard and John

GoF say:

The **design patterns** are descriptions of communicating objects and classes that are customized to solve a **general design problem** in a particular context.

Design Patterns: Elements of Reusable Object Oriented Software," Gamma, Helm, Johnson, Vlissides, Addison-Wesley, 1995

Design Patterns: Programming Languages

Aimes towards languages that have language level support for Object Oriented Programming.

Not exclusively, But it would be easier to apply with OOP!

Different OOP languages have different mechanisms for applying patterns.

The Gang of For

- Defines a catalog of different design patterns.

- Three different types:
  - **Creational Patterns** – Deal with **object creation**, making it easier and more flexible.
  - **Structural Patterns** – Concerned with how classes and objects are composed to form larger structures
  - **Behavioral Patterns** – Deal with **object interaction and responsibility**, improving communication between objects.

Creational Design Patterns

- Patterns that deal with **object creation**.
- Aim to **decouple the client code from concrete classes**.

- **Purpose:**
  - Make systems **flexible and reusable**.
  - Avoid **tight coupling** between classes and objects.
  - Provide a **single point of control** for object creation.

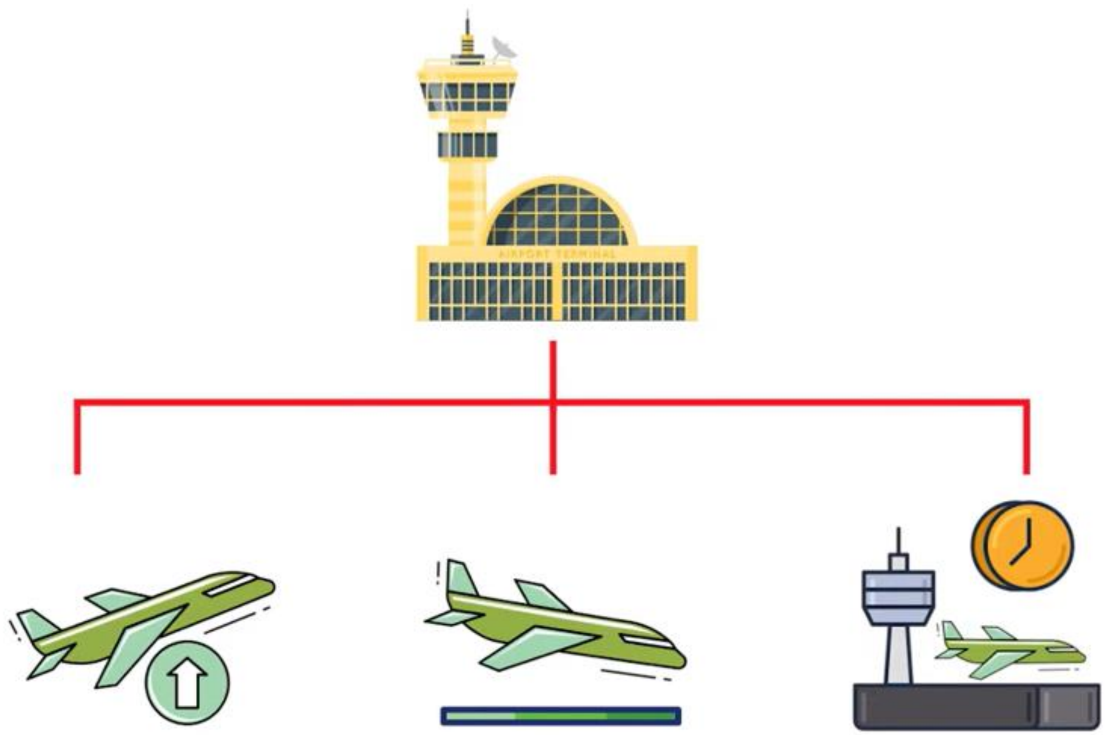
Classification of GoF patterns

Creational	**Structural**	**Behavioral**
Factory Method	Adapter	Interpreter
Abstract Factory	Bridge	Template Method
Builder	Composite	Chain of Responsibility
Prototype	Decorator	Command
Singleton	Flyweight	Iterator
	Facade	Mediator
	Proxy	Memento
		Observer
		State
		Strategy
		Visitor

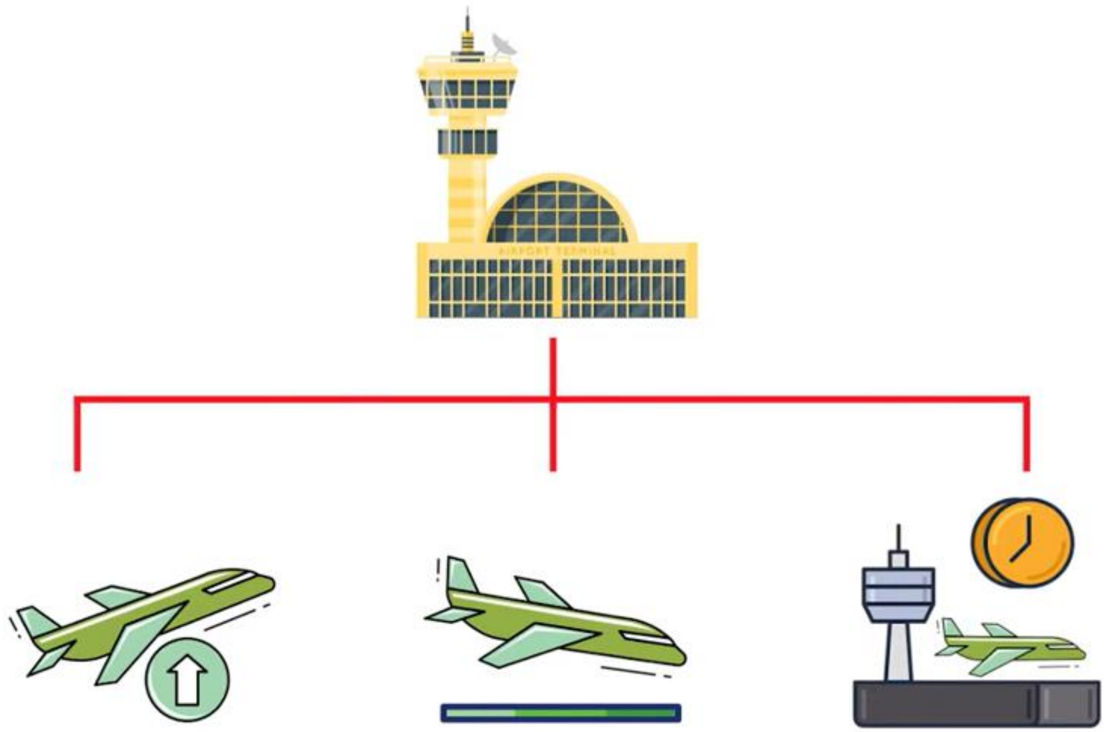
Quiz

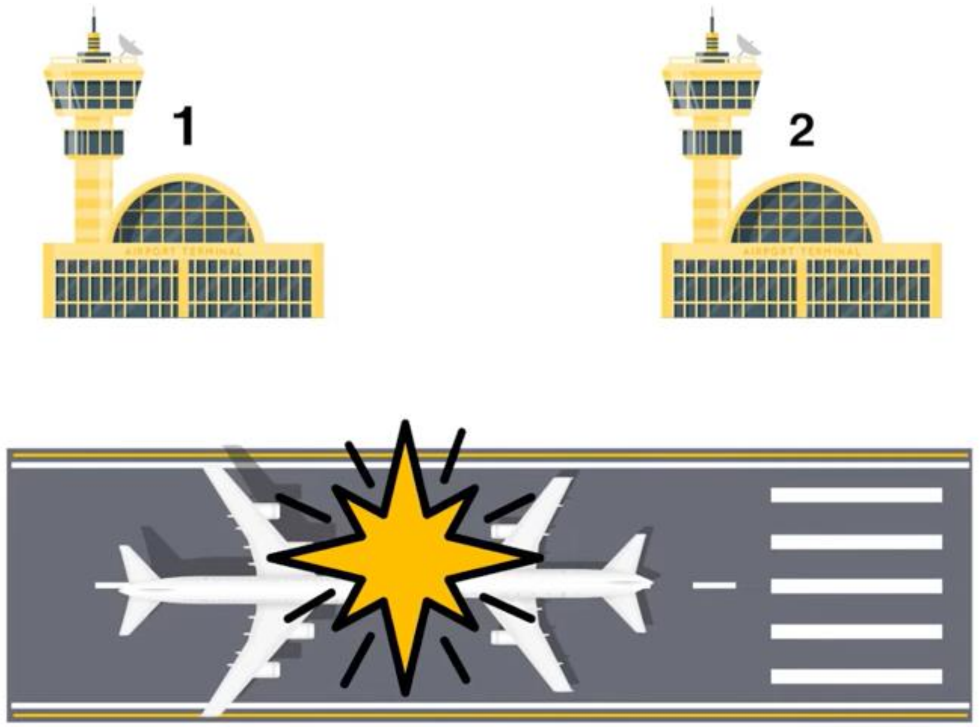
Singleton

Introduction



Introduction





Introduction

```
class ControlTower:
	def __init__(self):
		print('Initializing ControlTower...')
```

Introduction

```
class ControlTower:
     def __init__(self):
         print('Initializing ControlTower...')

tower1 = ControlTower()
tower2 = ControlTower()

print(tower1 is tower2)
```

False

Singleton Pattern

- **Intent:** Ensure a class has only one instance and provide a global point of access to it.
- **Problem:** How can we guarantee that **one and only one instance** of a class is created?
- **Solution:**
  - The class itself is responsible for **creating its single instance**.
  - Provide an **access method** that always returns the same instance

Singleton – Participants

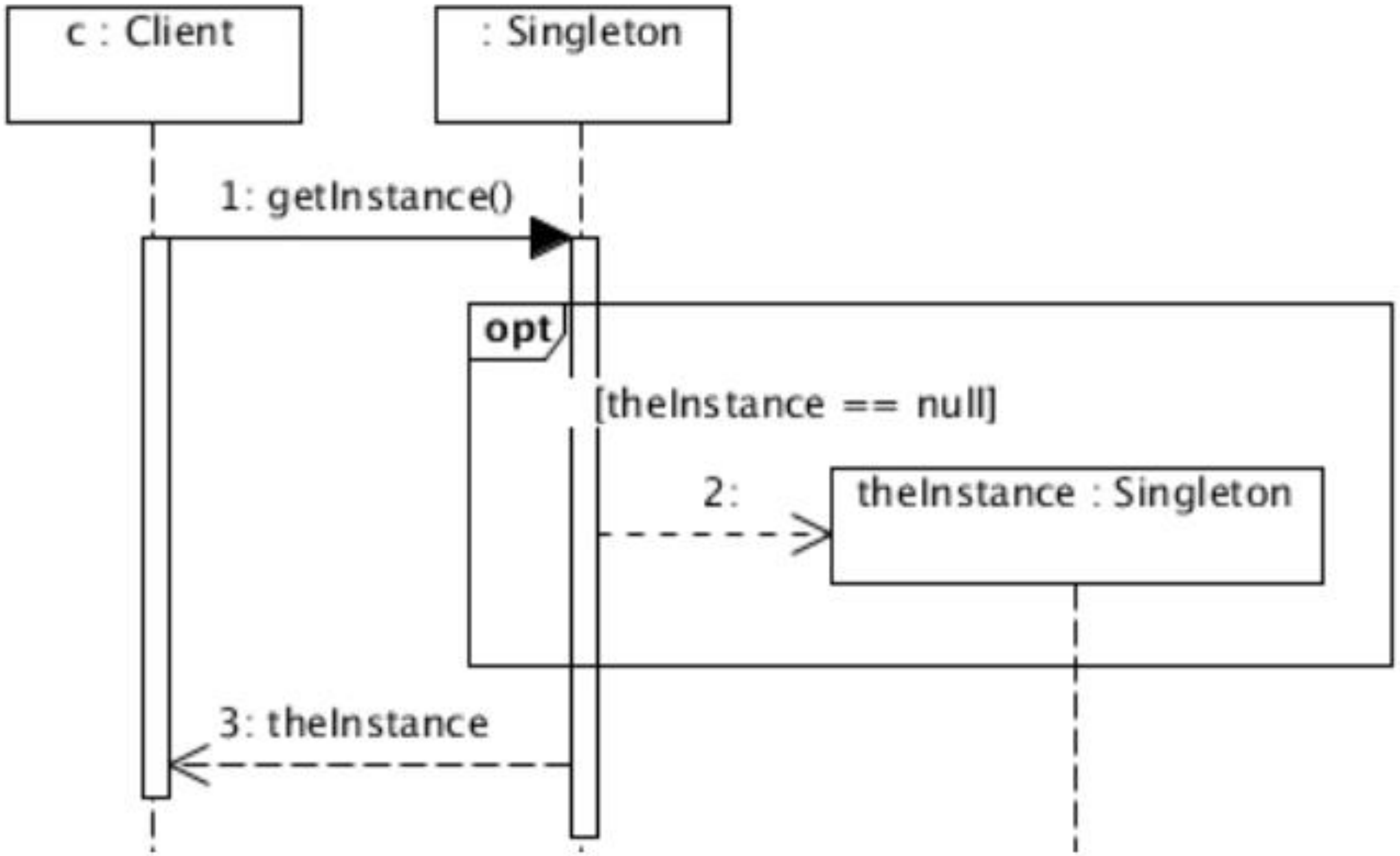
1. Singleton

- Ensures only **one instance of the class** is ever created.
- May be responsible for **creating and managing** its own single instance.

2. Client

- Requests access to the Singleton.
- Always receives the **same shared object**, not a new one.

Singleton sequence diagram



Singleton: Basic Implementation

```
class ControlTower:
    _instance = None

    def __new__(cls):
        if cls._instance is None: # check if an instance already exist
            cls._instance = super().__new__(cls)
            print('Creating the one and only ControlTower...')
        return cls._instance


tower1 = ControlTower()
tower2 = ControlTower()

print(tower1 is tower2)
```

new : controls how we create an instance

True

Singleton: Solution(2)

```
# Metaclass to enforce singleton behavior
class SingletonMeta(type):
    _instances = {}  # Dictionary to hold instances

    def __call__(cls, *args, **kwargs):
        if cls not in cls._instances:
            # If no instance exists, create one and store it
            instance = super().__call__(*args, **kwargs)
            cls._instances[cls] = instance
        return cls._instances[cls]

class ControlTower(metaclass=SingletonMeta):
                    …
```

Examples of Singleton

Software Examples:

- **Logger** – One global logging object used throughout the application.
- **Database Connection**– Avoids creating multiple connections unnecessarily.

Why These Need to Be Singletons?

- Centralized control
- Avoid conflicts and inconsistent state
- Saves memory and system resources

Singleton – Limitations

- The main limitation of the Singleton pattern is that it **allows only a single instance** of the class to exist, whereas most practical applications often require **multiple instances** to handle different tasks or data.

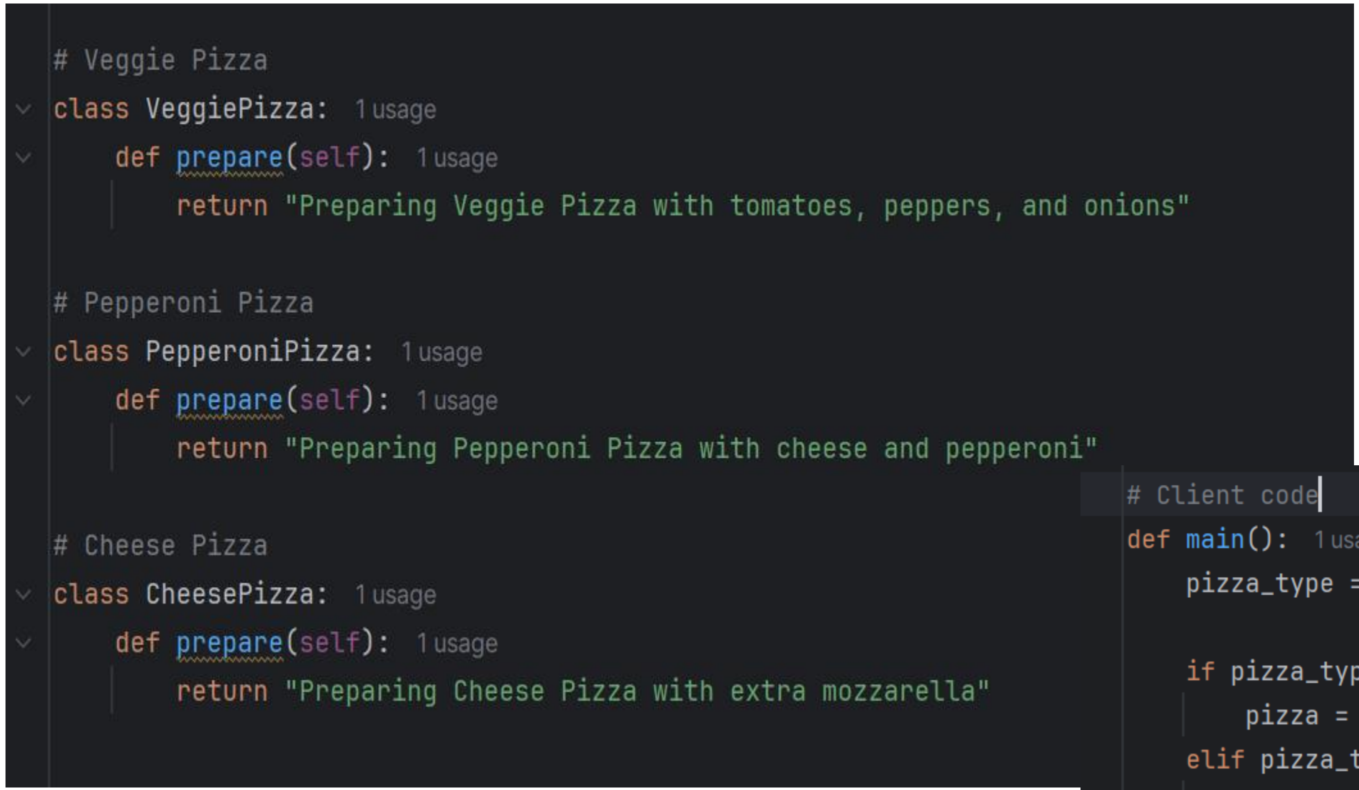
- Additionally, in a multithreaded environment, multiple threads **compete to access the single instance**, which can lead to **performance bottlenecks** and degrade overall application efficiency.

Factory Method

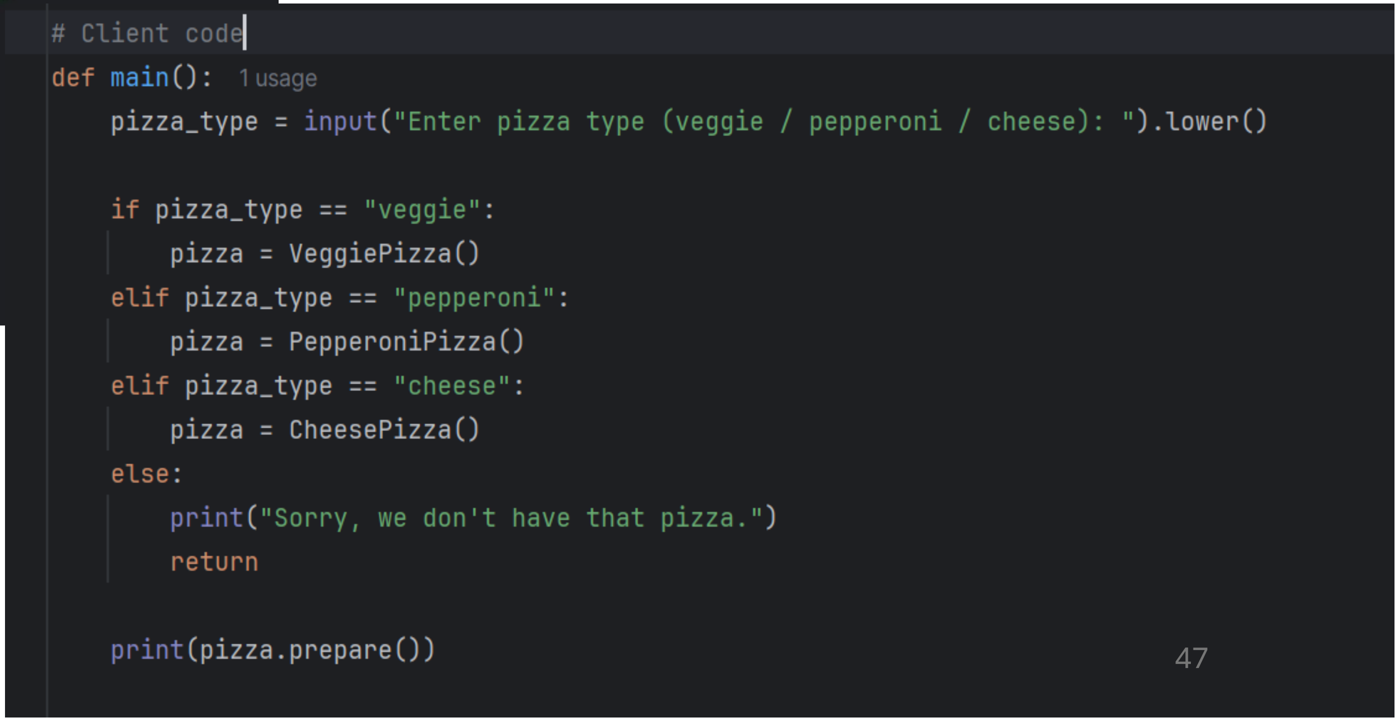
PizzaChop

You are building an app for PizzaChop, which serves three different types of pizza: *VeggiePizza, PepperoniPizza, and CheesePizza*. Each type of pizza has its own preparation steps. Depending on the user’s request (input), the app should create and prepare the corresponding pizza

PizzaChop



```
# Veggie Pizza
class VeggiePizza:  1 usage
    def prepare(self):  1 usage
        return "Preparing Veggie Pizza with tomatoes, peppers, and onions"

# Pepperoni Pizza
class PepperoniPizza:  1 usage
    def prepare(self):  1 usage
        return "Preparing Pepperoni Pizza with cheese and pepperoni"

# Cheese Pizza
class CheesePizza:  1 usage
    def prepare(self):  1 usage
        return "Preparing Cheese Pizza with extra mozzarella"
```



```
# Client code
def main():  1 usage
    pizza_type = input("Enter pizza type (veggie / pepperoni / cheese): ").lower()

    if pizza_type == "veggie":
        pizza = VeggiePizza()
    elif pizza_type == "pepperoni":
        pizza = PepperoniPizza()
    elif pizza_type == "cheese":
        pizza = CheesePizza()
    else:
        print("Sorry, we don't have that pizza.")
        return

    print(pizza.prepare())
```

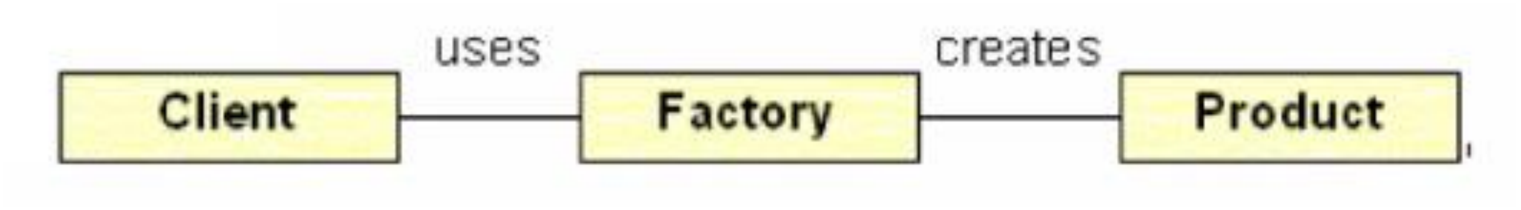
Problems

- Client code directly creates objects
- Tight coupling between client and concrete classes
- Hard to extend or add new products
- Modifying existing code increases risk of errors
- Reduces maintainability and flexibility

Factory Method

Intent

The Factory Method pattern encapsulates the process of creating objects, separating object creation from usage. This allows client code to work with abstract interfaces without knowing the concrete classes being instantiated



Forces

- We want to have a set of reusable classes which are flexible enough to be extended.

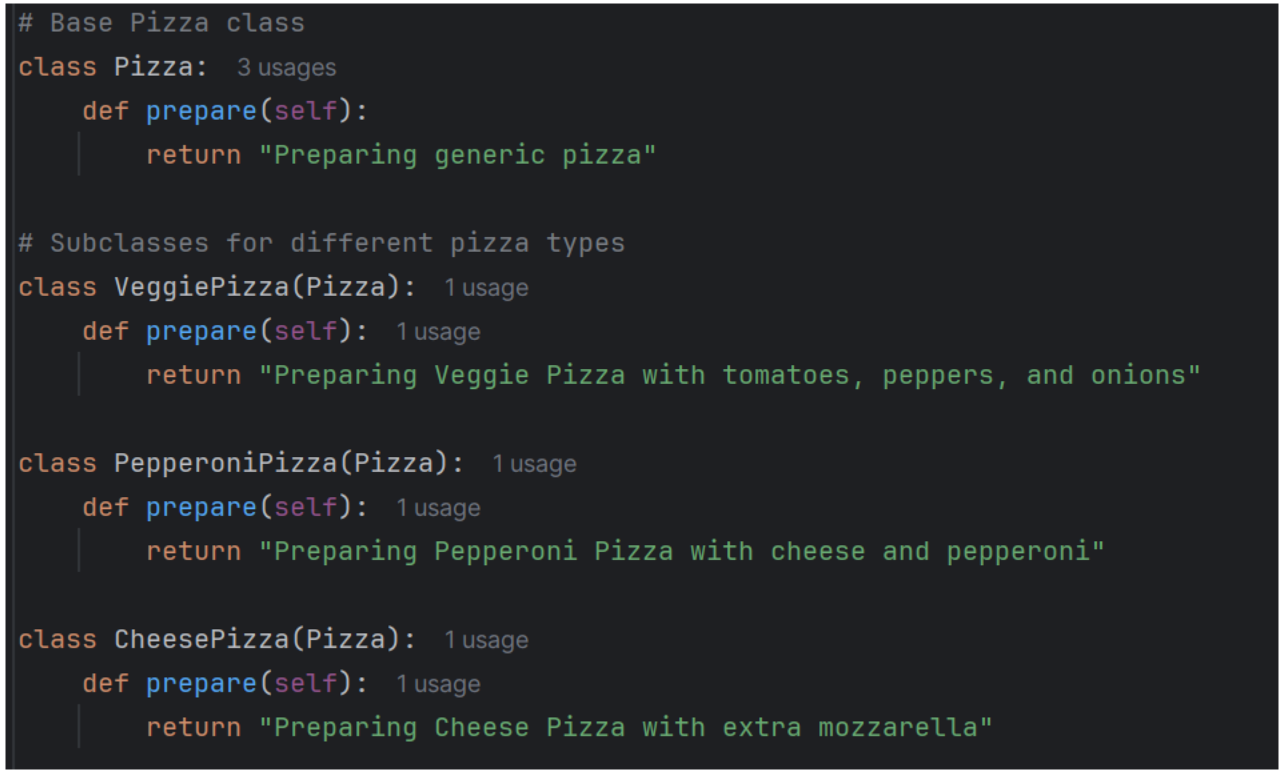
- The client does not know the type of object that needs to be created in advance and still wants to perform operations on them.

Applicability

Factory Method is needed when:

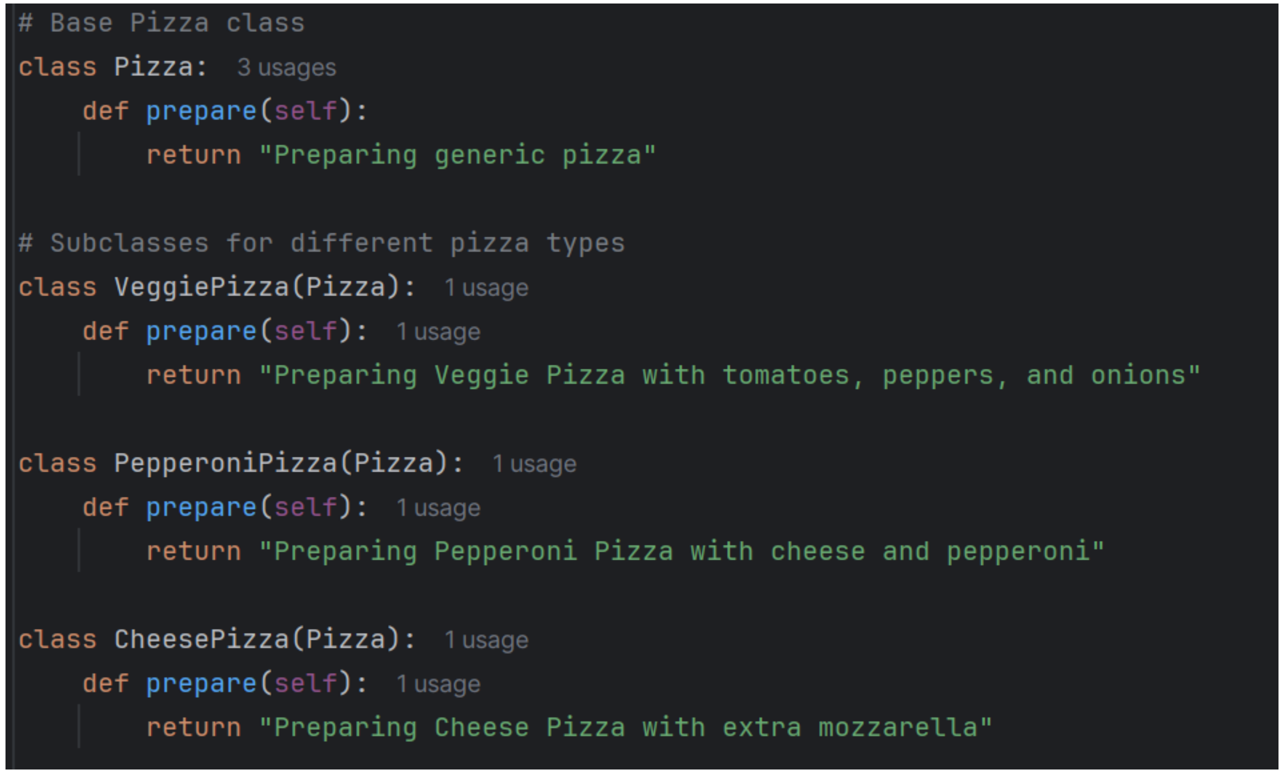
- A class can’t anticipate the types of objects it must create.
- A class wants its subclasses to specify the object to create.
- The designer wants to localize knowledge of helper sub classes.

PizzaChop

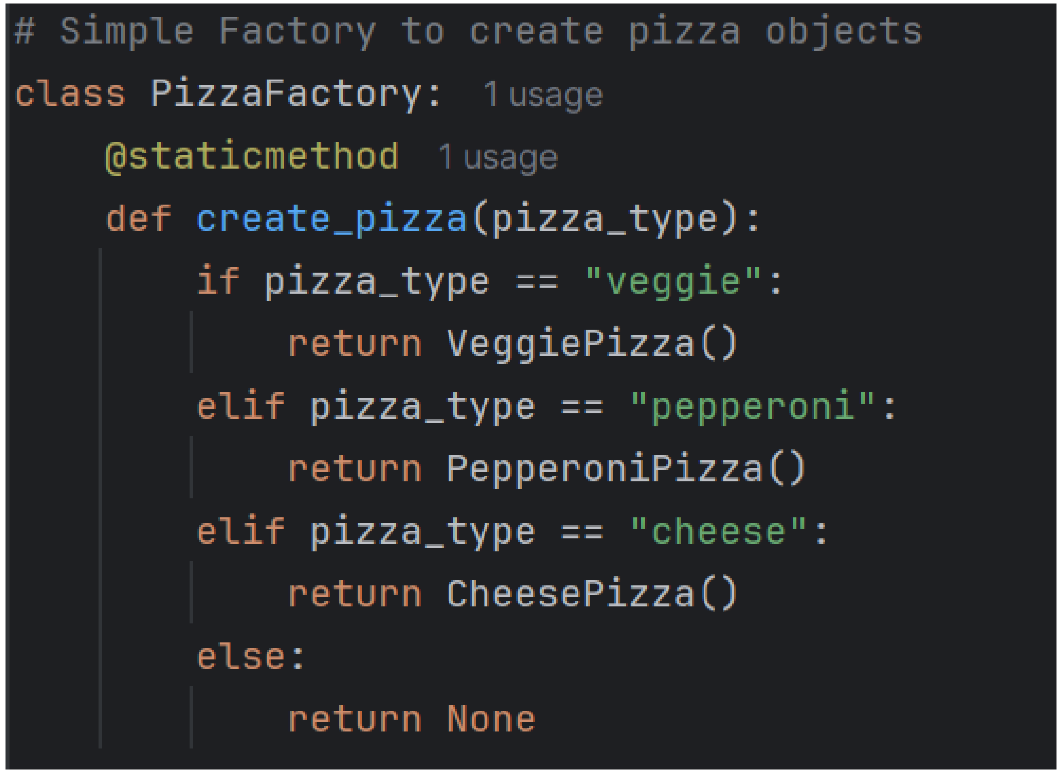


```
# Base Pizza class
class Pizza:  3 usages
    def prepare(self):
        return "Preparing generic pizza"

# Subclasses for different pizza types
class VeggiePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Veggie Pizza with tomatoes, peppers, and onions"

class PepperoniPizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Pepperoni Pizza with cheese and pepperoni"

class CheesePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Cheese Pizza with extra mozzarella"
```

PizzaChop

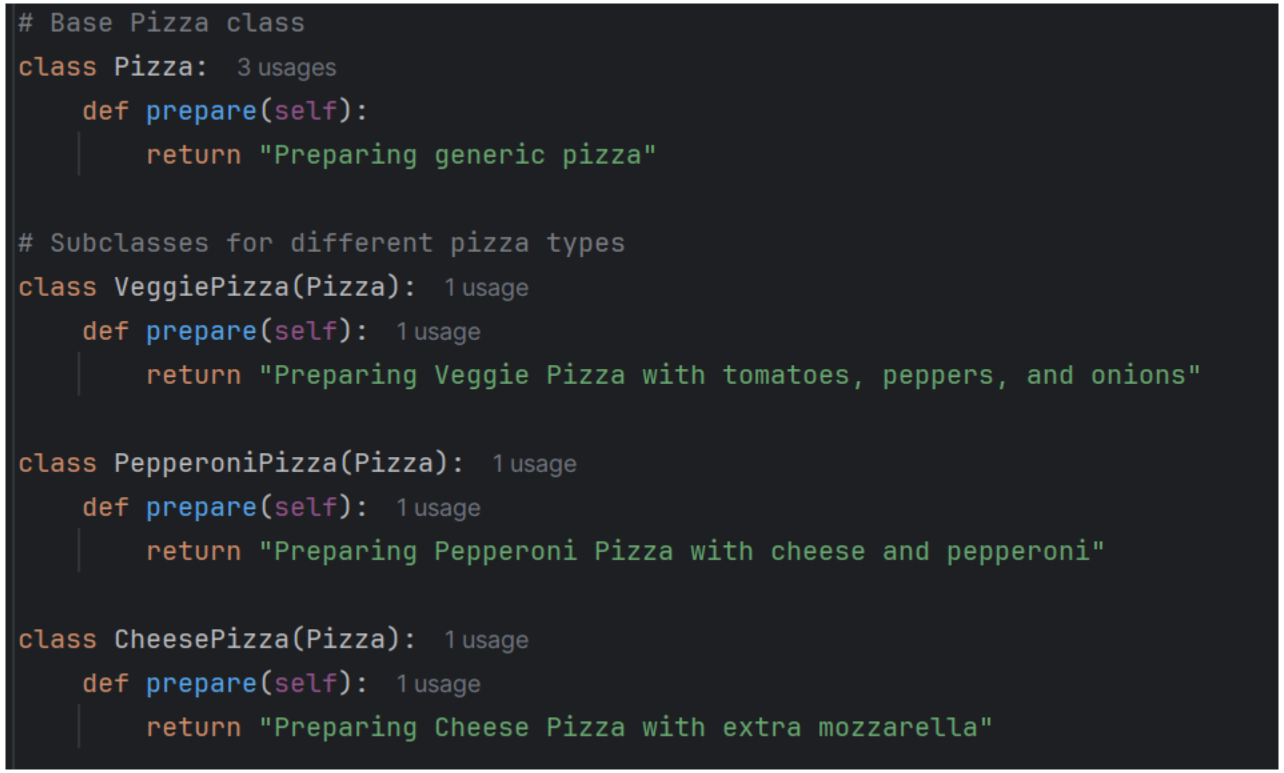


```
# Base Pizza class
class Pizza:  3 usages
    def prepare(self):
        return "Preparing generic pizza"

# Subclasses for different pizza types
class VeggiePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Veggie Pizza with tomatoes, peppers, and onions"

class PepperoniPizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Pepperoni Pizza with cheese and pepperoni"

class CheesePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Cheese Pizza with extra mozzarella"
```

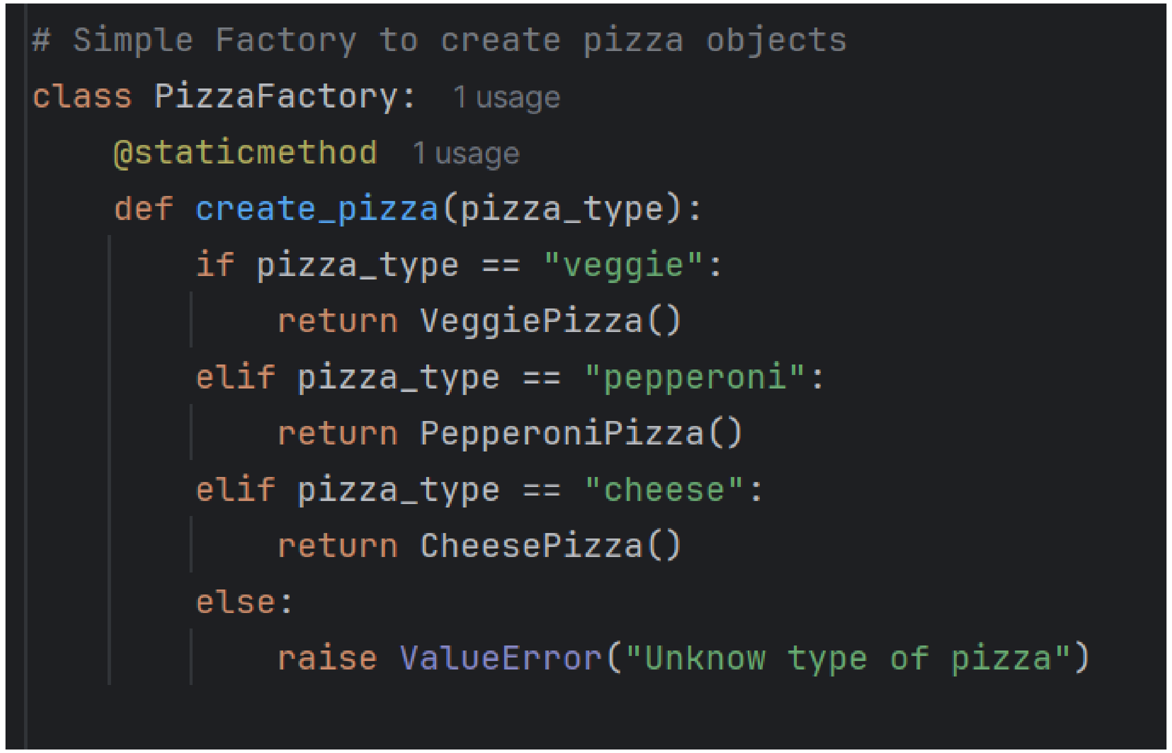


```
# Simple Factory to create pizza objects
class PizzaFactory:  1 usage
    @staticmethod  1 usage
    def create_pizza(pizza_type):
        if pizza_type == "veggie":
            return VeggiePizza()
        elif pizza_type == "pepperoni":
            return PepperoniPizza()
        elif pizza_type == "cheese":
            return CheesePizza()
        else:
            return None
```

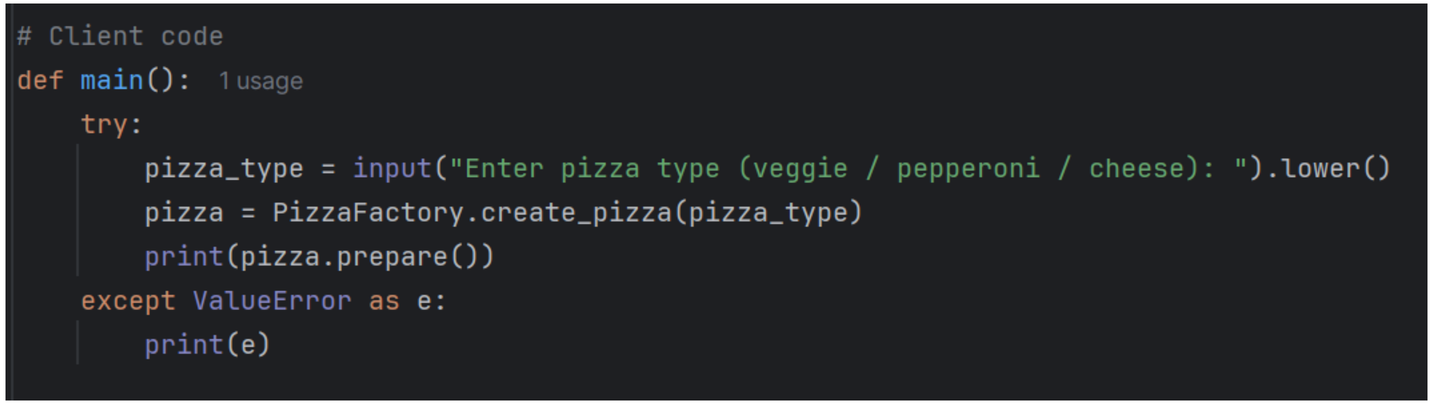
PizzaChop



```python
# Base Pizza class
class Pizza:  3 usages
    def prepare(self):
        return "Preparing generic pizza"

# Subclasses for different pizza types
class VeggiePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Veggie Pizza with tomatoes, peppers, and onions"

class PepperoniPizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Pepperoni Pizza with cheese and pepperoni"

class CheesePizza(Pizza):  1 usage
    def prepare(self):  1 usage
        return "Preparing Cheese Pizza with extra mozzarella"
```



```python
# Simple Factory to create pizza objects
class PizzaFactory:  1 usage
    @staticmethod  1 usage
    def create_pizza(pizza_type):
        if pizza_type == "veggie":
            return VeggiePizza()
        elif pizza_type == "pepperoni":
            return PepperoniPizza()
        elif pizza_type == "cheese":
            return CheesePizza()
        else:
            raise ValueError("Unknow type of pizza")
```



```python
# Client code
def main():  1 usage
    try:
        pizza_type = input("Enter pizza type (veggie / pepperoni / cheese): ").lower()
        pizza = PizzaFactory.create_pizza(pizza_type)
        print(pizza.prepare())
    except ValueError as e:
        print(e)
```

Factory Method

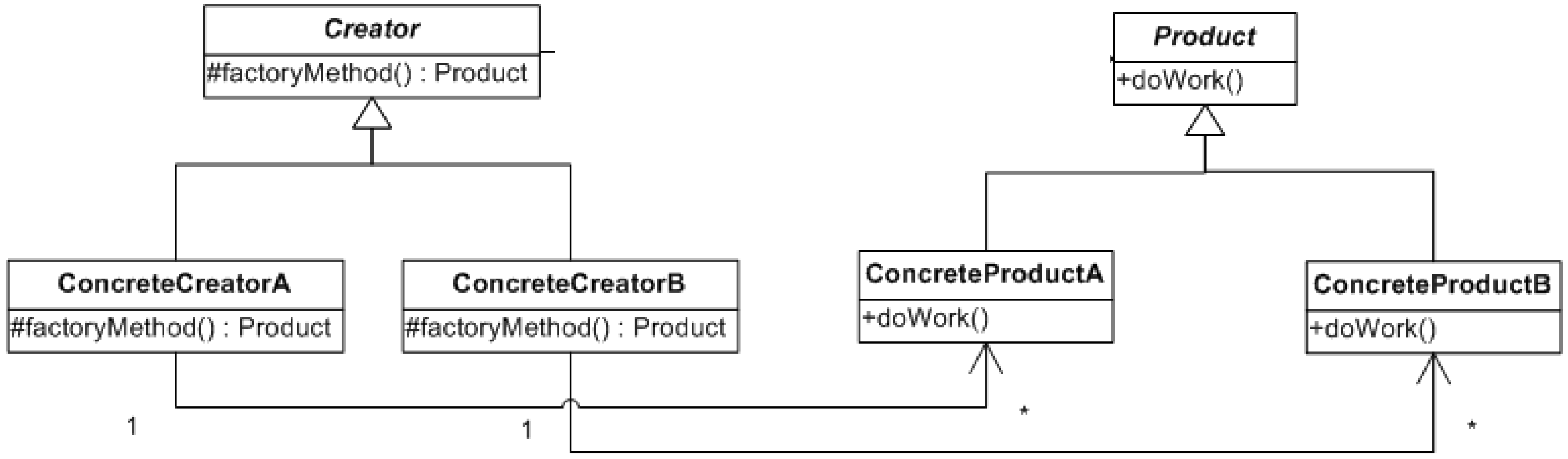
Structure

Product (Document) defines the interface of objects the factory method creates.

ConcreteProduct (LatexDocument) implements the Product interface.

Creator (DocumentStatisticsCalculator) declares the factory method, which returns an object of type Product; may call the factory method to create a Product object.

ConcreteCreator (LatexStatisticsCalculator) overrides the factory method to return an instance of a ConcreteProduct.



Consequences

- The client code deals only with the product interface, therefore it can work with any user defined Concrete Product classes (decoupling subclass details from client classes).
- New concrete classes can be added without updating the existing client code.
- It may lead to many subclasses if the product objects requires one or more additional objects.

Factory Method

Benefits

- We **avoid tight coupling** between the **Creator** and the **concrete Product objects**.
- We keep the **Product object creation code** in **one place.**
- We can **introduce new types of products** into the program without changing **Creator** code.

Liabilities

- We need to add **several small subclasses** to implement the pattern.