SOFTWARE CONCEPTS PRESENTATION

PRESENTED TO
Mahmood saqar

PRESENTED BY
Abdulrahman Elsayed,
mohammed Ibrahim
mohammed Ahmed

12\4_2024

Agenda

1. **Design patterns**
   - 1.1 definision&examples
   - 1.2 components
   - 1.3 pythonic code
2. **Architecture patterns**
   - 2.1 definision&examples
   - 2.2 components
   - 2.3 pythonic code
3. **solid principles**
   - 3.1 definision&examples
   - 3.2 components
   - 3.3 pythonic code
4. Test-Driven Development

design patterns

- It’s a general solution to common software design problems
- we have tow types (GOF pattern&Modern pattern)

(use case)

if we want to implement an undo function we can use patterns that concert with this repeated problem like

Command Pattern

Memento Pattern

1 Design patterns

A

Purpose

B

Structure

C

Benefits

D

components

E

pythonic code

1 Design patterns

A

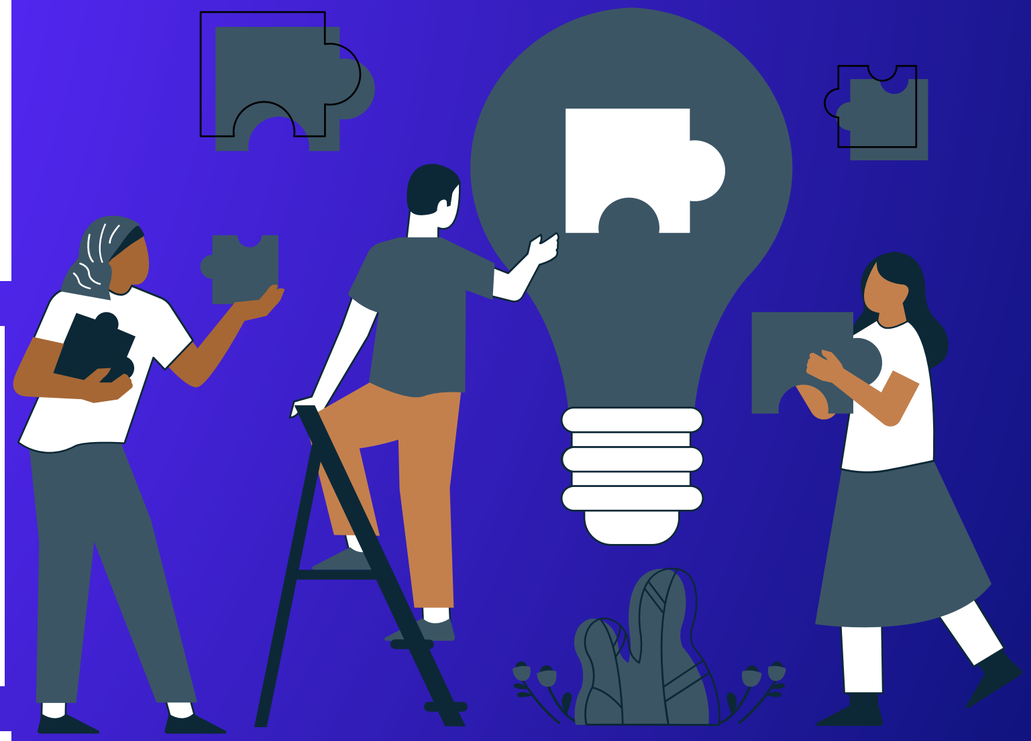
- Address recurring problems in software development.
- Promote code reusability, maintainability, and flexibility.

B

- Not concrete code implementations, but rather descriptions or templates.
- Often involve classes, objects, and interactions between them.

C

- Improve code quality by providing well-tested solutions.
- Make code easier to understand and maintain by others.
- Reduce development time by leveraging existing solutions.



1 Design patterns

COMPONENTS

Creational Patterns (Focus on object creation):

- Singleton: a solution for how can you make all instances reference to one object of that class, not many to save memory and make all calculations in one like counting the number of users (e.g., Database connection pool, Configuration manager).

- Prototype: A solution for how can avoid unnecessary and costly initialization, especially in complex objects or exact attributes but different values by providing shallow& deep copies (e.g., Prototyping game characters with different config)

- Factory Method: A solution for making your code more flexible and maintainable by separating constructing new objects and their logic about

1 Design patterns

pythonic code (singlton &prototype)

```
class count_users():
    count = 0
    def count_users(self):
        self.count += 1

    def result(self):
        return self.count

class singleton(count_users):
#lazy intialization
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance

    @staticmethod
    def creat():
        return singleton()

c1 = singleton.creat()
c1.count_users()
c2 = singleton.creat()
c2.count_users()
print(c2.result())  # Output will be
```

```
class Shape:
 def __init__(self, x, y, color):
 self.x = x
 self.y = y
 self.color = color

 def move(self, dx, dy):
 self.x += dx
 self.y += dy

 def __str__(self):
 return f"Shape(x={self.x}, y={self.y}, color={self.color})"

class Circle(Shape):
 def __init__(self, x, y, color, radius):
 super().__init__(x, y, color)
 self.radius = radius

 def __clone__(self):
 return Circle(self.x, self.y, self.color, self.radius)

class Square(Shape):
 def __init__(self, x, y, color, side_length):
 super().__init__(x, y, color)
 self.side_length = side_length

 def __clone__(self):
 return Square(self.x, self.y, self.color, self.side_length)

# Usage
circle_prototype = Circle(10, 20, "red", 5)
square_prototype = Square(30, 40, "blue", 10)

circle1 = circle_prototype.__clone__() # Deep copy using __clone__()
square1 = square_prototype.__clone__()

circle1.move(5, 3)
square1.move(-2, 1)

print(circle_prototype)
print(circle1)
print(square_prototype)
print(square1)
```

1 Design patterns

pythonic code (factory pattern)

```
class Shape:
 """Abstract Shape class."""
 def __init__(self):
  pass

 def draw(self):
  """Abstract method for drawing the shape."""
  raise NotImplementedError("Subclasses must implement draw()")

class Circle(Shape):
 """Circle class."""
 def __init__(self, radius):
  super().__init__()
  self.radius = radius

 def draw(self):
  print("Drawing a circle with radius", self.radius)

class Square(Shape):
 """Square class."""
 def __init__(self, side_length):
  super().__init__()
  self.side_length = side_length
 def draw(self):
 print("Drawing a square with side length", self.side_length)

class ShapeFactory:
 """Shape Factory class to create shapes."""
 @staticmethod
 def create_shape(shape_type):
 """Creates a shape object based on the given type."""
 if shape_type == "circle":
 return Circle(5) # Set default radius
 elif shape_type == "square":
 return Square(10) # Set default side length
 else:
 raise ValueError("Invalid shape type")

# Usage
shape_type = input("Enter shape type (circle or square): ")
shape = ShapeFactory.create_shape(shape_type)
shape.draw()

# Example output (if user enters "circle"):
# Drawing a circle with radius 5
```

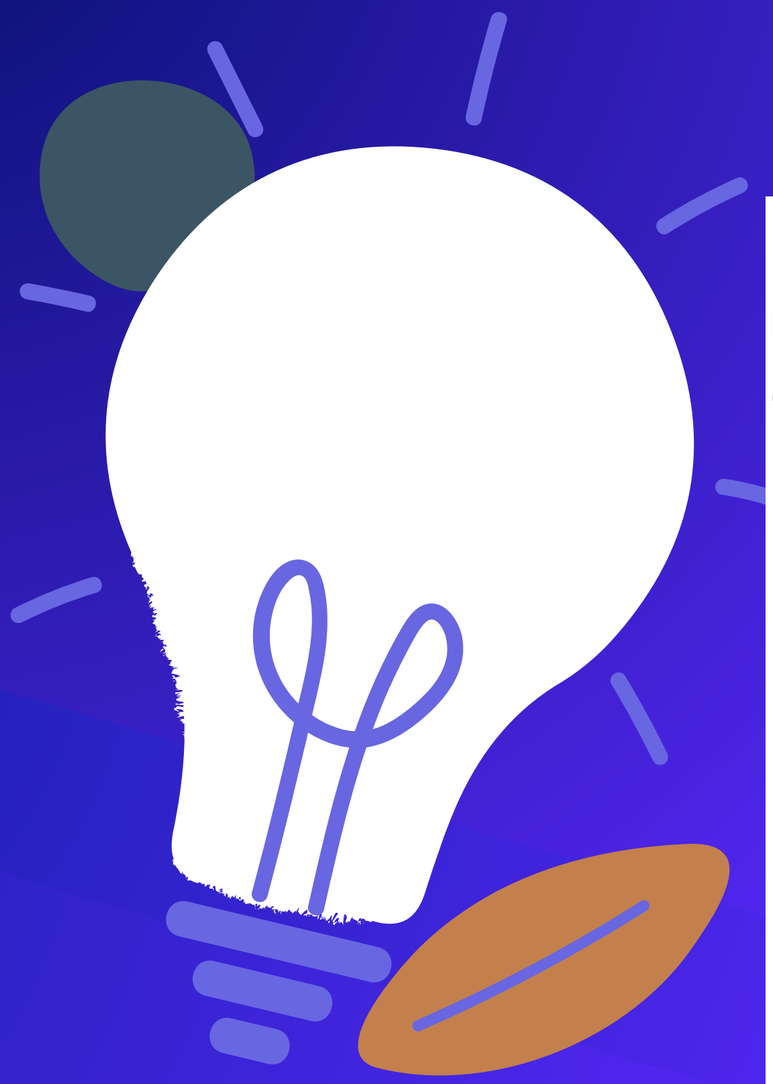
1 Design patterns

COMPONENTS

Structural Patterns (Focus on class and object structure):

1. Proxy: A solution provided to make every class make its own logic regardless if there is any condition I have to check first like N.O.Users, check pass (e.g., Implementing access control for sensitive data).
2. Decorator: A solution for decreasing dependency between classes and a number of possible combinations between objects and others (e.g., Adding logging or security checks to existing functionality)
3. Adapter: Makes incompatible interfaces work together by converting from one interface to another interface that is compatible with the client (e.g., Integrating a third-party library with a different API).
4. The Flyweight:It optimizes memory usage by sharing a common state between a large number of similar objects
5. Composite: Treats a group of objects as a single object (e.g., Representing a hierarchical file system structure).
6. Bridge: Decouples an abstraction from its implementation (e.g., Separating a drawing API from the specific shapes it can draw).

Proxy pattern



Proxy: A solution provided to make every class make its own logic regardless if there is any condition I have to check first like N.O.Users, check pass (e.g., Implementing access control for sensitive data).

- **Benefits of Proxy Pattern:**

Access Control and Management:

- **Content Filtering: Organizations can use proxy servers to restrict access to certain websites or types of content, promoting productivity and protecting employees from inappropriate content.**
- **User Management: Businesses can manage and control internet access for their employees through a central proxy server, allowing them to set policies and monitor internet usage**

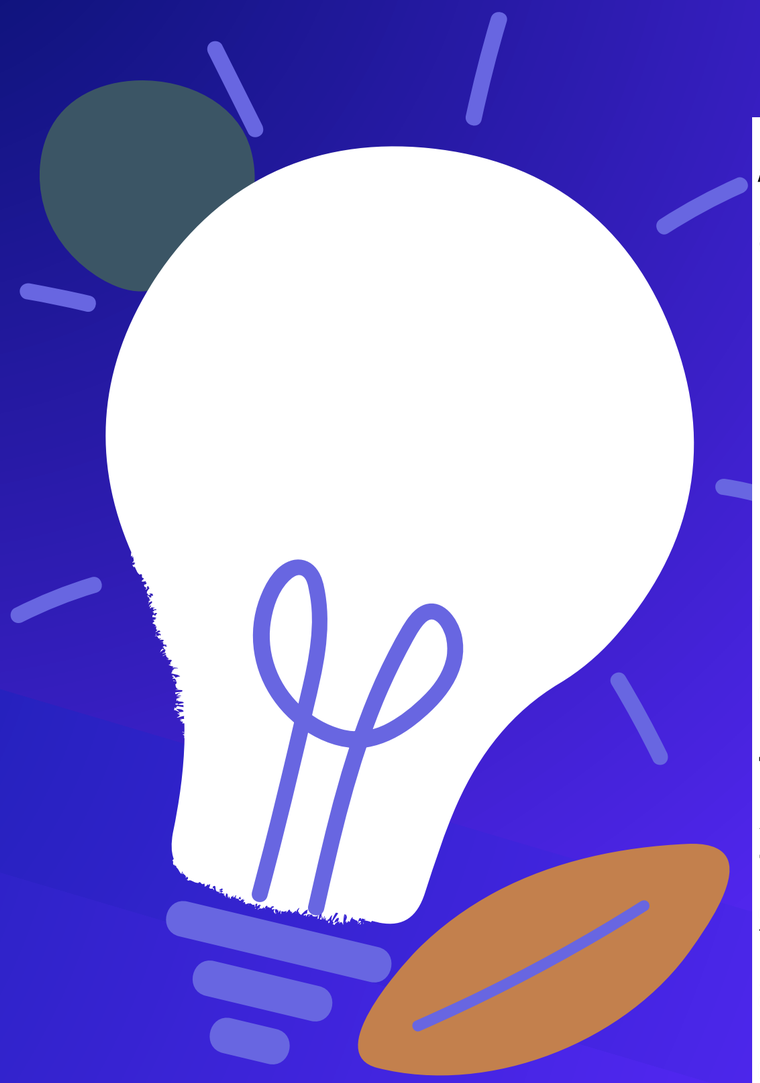
1 Design patterns

pythonic code (proxy pattern)

```
class RealSubject:
 """The real object with functionality (protected resource)."""
 def __init__(self, data):
  self.data = data

 def get_sensitive_data(self):
  print("Accessing sensitive data:", self.data)

class SecurityProxy(RealSubject):
 """The security proxy controlling access to the real object."""
 def __init__(self, real_subject, username, password):
  super().__init__(real_subject.data)  # Inherit data
  self.real_subject = real_subject
  self.username = username
  self.password = password
  self.authenticated = False

 def authenticate(self, username, password):
  if username == self.username and password == self.password:
   self.authenticated = True
   print("Authentication successful.")
  else:
   self.authenticated = False
   print("Authentication failed.")

 def get_sensitive_data(self):
  if self.authenticated:
   return super().get_sensitive_data()  # Call parent's method
  else:
   print("Access denied: Not authenticated.")
tion
```

```
# Usage
real_subject = RealSubject("Top Secret Information")
proxy = SecurityProxy(real_subject, "authorized_user", "secret_password")

# Try accessing data without authentication
proxy.get_sensitive_data()  # Output: Access denied: Not authenticated.

# Authenticate and then access data
proxy.authenticate("authorized_user", "secret_password")
proxy.get_sensitive_data()  # Output: Authentication successful. Accessing sensitive data: Top Secret Informa
```

Adaptor pattern



Adapter: Makes incompatible interfaces work together by converting from one interface to another interface that is compatible with the client (e.g., Integrating a third-party library with a different API).

- Benefits of Adaptor Pattern:

1. Increased Compatibility: Adapters allow you to connect two incompatible interfaces. This is especially useful when working with different systems, libraries, or legacy code that may have different data formats or communication protocols.
2. Reusability: Adapters can be designed to be reusable for similar situations. By encapsulating the conversion logic in the adapter, you can avoid duplicating code whenever you need to connect to a system with a different interface.

1 Design patterns (Adaptor pattern)

```
class LegacyRectangle:
    """Legacy class representing a rectangle with incompatible methods."""
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height

    def get_perimeter(self):
        return 2 * (self.width + self.height)


class ModernShape:
    def get_area(self):
        pass


class RectangleAdapter(ModernShape):
    def __init__(self, legacy_rectangle):
        self.legacy_rectangle = legacy_rectangle

    def get_area(self):
        return self.legacy_rectangle.get_area()

# Usage
legacy_rectangle = LegacyRectangle(5, 3)

# Using legacy methods
print(legacy_rectangle.get_area())  # Output: 15
print(legacy_rectangle.get_perimeter())  # Output: 16

# Adapting to ModernShape interface
modern_shape = RectangleAdapter(legacy_rectangle)

# Using the modern interface method
print(modern_shape.get_area())  # Output: 15 (using the adapter)
```

1 Design patterns (Adaptor pattern)

```
class FormattedText:
 def __init__(self, font, style):
  self.font = font
  self.style = style

 def get_formatted_text(self, text):
  formatted_text = f"<span style='font-family: {self.font}; {self.style}'>{text}</span>"
  return formatted_text


class FormattedTextFactory:
 _pool = {}

 @classmethod
 def get_formatted_text(cls, font, style):
  key = (font, style)
  if key not in cls._pool:
   cls._pool[key] = FormattedText(font, style)
  return cls._pool[key]
```

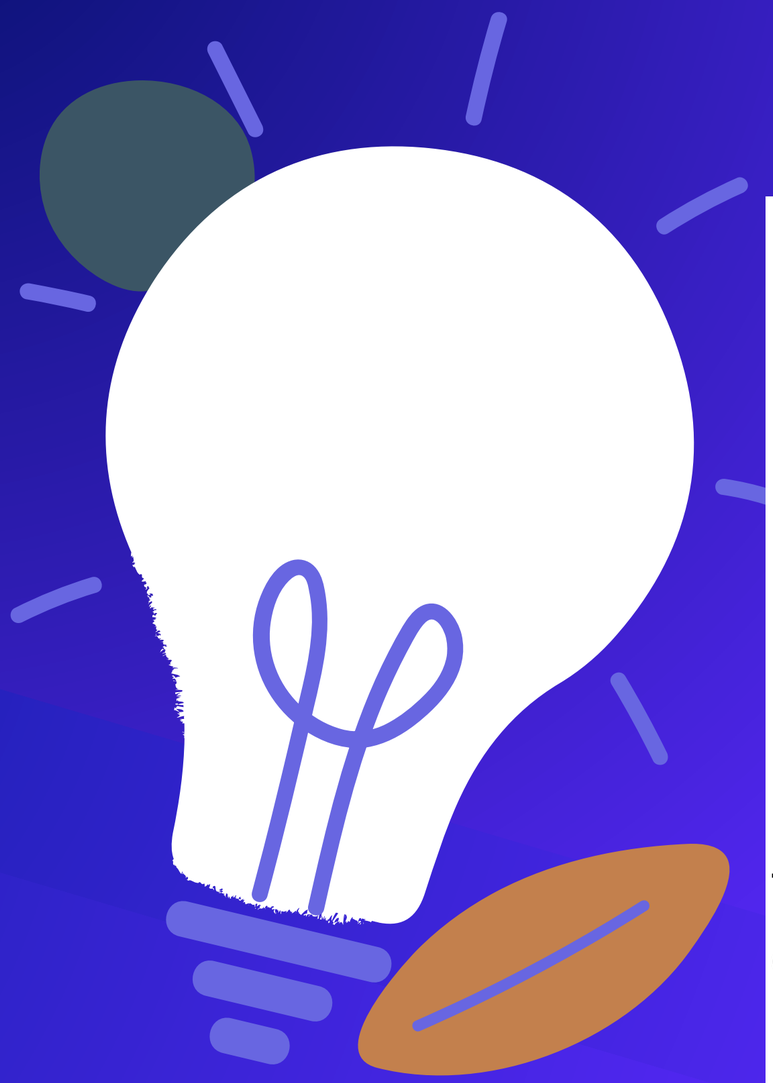
```
formatted_text = FormattedTextFactory.get_formatted_text("Arial", "font-weight: bold")
text1 = formatted_text.get_formatted_text("This is bold text.")
text2 = formatted_text.get_formatted_text("Another bold sentence.")

formatted_text2 = FormattedTextFactory.get_formatted_text("Times New Roman", "font-style: italic")
text3 = formatted_text2.get_formatted_text("This is italic text.")
```

1 Design patterns

Behavioral Patterns (Focus on communication and interaction):

1. Observer: Establishes communication between objects (e.g., Implementing a publish-subscribe system).
2. Strategy: Chooses an algorithm at runtime (e.g., Sorting data using different algorithms based on user selection).
3. Template Method: Defines an algorithm skeleton with customizable steps (e.g., Implementing a data processing pipeline with core steps and flexible options).
4. Command: Encapsulates a request as an object (e.g., Implementing undo/redo functionality in a text editor).
5. Iterator: Provides a way to access elements of a collection sequentially (e.g., Iterating over elements in a list or a complex data structure).
6. Mediator: Defines an object that coordinates communication between a group of objects (e.g., Implementing communication between UI elements in a complex application).
7. Chain of Responsibility: Passes a request along a chain of objects until one handles it (e.g., Implementing authorization checks with different levels of permission).
8. State: Allows an object to alter its behavior when its internal state changes (e.g., Implementing different game character states like idle, running, jumping).

Observer pattern



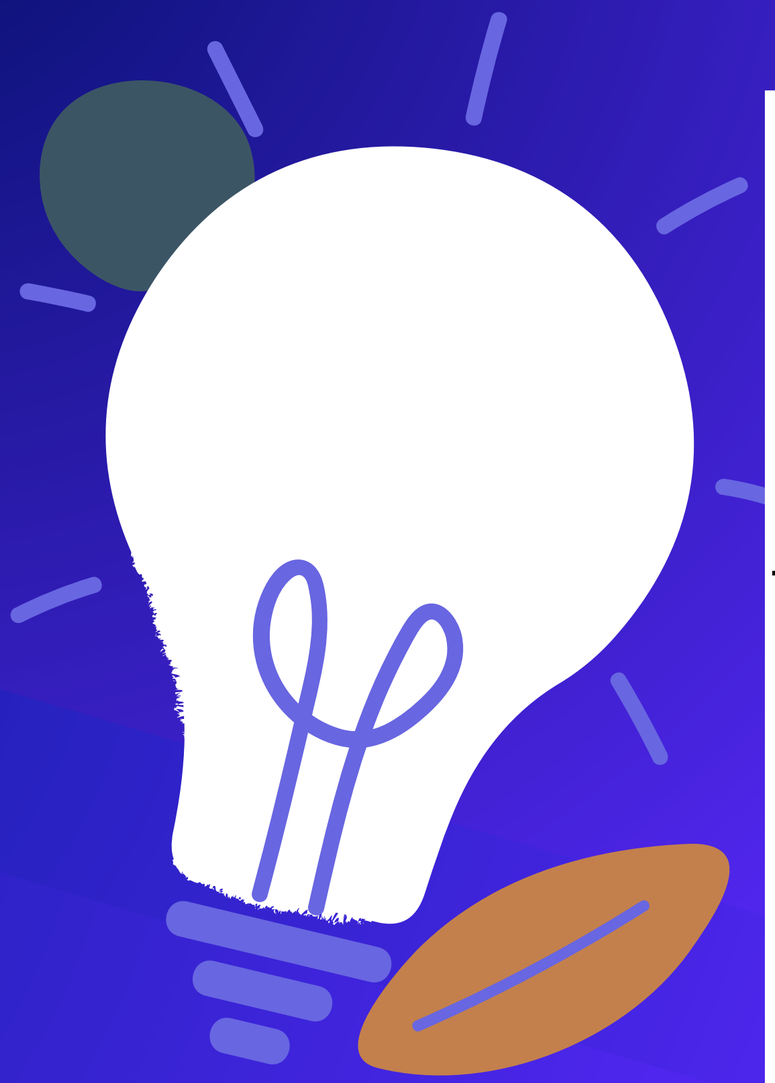
- The Observer Pattern:imagin if we have 1 courses and multiple students and students want to check if there are any changes in content, what should they do:
  the solution is if there are any changes in courses(subjects) we notify the students(observers)

- Benefits of Observer Pattern:

Loose Coupling: Observers don't need to know the internal details of the Subject. They just need to implement the update method to react to changes.

1 Design patterns (Observer pattern)

```
class WeatherData:
 """Subject (Weather Station)"""
 def __init__(self):
  self.observers = []
  self.temperature = None
  self.humidity = None
  self.pressure = None
 def register_observer(self, observer):
  """Attach an observer to the subject"""
  self.observers.append(observer)

 def unregister_observer(self, observer):
  """Detach an observer from the subject"""
  self.observers.remove(observer)

 def notify_observers(self):
  """Notify all registered observers about data changes"""
  for observer in self.observers:
   observer.update(self.temperature, self.humidity, self.pressure)
 def set_measurements(self, temperature, humidity, pressure):
 """Update weather data and notify observers"""
 self.temperature = temperature
 self.humidity = humidity
 self.pressure = pressure
 self.notify_observers()
class Observer:
 """Interface for weather observers"""
 def update(self, temperature, humidity, pressure):
 pass
class NewsChannel(Observer):
 """Concrete observer (News Channel)"""
 def update(self, temperature, humidity, pressure):
 print("News Channel: Updating weather forecast...")
 # Use data to update weather forecast
# Example Usage
weather_data = WeatherData()
news_channel = NewsChannel()
weather_app = WeatherApp()

weather_data.register_observer(news_channel)
weather_data.register_observer(weather_app)

weather_data.set_measurements(25, 60, 1013)
```



Templet pattern

- The Templet Pattern: imagine we have different multible objects that follow the same steps to process is that accepteble if we duplicate the same code but on different objects ,so what should we do:

  the solution is to make pipeline for the same steps and that is what Template principles make

- **Benefits of Templet Pattern:**
- Code Reusability: The core algorithm structure is defined once in the superclass, promoting code reuse.
- Extensibility: Subclasses can customize specific steps without affecting the overall flow.

1 Design patterns

Template pattern

```
class GameCharacter:
    def __init__(self, name):
        self.name = name

    def move(self):
        # Template method defines the overall movement flow
        self.check_environment()
        self.plan_movement()
        self.perform_movement()

    def check_environment(self):
        print(f"{self.name} is checking the environment for obstacles.")

    def plan_movement(self):
        # Abstract method, subclasses define specific movement plan
        pass

    def perform_movement(self):
        print(f"{self.name} is performing the planned movement.")
```

- class Warrior(GameCharacter):
- def plan_movement(self):
- print(f"{self.name} (warrior) plans a direct attack move.")
-
- class Archer(GameCharacter):
- def plan_movement(self):
- print(f"{self.name} (archer) plans a strategic flanking move.")
-
- # Example Usage
- warrior = Warrior("Aragorn")
- archer = Archer("Legolas")
-
- warrior.move()
- # Output:
- # Aragorn is checking the environment for obstacles.
- # Aragorn (warrior) plans a direct attack move.
- # Aragorn is performing the planned movement.
-
- archer.move()
- # Output:
- # Legolas is checking the environment for obstacles.
- # Legolas (archer) plans a strategic flanking move.
- # Legolas is performing the planned movement.

Architecture pattern

Software Architecture: is the blueprint of building software. It shows the overall structure of the software, the collection of components in it, and how they interact with one another

Architecture pattern: Is a set of architectural design decisions They determine how the system is organized, identifying its main components and theirrelationships they are not the final architecture themselves , its blueprint.

Examples:

-Client Architecture Patterns

MVC MVT MVP MVVM

2Architecture pattern

A

Purpose

B

Structure

C

Benefits

D

components

2Architectural patterns

A

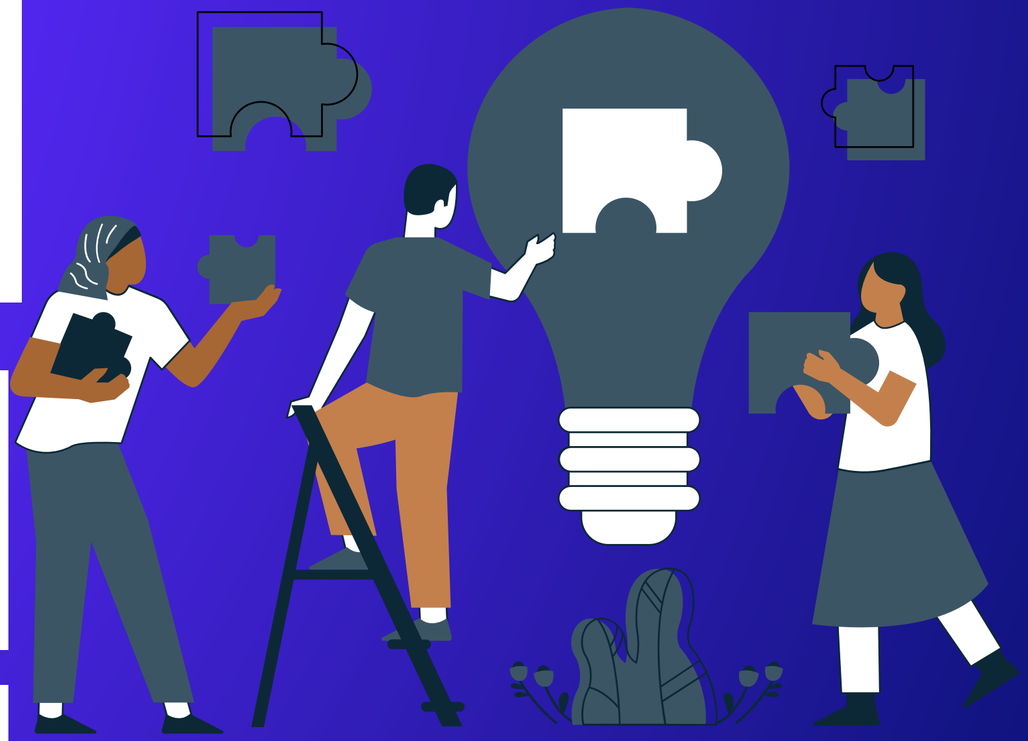
- These patterns offer guidelines, best practices, flexibility, maintainability, and scalability.
- By leveraging architectural patterns, developers can build robust, adaptable, and high-quality software systems.

B

Each design has its own structure

C

- Problem Solving , Guidance and Direction, Communication and Collaboration
- Makes it easy to handle the system parts.



2Architectural patterns

COMPONENTS

components for Architectural patterns

Model:

- Represents the data and business logic of the application.

View:

- Represents the user interface of the application.

Controller:

- Acts as an intermediary between the model and view.

Services:

Provide reusable functionality that is shared across multiple components or modules.

Interfaces:

Define the contract or communication protocol between different components of the system.

Middleware:

Enhances the scalability, reliability, and security of the system.

Data Access Layer:

Handles interactions with the underlying data storage, such as databases or external APIs.

2Architectural patterns

Model-View-Template (MVT)Model:

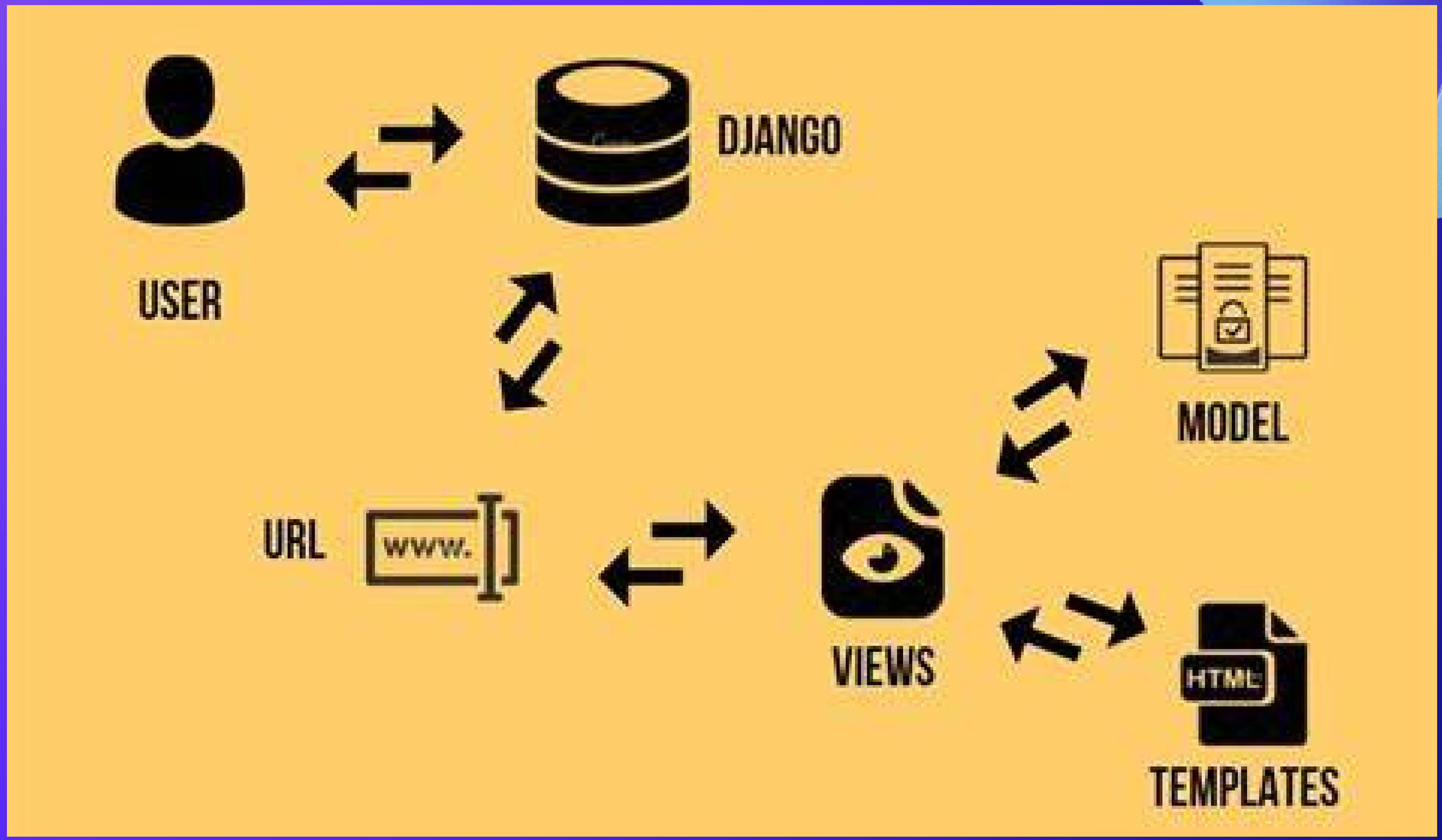
Model View Template

Model: Models handle tasks such as data validation, manipulation, and retrieval from the database

View: Views contain the logic for processing user input, interacting with models to retrieve or manipulate data, and rendering templates to generate dynamic content.

Template: The Template represents the user interface layer of the application. It defines the structure and layout of the HTML pages that are rendered to the user

Django, a popular web framework for Python, follows the Model-View-Template (MVT) architectural pattern


USER
DJANGO
MODEL
URL
www.
VIEWS
HTML
TEMPLATES

2Architectural patterns

MVC **vs** MVP

Model: Represents the data and business logic of the application.

Model:Represents the data and business logic of the application, similar to the Model in MVC.

model----------view

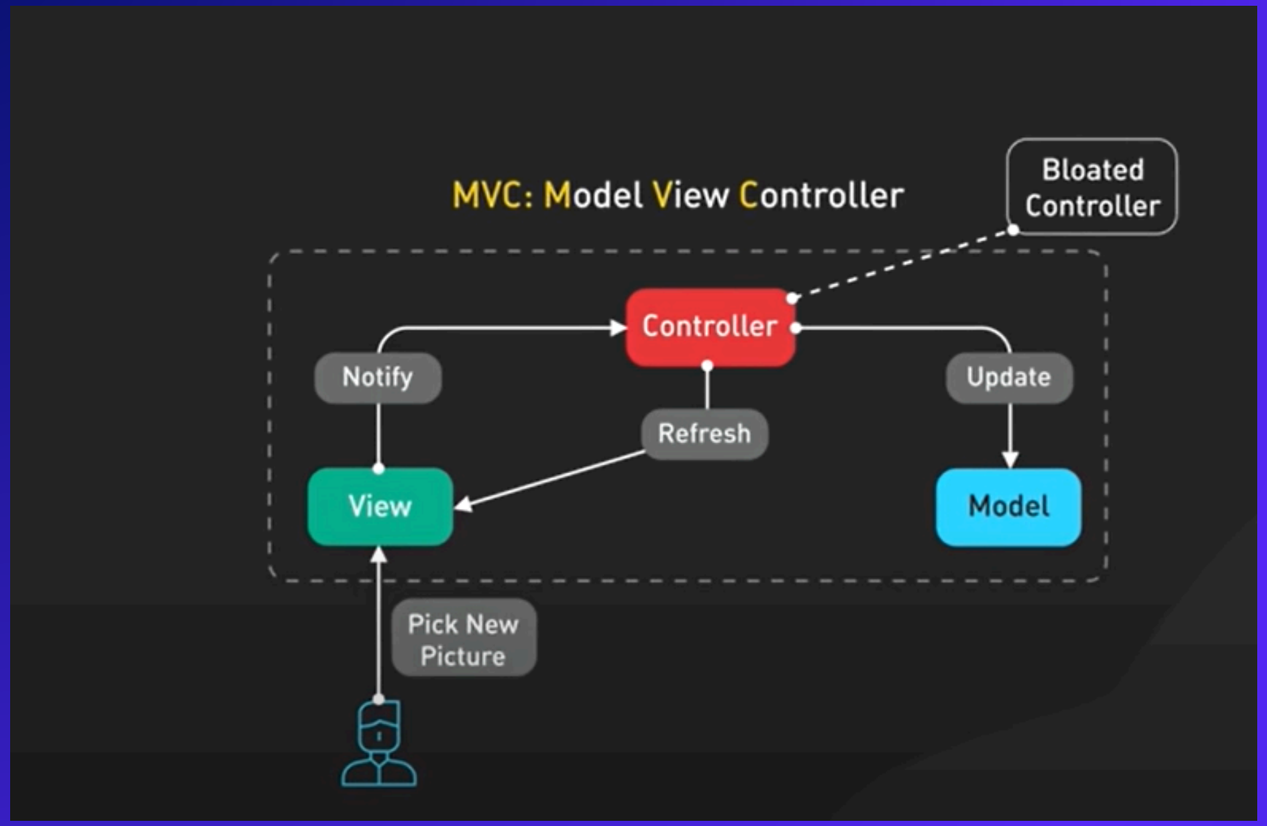
model--P----view

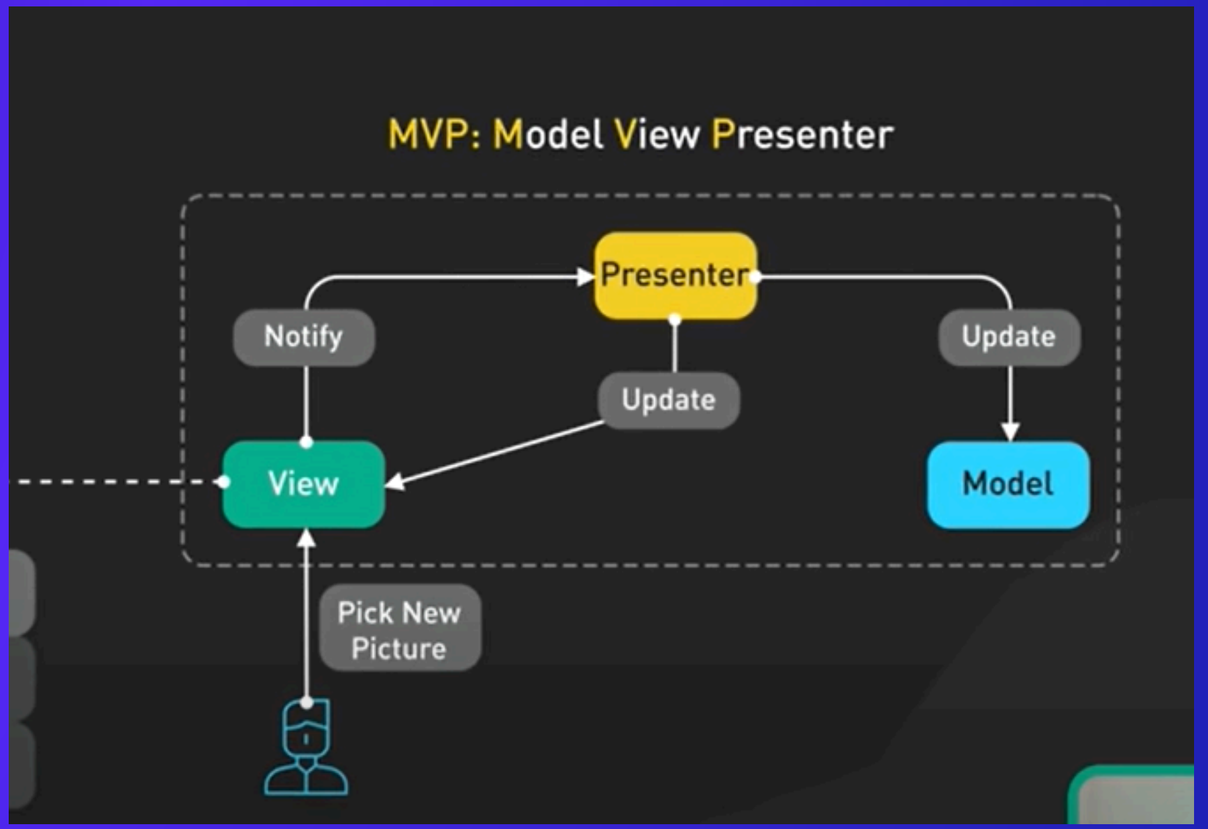
View: Represents the user interface (UI) components of the application

View: Represents the user interface (UI) components of the application

Controller: Acts as an intermediary between the Model and the View

Presenter:Acts as an intermediary between the Model and the View, similar to the Controller in MVC. but ,MVP typically holds a reference to the View, allowing it to directly manipulate the View's behavior.



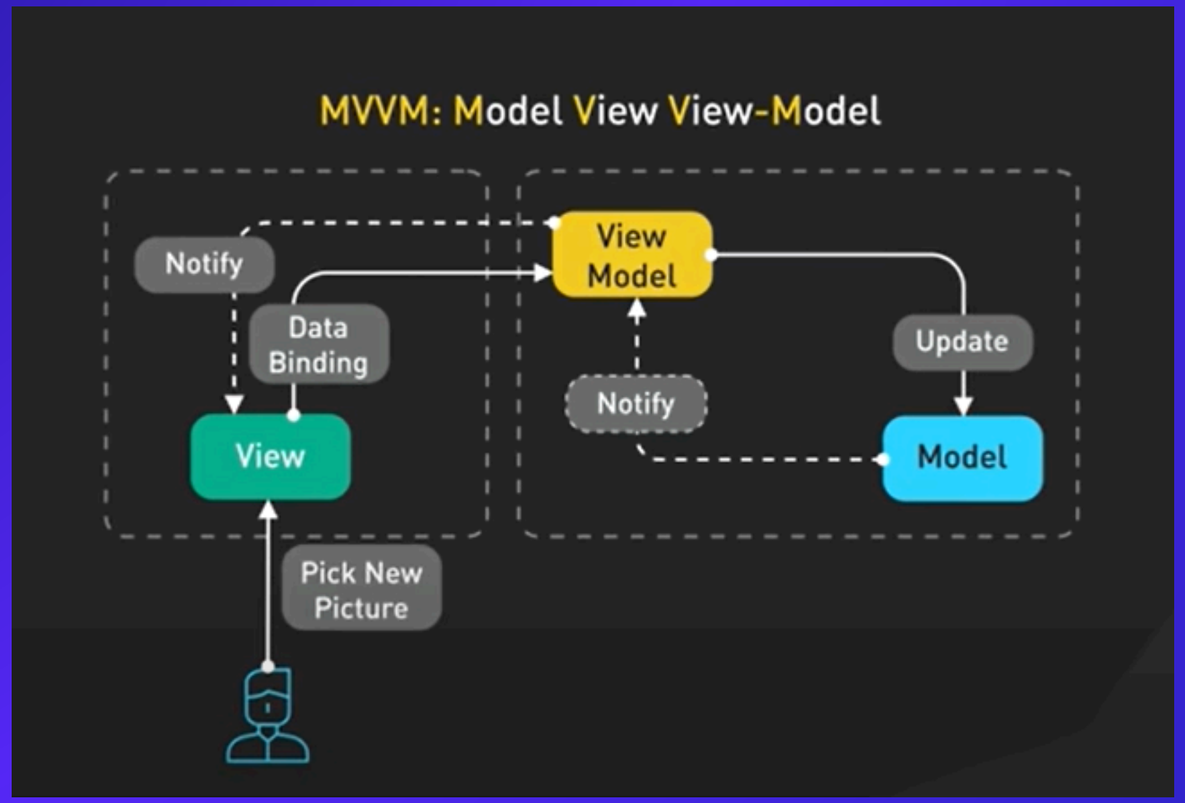


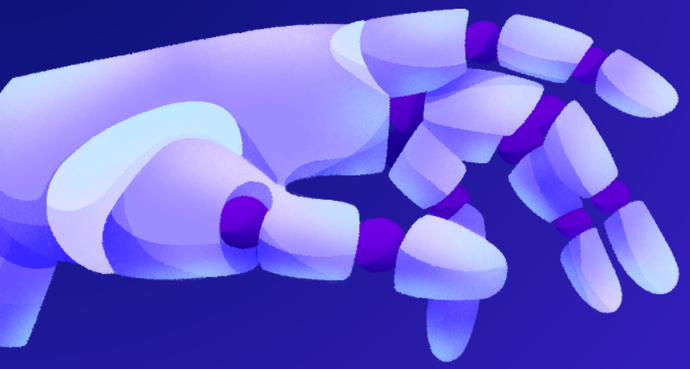
MVP may be preferable for larger, more complex applications where testability and flexibility are important

MVC may be more suitable for smaller projects or applications with simpler user interfaces.

MVVM

ViewMode: store & manage data, but UI data,
it know the data



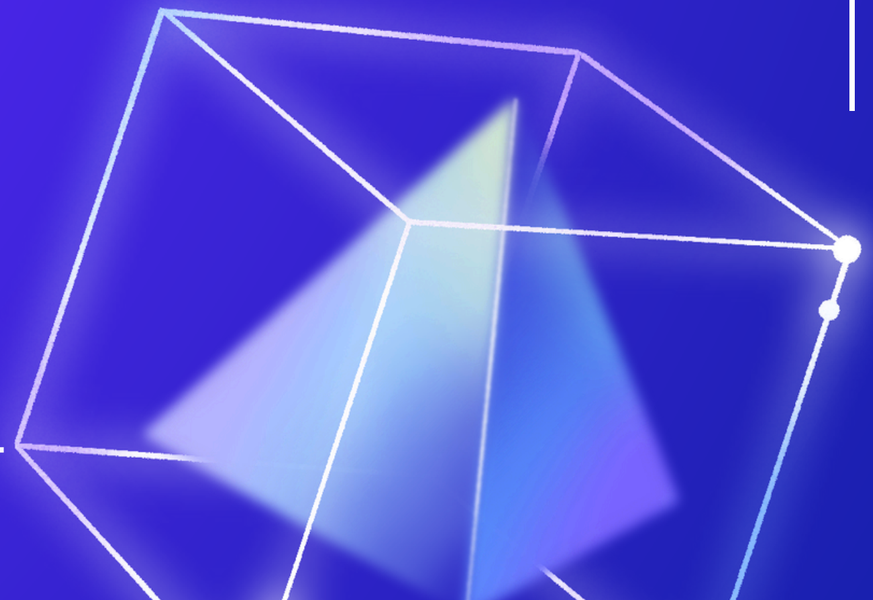


3- SOLID Principles

The SOLID principles are a set of design principles in object-oriented programming intended to make software designs more understandable, flexible, and maintainable.

Here's a overview of each principle:

1. **Single Responsibility Principle**
2. **Open/Closed Principle**
3. **Liskov Substitution Principle**
4. **Interface Segregation Principle**
5. **Dependency Inversion Principle**

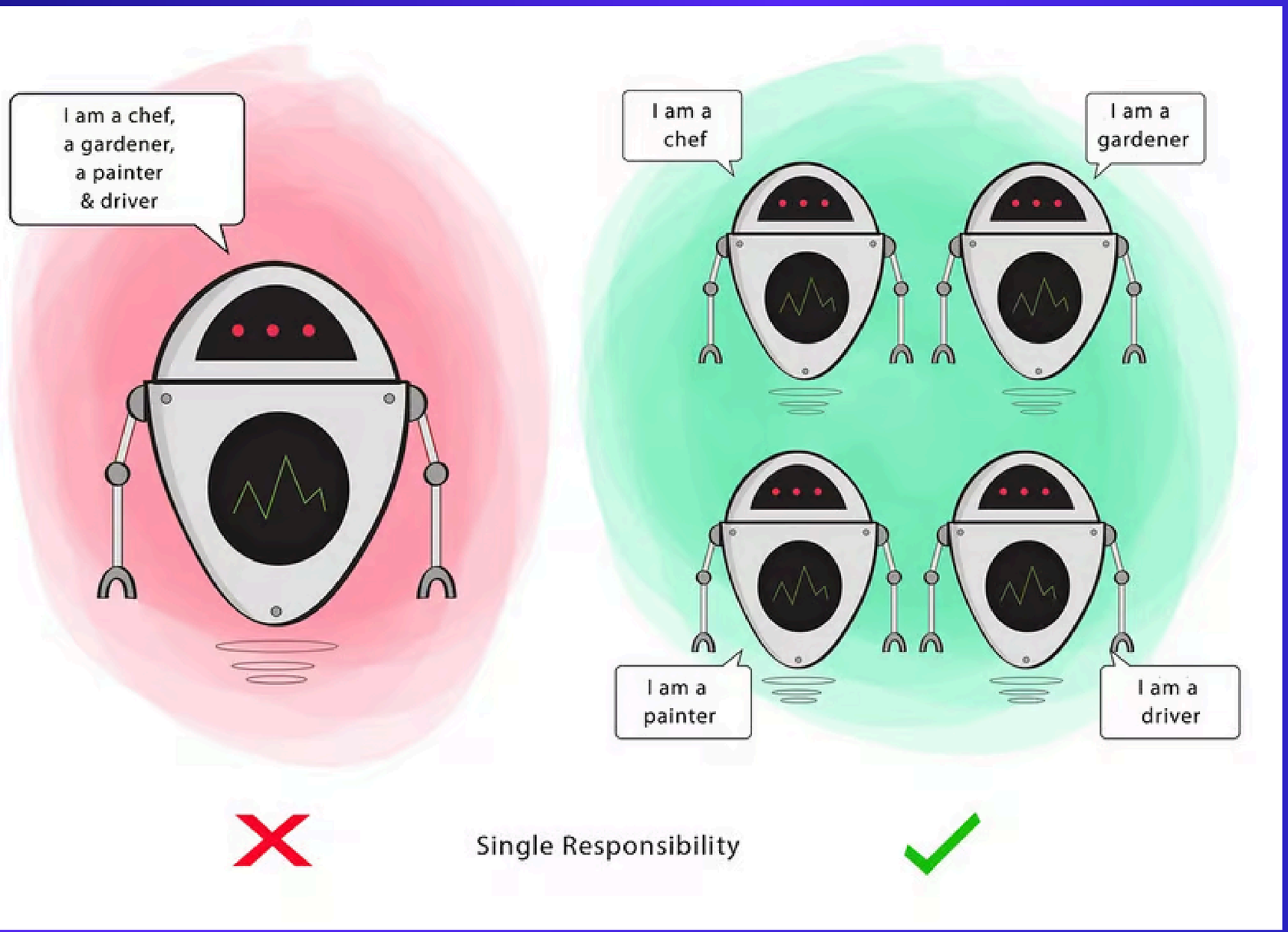


3.1 – Single Responsibility Principle

- A class should have only one reason to change, meaning it should have only one responsibility or purpose.

Advantages :-

1. Testing
2. lower coapling
3. Easier to understand
4. organized

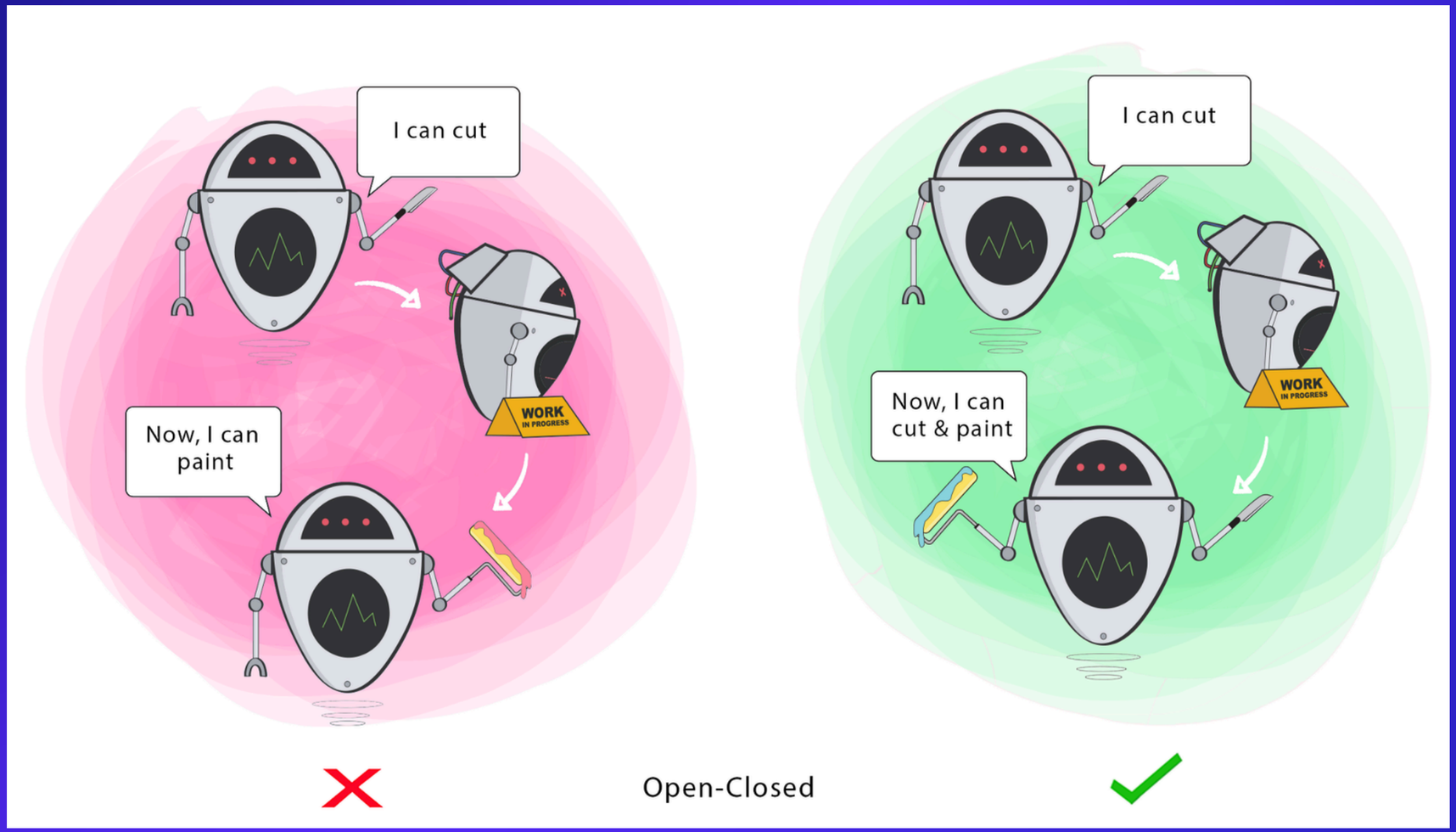

I am a chef, a gardener, a painter & driver
I am a chef
I am a gardener
I am a painter
I am a driver
Single Responsibility

3.2 - Open/Closed Principle

- Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

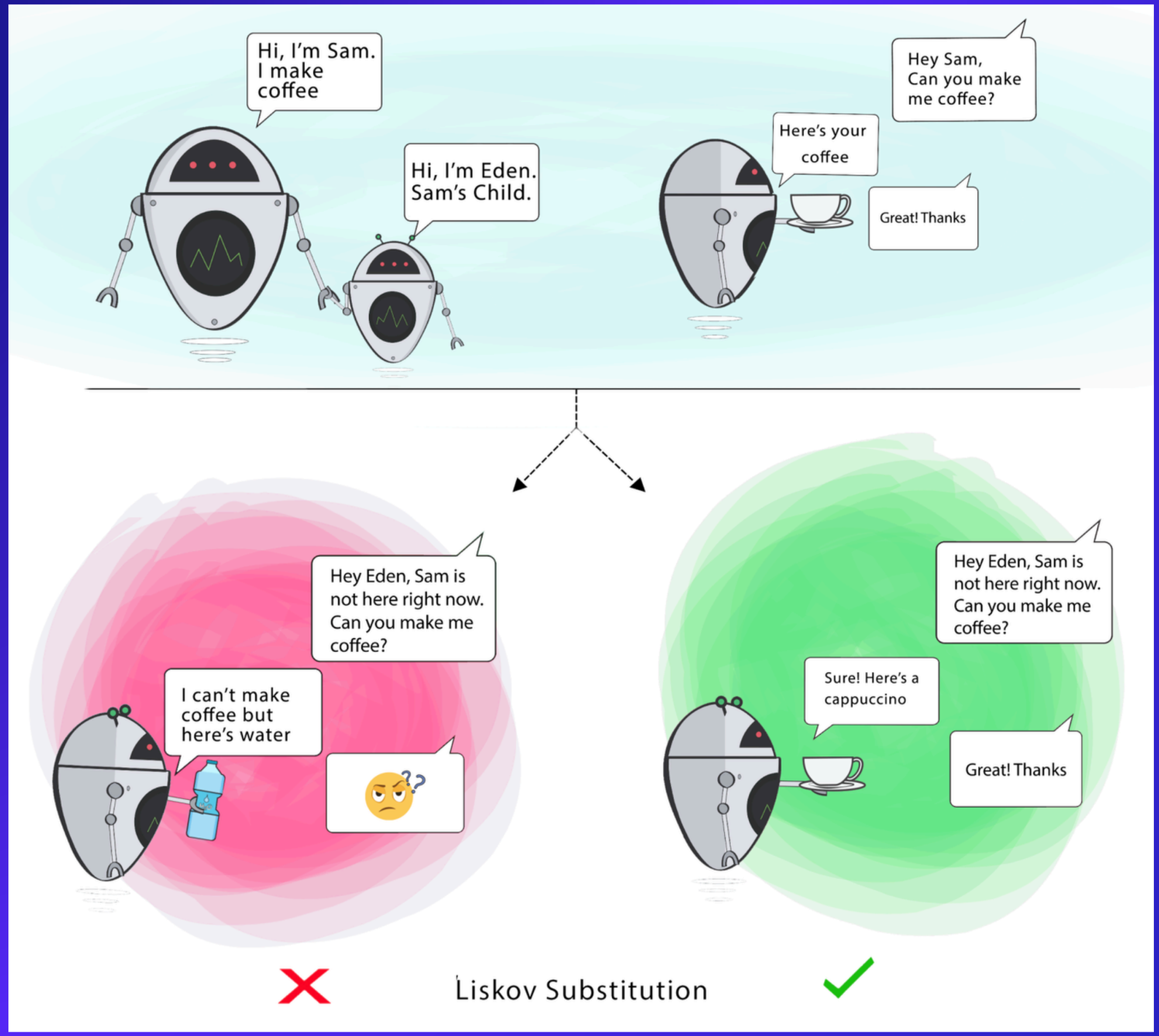
Advantages :-

- This principle encourages the use of abstraction and polymorphism to allow behavior to be extended without modifying existing code.
- It promotes modular and reusable code, enabling easier maintenance and scalability.


I can cut
WORK
IN PROGRESS
Now, I can
paint
I can cut
WORK
IN PROGRESS
Now, I can
cut & paint
Open-Closed

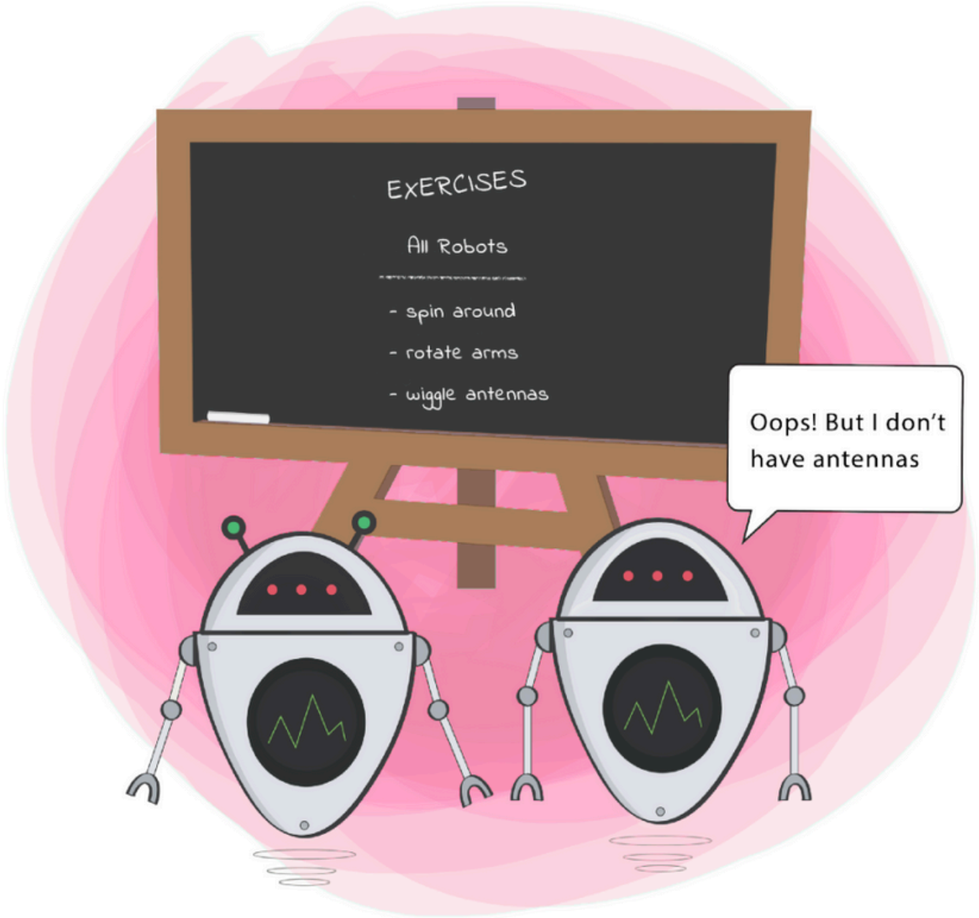
3.3 – Liskov Substitution Principle

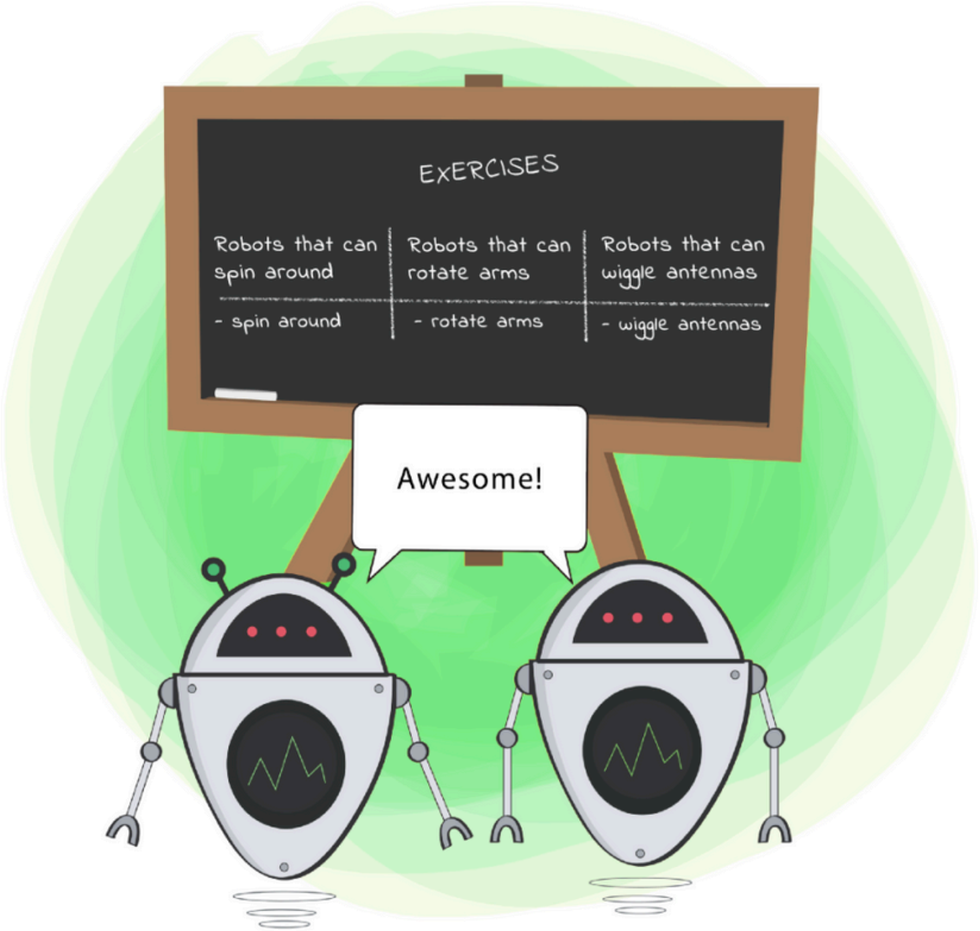
- Objects of a superclass should be replaceable with objects
- Subtypes must adhere to the contract established by their supertypes, meaning they should behave in a manner consistent with the base type.
- Violating this principle can lead to unexpected behavior and can break the substitutability of objects, causing errors in the program.


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks
Liskov Substitution

3.4 - Interface Segregation Principle

- Clients should not be forced to depend on interfaces they do not use.
- Instead of creating large, monolithic interfaces, it's better to define smaller, more specific interfaces that cater to the needs of individual clients.
- This principle helps prevent "interface pollution" and ensures that classes only depend on the methods they actually need, leading to cleaner and more maintainable code.

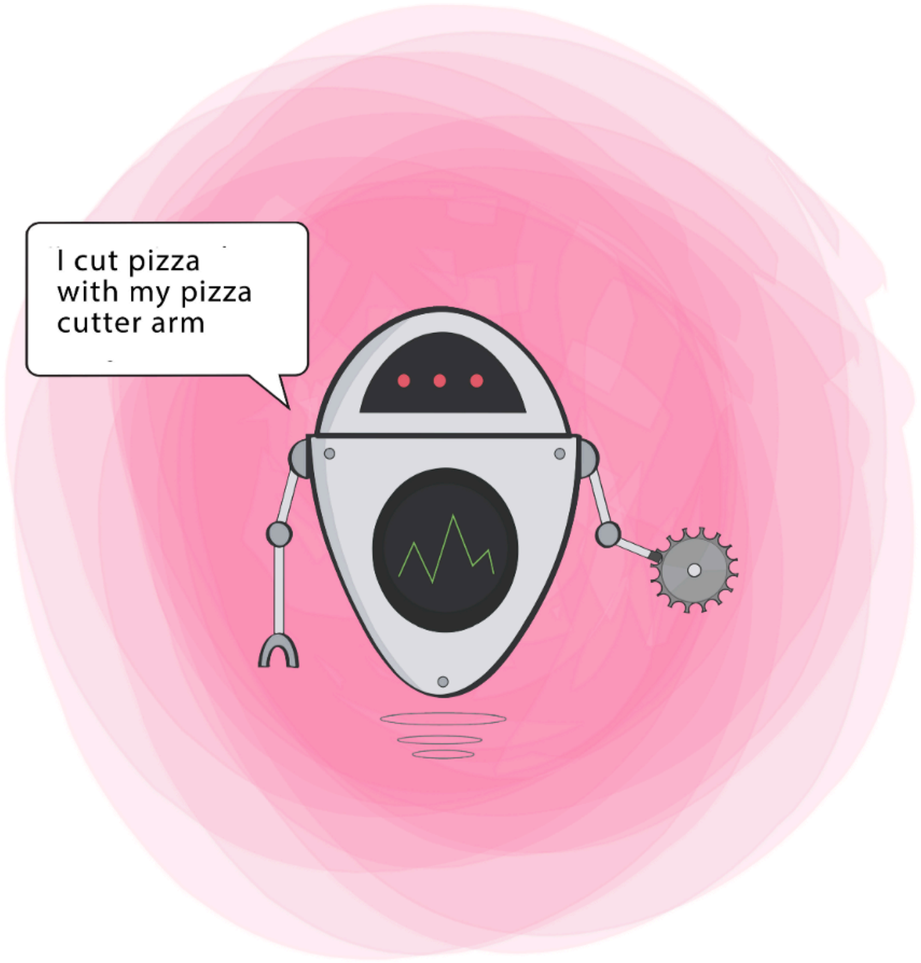


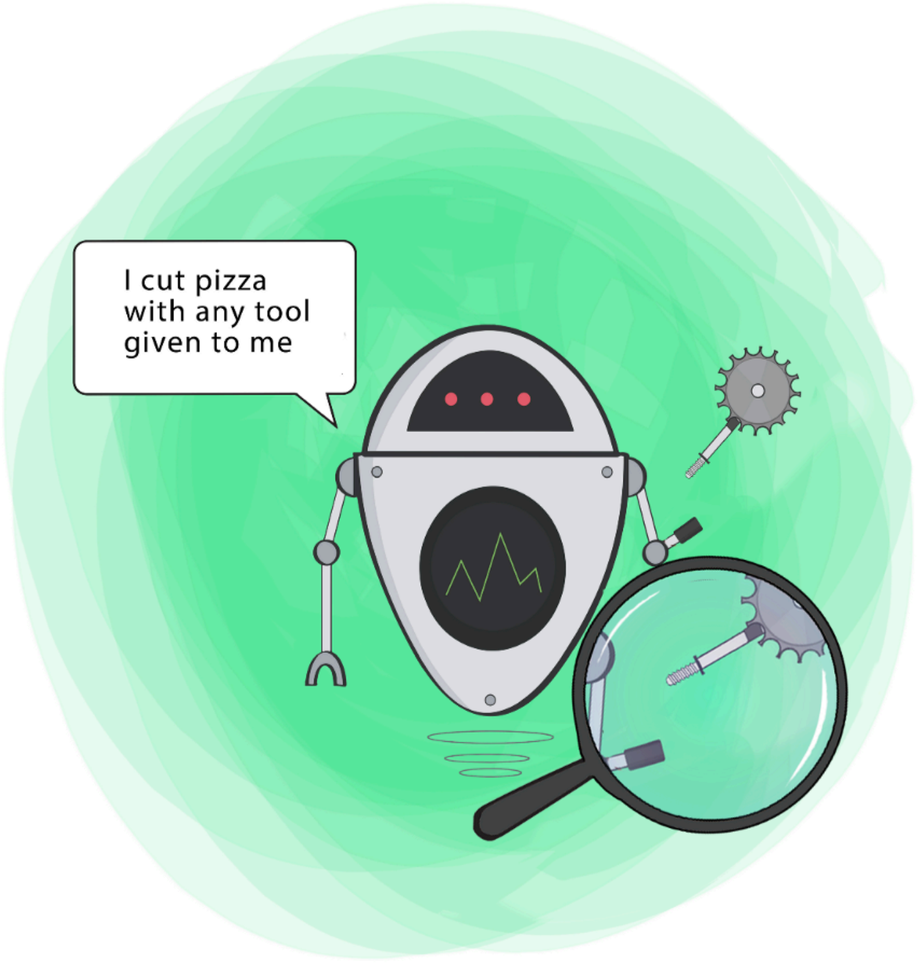


Interface Segregation

3.5 -Dependency Inversion Principle

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.
- By decoupling high-level and low-level modules through abstractions, this principle promotes flexibility, extensibility, and ease of testing.
- Dependency Injection (DI) is a common technique used to implement the Dependency Inversion Principle.



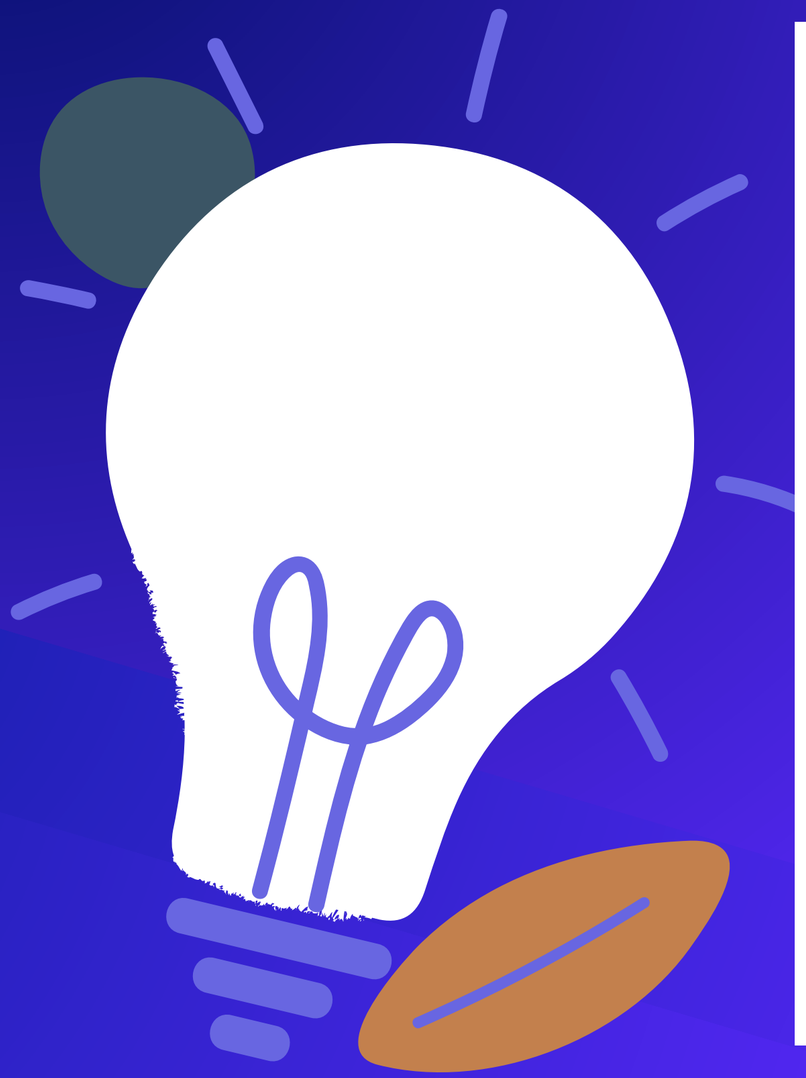


Dependency Inversion

Test-Driven Development

- imagine you want to make large program to client and you have to handle each constrains and every possible client interact with Application to handle and so that there are many problems

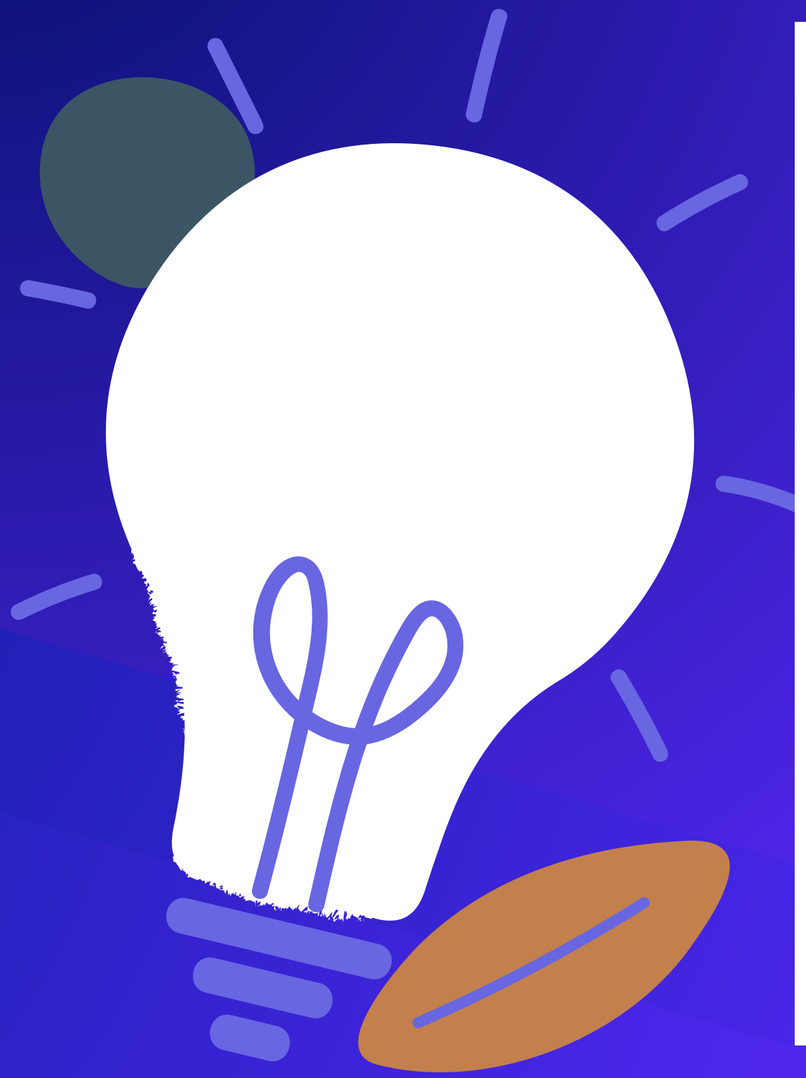
- **Poor Code Quality: loss in writing clear, concise, and testable code by focusing on the specific functionality before implementation.**
- **Bad Design: Loss in thinking about the desired behavior first, leading to a more well-defined and maintainable codebase.**
- **Delay Bug Detection: there are no Tests that catch errors early in the development cycle, making them easier to fix and reducing the cost of bug fixing later.**
- **bad Documentation: there are no Tests that they themselves serve as documentation, clarifying the expected behavior of different parts of the code.**



Steps

- TDD recommends writing tests that would check the functionality of your code prior to your writing the actual code.

- Only when you are happy with your tests and the features it tests, do you begin to write the actual code in order to satisfy the conditions imposed by the test that would allow them to pass.



Steps

- Here's the core idea:
- Start with a Requirement: Identify a specific feature or functionality you want to develop.
- Write a Failing Test: Create an automated test that initially fails because the desired functionality isn't implemented yet. This test defines the expected behavior of the code.
- Write the Minimum Code: Implement the minimal amount of code necessary to make the failing test pass. This ensures the code fulfills the intended functionality.
- Refactor: Refactor the code to improve readability, maintainability, and performance without affecting its functionality (as validated by the tests).
- Repeat: Repeat steps 1-4 for each new feature or functionality.

3.Test-Driven Development

```
class Item:
    def __init__(self, name, price):
        self.name = name
        self.price = price

class Discount:
    def __init__(self, percentage):
        self.percentage = percentage

class User:
    def __init__(self, email):
        self.email = email

class ShoppingCart:
    def __init__(self, user=None):
        self.items = []
        self.total_price = 0.0  # Assuming attribute for total price
        self.user = user  # Optional user for authentication

    def add_item(self, item):
        if self.user is None:
            raise PermissionError("User needs to be authenticated to add items")
        self.items.append(item)
    def remove_item(self, item):
    self.items.remove(item)

    def calculate_total_price(self):
    total = 0
    for item in self.items:
    total += item.price
    return total

    def apply_discount(self, discount):
    total_price = self.calculate_total_price()
    discount_amount = total_price * (discount.percentage / 100)
    self.total_price -= discount_amount


# Test Cases
def test_add_item_to_cart():
    """Tests adding an item to the shopping cart."""
    cart = ShoppingCart()
    item = Item("Apple", 1.50)
    cart.add_item(item)

    assert len(cart.items) == 1
    assert cart.items[0] == item
```

3.Test-Driven Development

```
def test_remove_item_from_cart():
 """Tests removing an item from the shopping cart."""
 cart = ShoppingCart()
 item1 = Item("Apple", 1.50)
 item2 = Item("Bread", 2.25)
 cart.add_item(item1)
 cart.add_item(item2)

 cart.remove_item(item1)

 assert len(cart.items) == 1
 assert cart.items[0] == item2
```

- def remove_item(self, item):
- self.items.remove(item)
-
- def calculate_total_price(self):
- total = 0
- for item in self.items:
- total += item.price
- return total
-
- def apply_discount(self, discount):
- total_price = self.calculate_total_price()
- discount_amount = total_price * (discount.percentage / 100)
- self.total_price -= discount_amount
-
-
- # Test Cases
- def test_add_item_to_cart():
- """Tests adding an item to the shopping cart."""
- cart = ShoppingCart()
- item = Item("Apple", 1.50)
- cart.add_item(item)
-
- assert len(cart.items) == 1
- assert cart.items[0] == item

To begin writing tests in Python we will use the unittest module that comes with Python.
(Write Failing Test) create a new file mytests.py, which will contain all tests.

Notice that we are importing helloworld() function from mycode file. In the file mycode.py we will initially just include the code below, which creates the function but doesn't return anything at this stage:

```
import unittest
from mycode import *

class MyFirstTests(unittest.TestCase):

def test_hello(self):
        self.assertEqual(hello_world(), 'hello world')
```

(Run and Fail Test) Running python mytests.py will generate the following output in the command line:

This clearly indicates that the test failed, which was expected.

Fortunately, we have already written the tests, so we know that it will always be there to check this function, which gives us confidence in spotting potential bugs in the future.

```
F

======================================================================

FAIL: test_hello (__main__.MyFirstTests)

----------------------------------------------------------------------

Traceback (most recent call last):

File "mytests.py", line 7, in test_hello

self.assertEqual(hello_world(), 'hello world')

AssertionError: None != 'hello world'

----------------------------------------------------------------------

Ran 1 test in 0.000s

FAILED (failures=1)
```


Thank
you