Software Engineering Project

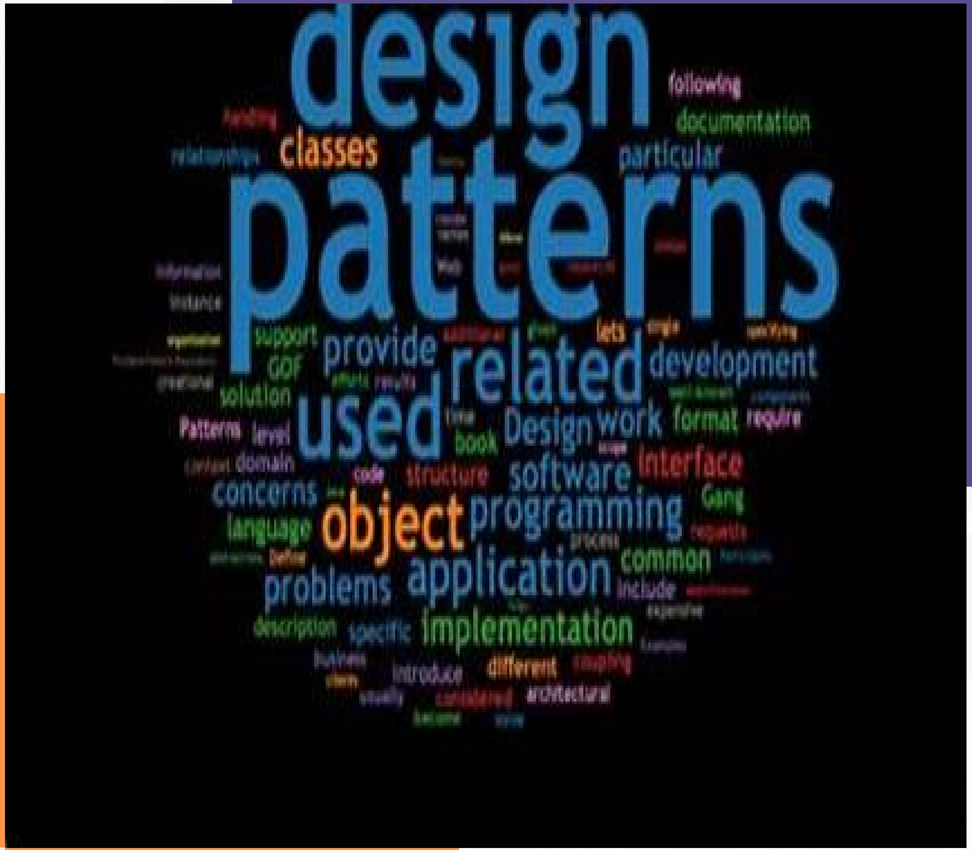
Ahmed Mohamed Ahmed Ali
Farag Mousa Farag
Sherief mohamed

Conclusion

Table Of Contents

WPS

--A picture is worth a thousand words



01

software design pattern in python

Model-View-Controller (MVC):

- Design Patterns is the most essential part of Software Engineering, as they provide the general repeatable solution to a commonly occurring problem in software design. They usually represent some of the best practices adopted by experienced object-oriented software developers.

- We can not consider the Design Patterns as the finished design that can be directly converted into code. They are only templates that describe how to solve a particular problem with great efficiency.

- Design patterns help developers write cleaner, more maintainable code by promoting best practices and standardizing common solutions.

--A picture is worth a thousand words

Software design patterns in Python include:

Observer Pattern

Singleton Pattern

Decorator Pattern

Factory Pattern

Strategy Pattern

--A picture is worth a thousand words

Singleton Pattern:

- Ensures that a class has only one instance and provides a global point of access to it.

lab.py

2 usages

```python
class Singleton:
    _instance = None

    def __new__(cls):
        if cls._instance is None:
            cls._instance = super().__new__(cls)
        return cls._instance
s1 = Singleton()
s2 = Singleton()
print(s1 is s2)
```

--A picture is worth a thousand words

Factory Pattern:

- Defines an interface for creating objects, but lets subclasses decide which class to instantiate.

- is a creational design pattern used to create concrete implementations of a common interface.

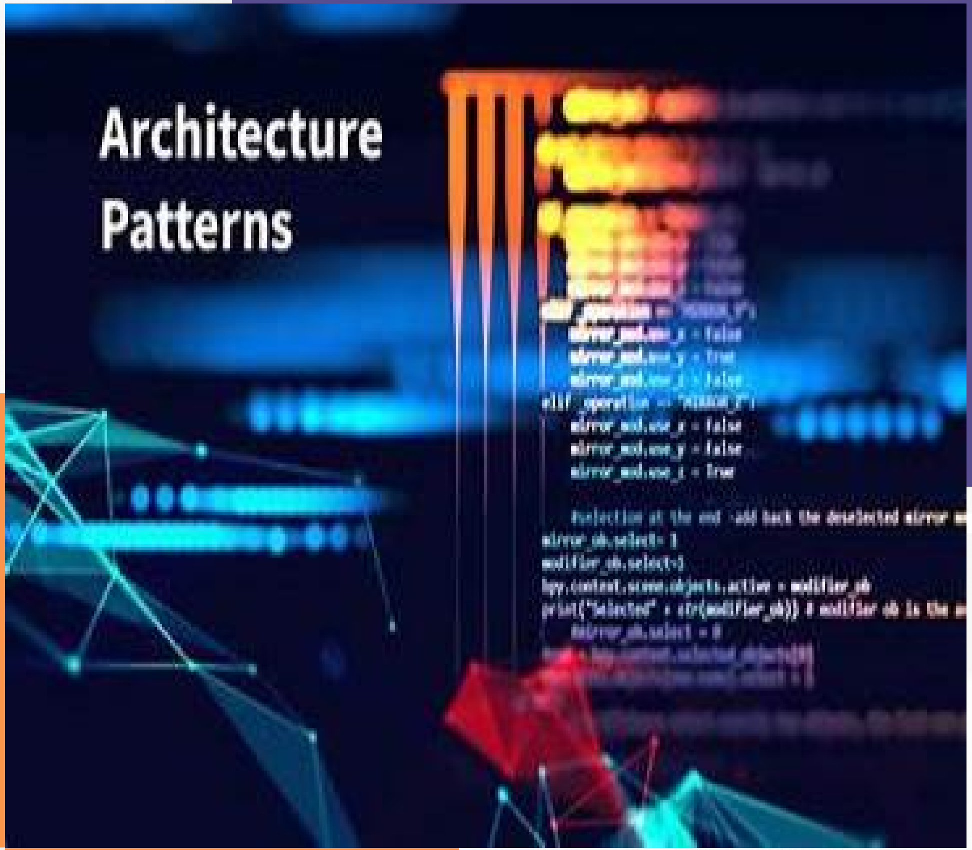
--A picture is worth a thousand words

Observer Pattern&Decorator Pattern:

- Strategy Pattern is Defines a family of algorithms, encapsulates each one, and makes them interchangeable.

- Decorator Pattern is Allows behavior to be added to individual objects dynamically without affecting the behavior of other objects from the same class.

--A picture is worth a thousand words



02

software architecture pattern in python

--A picture is worth a thousand words

A software architecture pattern

A software architecture pattern in Python is a general, reusable solution to a commonly occurring problem in software design. It provides a structured approach for organizing the components of a software system.

--A picture is worth a thousand words

software architecture patterns in Python include:

Event-Driven Architecture

Model-View-Controller (MVC)

Layered Architecture

--A picture is worth a thousand words

Model-View-Controller (MVC):

- Model: Responsible for managing the data of the application.

Example: Suppose a program is running:

Create a class only :

lab.py

```python
class Task:
    def __init__(self, description, status):
        self.description = description
        self.status = status
```

-- A picture is worth a thousand words

Model-View-Controller (MVC):

- **View**: Responsible for displaying the data to the user

Example: Suppose a program is running:

Print a class :

lab.py

```python
def show_tasks(tasks):
    for task in tasks:
        print(f"{task.description} - Status: {task.status}")
```

--A picture is worth a thousand words

Model-View-Controller (MVC):

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

Example: Suppose a program is running:

controll from anything of a class :

lab.py

```python
class TaskController:
    def __init__(self):
        self.tasks = []

    def add_task(self, description):
        new_task = Task(description, "Incomplete")
        self.tasks.append(new_task)

    def show_tasks(self):
        show_tasks(self.tasks)
```

--A picture is worth a thousand words

Layered Architecture:

- Divides the application into layers, where each layer performs a specific set of functions.

- It is divided into several layers:
  - Presentation Layer (UI):
    - This layer is responsible for interacting with the user.

lab.py

```python
class PresentationLayer:
    def display_message(self, message):
        print(f"Displaying message: {message}")
```

--A picture is worth a thousand words

Layered Architecture:

- Divides the application into layers, where each layer performs a specific set of functions.

- It is divided into several layers:
  - Business Logic Layer:
    - This layer contains the application's logic and rules.

lab.py

```python
class BusinessLogicLayer:
    def process_message(self, message):
        # Perform any business logic processing here
        return message.upper()
```

--A picture is worth a thousand words

Layered Architecture:

- Divides the application into layers, where each layer performs a specific set of functions.

- It is divided into several layers:
  - Data Access Layer:
    - This layer interacts with the data source (e.g., a database).

lab.py

```python
class DataAccessLayer:
    def save_message(self, message):
        # Save the message to a database or file
        print(f"Saving message: {message}")
```

--A picture is worth a thousand words

03

SOLID principles in python

--A picture is worth a thousand words

Sales Channels

S — Single Responsibility Principle

O — Open/Closed Principle

L — Liskov Substitution Principle

I — Interface Segregation Principle

D — Dependency Inversion Principle

--A picture is worth a thousand words

Single Responsibility Principle:

- A class should have a single responsibility.
- every class should have only one reason to change.

lab.py

```python
class TaskManager:
    def add_task(self, task):
        print(f"Task added: {task}")
    class LogManager:
        def log_message(self, message):
            print(f"Logging message: {message}")
```

lab.py

```python
class TaskManager:
    def add_task(self, task):
        print(f"Task added: {task}")
class LogManager:
    def log_message(self, message):
        print(f"Logging message: {message}")
```

--A picture is worth a thousand words

Open/Closed Principle (OCP):

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

lab.py

```python
# Example demonstrating OCP
class Shape:
    def area(self):
        def area(self):
            print("Calculating area of rectangle")
    def area(self):
        print("Calculating area of circle")
```

lab.py

```python
# Example demonstrating OCP
class Shape:
    def area(self):
        pass

class Rectangle(Shape):
    def area(self):
        print("Calculating area of rectangle")

class Circle(Shape):
    def area(self):
        print("Calculating area of circle")
```

--A picture is worth a thousand words

Liskov Substitution Principle (LSP):

- Subclasses (Derived) classes must be substitutable for their base classes.
- Liskov's principle is easy to understand but hard to detect in code.
- This principles confirms that our abstraction is correct and helps us get a code that is easy reusable.

lab.py

```python
# Example demonstrating LSP
2 usages
class Bird:
    def fly(self):
        pass

class Eagle(Bird):
    def fly(self):
        print("Eagle flying high")

class Ostrich(Bird):
    def fly(self):
        print("Ostrich cannot fly")
```

lab.py

```python
# Example demonstrating LSP
2 usages
class Bird:
    def fly(self):
        pass

class Eagle(Bird):
    def fly(self):
        print("Eagle flying high")

class Ostrich(Bird):
    def fly(self):
        raise NotImplementedError("Ostrich cannot fly")
```

sand words

Interface Segregation Principle (ISP):

- Clients should not be forced to depend on methods that they do not use.
- It states that interfaces should be small and focused rather than having one large interface that contains many unrelated functions.

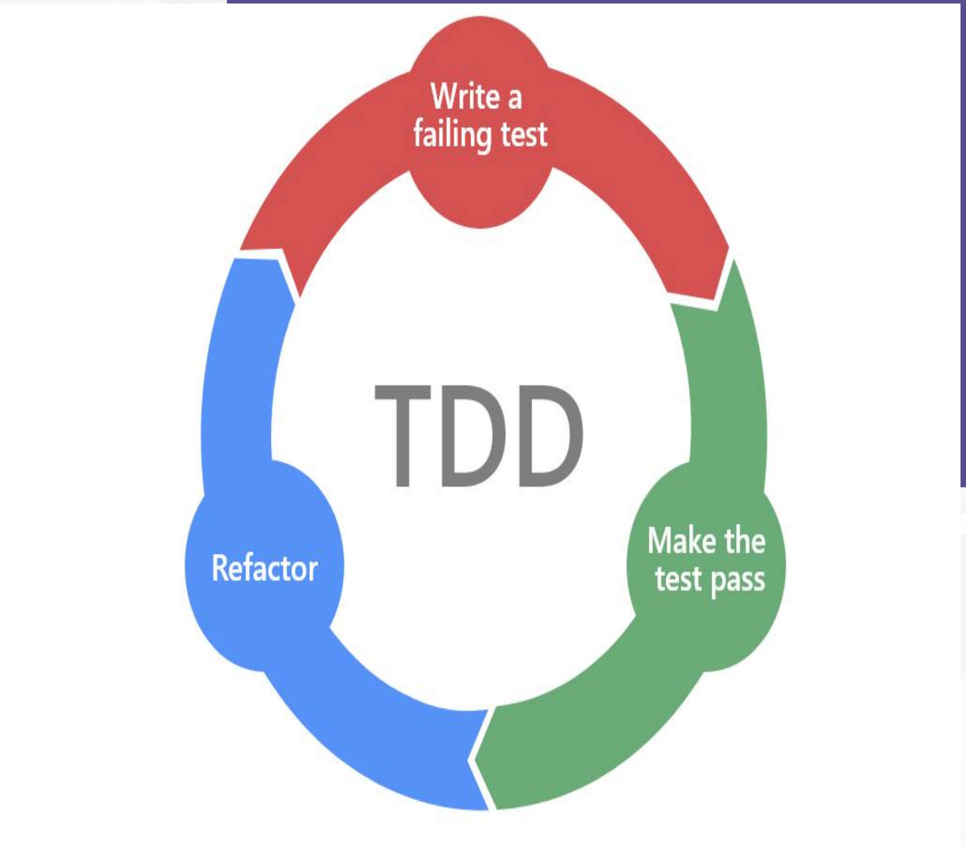
Dependency Inversion Principle (DIP):

- Depend on abstractions, not on concretions.

lab.py

```python
class MessageSender:
    def send_message(self, message):
        pass

class EmailSender(MessageSender):
    def send_message(self, message):
        print(f"Sending email: {message}")

class NotificationService:
    def __init__(self, sender):
        self.sender = sender

    def send_notification(self, message):
        self.sender.send_message(message)

email_sender = EmailSender()
notification_service = NotificationService(email_sender)
notification_service.send_notification("Hello, World!")
```

--A picture is worth a thousand words



04

Test-Driven Development

--A picture is worth a thousand words

Test-Driven Development (TDD) :

- Test-Driven Development (TDD) in Python is a software development approach where developers write tests for their code before writing the actual implementation.

- The TDD cycle typically consists of three steps:

  - Write a Test: Developers first write a test that defines the desired behavior of a function or a piece of code. These tests are often written using testing frameworks like unittest, pytest, or doctest in Python.

  - Run the Test (and Fail): Initially, the test fails since the corresponding code to fulfill its requirements hasn't been written yet.

  - Write Code to Pass the Test: Developers then write the minimal amount of code necessary to make the test pass. This means implementing the functionality required to meet the expectations defined by the test.

Test-Driven Development (TDD) :

- The goal of TDD is to ensure that code meets its requirements and behaves as expected, while also promoting code quality, maintainability, and flexibility.

--A picture is worth a thousand words

Adv of Test-Driven Development (TDD) :

- Improved Code Quality: TDD encourages developers to focus on writing clean, modular, and maintainable code since tests are written before the actual implementation.

- Rapid Feedback: TDD provides rapid feedback on the correctness of code changes. If a test fails, developers know immediately that they need to address the issue.

- Regression Testing: TDD ensures that existing functionality remains intact as new code is added or modified, reducing the likelihood of introducing regressions.

Adv of Test-Driven Development (TDD) :

- Increased Confidence: By having comprehensive test coverage, developers gain confidence in the stability and reliability of their codebase.

- Design Guided Development: TDD drives the design of code, leading to better architectural decisions and more modular, loosely coupled systems.

disadv of Test-Driven Development (TDD) :

- Initial Learning Curve: Developers may initially find it challenging to write tests before implementing the actual code.

- Time-Consuming: Writing tests upfront can be time-consuming, especially for complex systems.

- Over-Testing: There's a risk of writing too many tests, leading to test maintenance overhead.

--A picture is worth a thousand words

THANKS

--A picture is worth a thousand words