**Week 7 Lab 1 (**Task 1**)**

**SDLC (**software development lifecycle**)**

The **Software Development Life Cycle (SDLC)** is a structured process used for developing software. It defines the steps and stages required to build a high-quality software product systematically. Here's an overview of the common phases in SDLC:

**1. Planning**

- Objective: Identify project goals, feasibility, and resource requirements.

- Description: In this phase, stakeholders determine the scope of the project, budget, timeline, and necessary resources. It also involves identifying potential risks and outlining solutions.

**2. Requirements Analysis**

- Objective: Define what the software needs to do.

- Description: This phase involves gathering detailed information from stakeholders about the system’s functional and non-functional requirements. It creates a clear understanding of the software's purpose and functionality.

**3. Design**

- Objective: Create the architecture and design for the software solution.

- Description: The system’s architecture is defined, including the hardware and software layers, databases, user interfaces, and data flows. Technical specifications are created, laying the groundwork for coding.

**4. Development (Implementation)**

- Objective: Code the software based on design specifications.

- Description: Developers write the actual code for the system. This is where the software starts to take shape based on the previously defined designs and specifications. Coding languages, frameworks, and tools are chosen during this stage.

**5. Testing**

- Objective: Identify and fix bugs or issues in the software.

- Description: The software is rigorously tested to ensure it meets the requirements, is free of bugs, and works as intended. Various types of testing such as unit testing, integration testing, system testing, and user acceptance testing (UAT) are conducted.

**6. Deployment**

- Objective: Deliver the software to end users or clients.

- Description: Once the software has been tested and approved, it is deployed to the production environment. This phase may involve releasing the software in stages (e.g., beta releases) or all at once.

**7. Maintenance**

- Objective: Update, improve, and fix software after release.

- Description: After the software is deployed, it enters the maintenance phase. Any discovered bugs or issues are addressed, and updates or enhancements are made to keep the software functional and relevant to users’ needs.

**SDLC Models**

Various models of the SDLC exist, including:

- *Waterfall Model*: A linear, sequential approach.

- *Agile Model*: Iterative and incremental development.

- *V-Model*: Extension of the waterfall with a focus on testing at each stage.

- *Spiral Model*: Risk-driven, iterative approach for large projects.

Each model has its own advantages and is chosen based on the project’s complexity and requirements.

**SDLC Implementation in Week 6 Lab 3**

Implementing the **Software Development Life Cycle (SDLC)** for the provided Movie Rental System project involves following the SDLC phases to ensure that the software is well-structured, efficient, and maintainable. Here’s how each phase of SDLC can be applied to this code:

**1. Planning**

- Objective: Define the scope and purpose of the Movie Rental System.

- Implementation:

- Identify the core features required, such as adding movies, renting movies, returning movies, and listing available or rented movies.

- Define who will use the system (e.g., customers, store staff).

- Set goals such as ease of use and scalability (e.g., expanding the customer base or integrating a larger movie database).

**2. Requirements Analysis**

- Objective: Understand the functional and non-functional requirements.

- Implementation:

- Functional Requirements:

- The system must allow users to rent and return movies.

- It should maintain the availability status of movies.

- Users must be able to see their rented movies.

- Non-functional Requirements:

- The system should be easy to navigate with a clear menu interface.

- It should handle input errors (e.g., invalid choices or non-existent movies).

- Scalability to add more customers and movies as the store grows.

**3. Design**

- Objective: Architect the solution with modular, maintainable code.

- Implementation:

- Class Design: The system uses an object-oriented approach with classes such as `Movie`, `Customer`, and `RentalStore`. Each class encapsulates relevant data and methods, making the design modular and easy to manage.

- Design Decisions:

- A dictionary is used to store customers for quick retrieval.

- Movie objects store their availability, and customer objects store rented movies.

- Methods like `rent\_movie()` and `return\_movie()` in the `Customer` class allow clear division of responsibilities between classes.

**4. Development (Implementation)**

- Objective: Write the code to implement the design.

- Implementation:

- The provided code implements key features:

- Movies are added to the rental store with title, genre, and year.

- Customers can rent or return movies, and the system tracks the movie's availability.

- Testing during development: Since this phase often overlaps with the testing phase, the developer can implement the basic functionality and manually test each feature as it’s built.

**5. Testing**

- Objective: Ensure the system works as expected and identify any bugs or issues.

- Implementation:

- Unit Testing: Each function can be tested individually. For example:

- Test if a movie becomes unavailable once rented and available again after being returned.

- Test edge cases, such as trying to rent a movie that’s already rented or returning a movie not rented by the user.

- Integration Testing: Ensure that the interaction between classes works correctly (e.g., renting a movie affects both the movie’s availability and the customer’s rented list).

- Error Handling: Ensure that input validation works, such as for movie titles, invalid menu choices, and missing customers.

**6. Deployment**

- Objective: Release the system for real-world use.

- Implementation:

- Once tested and validated, the code can be deployed for use by the movie rental store. The deployment could be in the form of a command-line interface (CLI), as seen in the code, or later adapted into a web or mobile application for wider accessibility.

**7. Maintenance**

- Objective: Continuously update and improve the system post-deployment.

- Implementation:

- Bug Fixes: Regularly monitor and fix bugs reported by users (e.g., incorrect movie availability status or failure to retrieve customers).

- Feature Enhancements: As the rental store grows, new features could be added, such as:

- Search functionality to find movies by genre, year, or director.

- A recommendation system for users based on rental history.

- A loyalty points system for frequent customers.

- Refactoring: As the system grows, the code may need to be optimized for better performance and readability, such as using databases to store movies and customers rather than lists and dictionaries.

SDLC Model in this Context

In this case, you might choose an Agile or Iterative SDLC model because the project can evolve over time with feedback from users. Initial deployment can start with core features (renting and returning movies), while later iterations can introduce more advanced features based on user needs.

By following these SDLC phases, the Movie Rental System will be built with a clear structure, ensuring that it meets user requirements and is adaptable for future needs.

**Week 7 Lab 1 (**Task 2**)**

The **Software Development Life Cycle (SDLC**) is a process that guides the development of a software product in a structured and methodical manner. It consists of several phases, and applying these principles to the code shown would help improve its design, quality, maintainability, and scalability.

Here's how **SDLC** could be applied to this code:

**1. Requirement Analysis**   - Current Code: The functionality is not very clear, but the code is doing basic mathematical operations (addition, multiplication) and creating some placeholder classes for shapes.  
   - SDLC Application: You need to gather requirements about what this code is supposed to do. Is this a calculator app? Does it require shape objects with area calculations? Is the legacy function needed?  
     - Define the purpose: Why does the code need to support `InflexibleShape`, `Circle`, `Square`, and `legacy\_function`?  
     - Define what the client expects: For example, the user might want more complex shapes or additional features like division, subtraction, etc.

**2. Design**  
   - Current Code: The design is unstructured. The `Calculator` class does simple operations but lacks functionality that could make it extensible.  
   - SDLC Application: After requirements, you can create a more robust \*\*design\*\*:  
     - Class Design: You can split responsibilities better. For instance, create a `MathOperations` class that handles basic operations, and a `Shape` class that deals with geometry, maybe implementing polymorphism or interfaces.  
     - Single Responsibility Principle (SRP): Each class should have a single, well-defined responsibility. The `legacy\_function` should not exist in the `Calculator` class because it violates SRP.

   Example Design:  
   - A `Shape` superclass that defines `calculate\_area` method for all shapes.  
   - A `Circle` and `Square` class that overrides this method to provide specific area calculations.  
   - A `MathOperations` class that supports all the basic arithmetic operations.

**3. Implementation (Coding)**  
   - Current Code: The current code has basic functionalities but lacks a robust implementation of shape areas and does not handle edge cases (like zero values or wrong inputs).  
   - SDLC Application:  
     - Better Use of OOP Principles\*\*: Implement the `calculate\_area` method for `Circle` and `Square`. For example:  
       ```python  
       class Circle(InflexibleShape):  
           def \_\_init\_\_(self, radius):  
               self.radius = radius  
  
           def calculate\_area(self):  
               return 3.1416 \* self.radius \*\* 2  
  
       class Square(InflexibleShape):  
           def \_\_init\_\_(self, side):  
               self.side = side  
  
           def calculate\_area(self):  
               return self.side \*\* 2  
       ```  
     - Input Validation: Implement checks in your code to make sure inputs are valid (e.g., positive numbers for dimensions).

**4. Testing**  
   - Current Code: No testing is done. The code just prints out results.  
   - SDLC Application:  
     - Unit Testing: Write test cases for each method to ensure they work correctly. For example, test if the addition works for all cases, and test the area calculation for different shapes.  
     - Use a framework like `unittest` or `pytest`:  
       ```python  
       import unittest  
       class TestCalculator(unittest.TestCase):  
           def test\_add(self):  
               calc = Calculator()  
               self.assertEqual(calc.add(2, 3), 5)  
  
           def test\_multiply(self):  
               calc = Calculator()  
               self.assertEqual(calc.multiply(2, 3), 6)  
       ```  
     - Test Coverage: Ensure that all code paths (e.g., all functions) are tested.  
     - Test for \*\*edge cases\*\* (like negative numbers, zero values, very large values).

**5. Deployment**  
   - Current Code: There's no provision for deployment or packaging.  
   - SDLC Application:  
     - Prepare your code for deployment by packaging it into a module.  
     - Use version control (e.g., Git) to manage changes and ensure deployment consistency.  
     - Ensure that external libraries are properly managed in `requirements.txt` or another dependency management tool.

**6. Maintenance**  
   - Current Code: The code is not structured for easy modification or updates.  
   - SDLC Application:  
     - Code Documentation: Ensure that all methods are well-documented, explaining their inputs and outputs.  
     - Refactoring: Keep the code flexible for future updates. As new features or shapes are introduced, add them easily without breaking the existing code.  
     - Bug Tracking: Implement a system to track bugs and ensure they are fixed in future releases.

---

Example: Improved Code After SDLC Application

```python  
class MathOperations:  
    def add(self, a, b):  
        return a + b  
  
    def multiply(self, a, b):  
        return a \* b

class Shape:  
    def calculate\_area(self):  
        raise NotImplementedError("Subclass must implement abstract method")

class Circle(Shape):  
    def \_\_init\_\_(self, radius):  
        self.radius = radius

    def calculate\_area(self):  
        return 3.1416 \* self.radius \*\* 2

class Square(Shape):  
    def \_\_init\_\_(self, side):  
        self.side = side

    def calculate\_area(self):  
        return self.side \*\* 2

Main code with user input or application flow  
if \_\_name\_\_ == "\_\_main\_\_":  
    calc = MathOperations()  
    print(f"Addition: {calc.add(5, 10)}")  
    print(f"Multiplication: {calc.multiply(5, 10)}")

    circle = Circle(7)  
    square = Square(4)  
    print(f"Circle Area: {circle.calculate\_area()}")  
    print(f"Square Area: {square.calculate\_area()}")  
```

---

By applying SDLC, you can systematically improve your code, ensuring it meets requirements, is well-designed, thoroughly tested, and easy to maintain and extend