**TASK:1**

1. The Software Development Life Cycle (SDLC) is a structured process used for software creation and maintenance. Each phase is clearly defined, with progression occurring only after the completion of the previous phase. The primary purpose of this life cycle is to ensure software quality. The SDLC consists of six phases: Planning, Analysis, Design, Implementation, Testing, and Maintenance.

**Phase 1: Planning**  
This crucial initial phase involves companies identifying solutions to their problems through software development while considering their requirements and available resources.

**Phase 2: Analysis**  
During this stage, a preliminary understanding of the software is developed. Discussions among IT support and management help clarify the specific needs of the company from software developers.

**Phase 3: Design**  
This phase focuses on the application’s visual layout and its internal operations. Developers seek feedback to refine the software design, aiming to create a well-structured model.

**Phase 4: Implementation**  
In this phase, code is generated with an emphasis on quality and efficiency, ensuring it meets management expectations and supports product growth.

**Phase 5: Testing**  
This stage involves a thorough examination of the code to maintain quality, with developers checking each line for errors and ensuring the software functions correctly.

**Phase 6: Maintenance**  
At this point, the software is ready for user deployment. This phase remains important as users can report issues, prompting ongoing support from developers.

**2. Applying SDLC to Movie Rental Systems**

Utilizing the SDLC for a Movie Rental System involves enhancing the entire process from planning to maintenance. Here’s how to apply SDLC principles step-by-step:

1. **Requirement Analysis**  
   In this phase, we define system objectives and gather requirements for the Movie Rental System:
   * **Functional Requirements:**
     + Enable customers to rent and return movies.
     + Maintain lists of available and rented movies.
     + Allow customers to manage their rented lists.
     + Facilitate the addition of new customers and movies.
     + Prevent renting movies that are already rented.
   * **Non-functional Requirements:**
     + Ensure the system is user-friendly.
     + Handle invalid inputs effectively.
     + Scale to accommodate larger customer and movie databases.
2. **Design**  
   This phase focuses on creating a detailed system blueprint, considering both high-level and low-level designs.
   * **Class Design:**
     + Classes like Movie, Customer, and Rental Store can be improved.
     + **Enhancements:**
       - Implement data validation in these classes for accuracy (e.g., checking year formats).
       - Introduce a Unique ID system to prevent duplicate entries.
   * **Error Handling:**
     + Establish error handling for unavailable movies and invalid inputs.
   * **Modularization:**
     + Refactor user interaction and main logic into separate modules for better scalability.
3. **Implementation**  
   This phase involves coding based on design decisions.
   * **Current Code:**
     + The existing code supports basic functionalities but lacks proper validation and error handling.
   * **Improvements:**
     + Add input validation to ensure valid entries for years, movie titles, and customer existence.
4. **Testing**  
   Testing is vital to ensure the system performs as expected.
   * **Unit Testing:**
     + Use testing frameworks like unit test or pytest to create unit tests for all methods in the Movie, Customer, and Rental Store classes.
     + Test scenarios such as:
       - Renting an available movie.
       - Attempting to rent an unavailable movie.
       - Returning a movie that wasn't rented.
       - Adding and finding movies/customers.
5. **Deployment**  
   Once testing is complete, the system is ready for deployment.
   * **Packaging:**
     + Package the system as an executable or an installable Python package using setuptools for easy use.
     + Ensure compatibility across different environments (e.g., Linux, Windows).
   * **Version Control:**
     + Utilize Git for tracking changes, managing features, and collaborating with developers.
6. **Maintenance**  
   Post-deployment, continuous maintenance is essential for functionality and relevance.
   * **Bug Fixes:**
     + Monitor for bugs in production and develop patches as necessary.
   * **Feature Upgrades:**
     + As the business expands, consider adding features such as:
       - Support for multiple rental locations.
       - Online rentals with a user interface.
       - Improved capacity for larger databases by transitioning to a database backend.

**Task 2: Software Design Principles**

There are ten key software design principles:

1. **KISS (Keep It Simple, Stupid)**  
   This principle emphasizes simplicity in design to enhance user accessibility.
2. **DRY (Don’t Repeat Yourself)**  
   Developers should write code once to avoid duplication and potential errors.
3. **Open/Closed Principle**  
   Code should be open for extension but closed for modification, allowing enhancements without altering existing code.
4. **Composition over Inheritance**  
   This principle prioritizes composition to minimize complexity by creating instances with distinct behaviors rather than relying on inheritance.
5. **Single Responsibility Principle**  
   Each class should focus on a single function to avoid complications and errors.
6. **Separation of Concerns**  
   Code should operate independently to prevent overlap, with one piece completed before moving to the next.
7. **YAGNI (You Aren’t Gonna Need It)**  
   Focus on current requirements, avoiding unnecessary features that could introduce errors.
8. **Avoid Premature Optimization**  
   Ensure the code runs correctly before optimizing for performance, which can complicate initial functionality.
9. **Refactor, Refactor, Refactor**  
   Maintain clarity and organization in code to avoid confusion and errors.
10. **Clean Code > Clever Code**  
    Prioritize clean, understandable code over clever but convoluted solutions.

**Identifying Violations of Principles in Code**

1. The UnstructuredCode class includes a method xyzzy for basic addition that has a convoluted name and purpose. Additionally, the legacy\_function appears unnecessary and could be integrated into a better-structured class.
2. The Calculator class features both add and multiply methods that are essentially redundant, indicating code duplication.
3. The InflexibleShape class and its subclasses (Circle, Square) contain empty methods, complicating the process of adding new shapes without modifying existing code.
4. The Circle and Square classes inherit from InflexibleShape but lack implementations for the calculate\_area method, showing improper use of inheritance.
5. The UnstructuredCode class overlaps with the Calculator class in functionality, breaching the single responsibility principle.
6. The functionality of UnstructuredCode and Calculator overlaps, violating the separation of concerns.
7. The InflexibleShape class defines a calculate\_area method that is not implemented in its subclasses, resulting in unnecessary code.
8. While there is no direct evidence of premature optimization, the existence of functions like legacy\_function is unwarranted without a clear optimization need.
9. The code's structure, characterized by repeated logic and inappropriate inheritance, contributes to messiness and maintenance challenges.
10. Methods like xyzzy and legacy\_function lack intuitiveness, complicating readability and maintainability.