**Task 1: What do you think is the need for Refactoring?**

* **Answer**: Refactoring is necessary to improve the readability, maintainability, and performance of the code. It helps to eliminate code smells, reduce complexity, and improve the overall design. Refactoring also makes it easier to add new features and reduces the likelihood of introducing bugs.

**Task 2: What are the Principles of Refactoring?**

* **Answer**:
  + **Simplicity**: Simplify the code to make it easier to understand and maintain.
  + **Eliminate Duplication**: Avoid repeating the same code; use abstraction to make the code cleaner.
  + **Minimize Dependencies**: Reduce the coupling between different modules or classes.
  + **Enhance Readability**: Make code easier to read and understand.
  + **Preserve Behavior**: Ensure that the refactor does not break existing functionality.

**Task 3: What are the steps for performing code refactoring?**

* **Answer**:
  1. **Identify the code that needs refactoring** (e.g., code smells, large methods, duplication).
  2. **Write tests** to ensure that the existing behavior is preserved after the refactor.
  3. **Refactor incrementally**, focusing on small changes at a time.
  4. **Run tests** after each change to verify correctness.
  5. **Refactor again if necessary**, iterating over the code to optimize and improve it further.

**Task 4: What makes Composite pattern useful when designing complex tree structures?**

* **Answer**:
  + **Correct Answer**: 2. Allows treating individual objects and compositions uniformly through a common interface.
  + **Explanation**: The Composite pattern simplifies working with tree structures by allowing clients to treat individual objects and collections of objects (compositions) uniformly.

**Task 5: Identify the code smell in the following class:**

public class Order {

private String orderid;

private String customerName;

private String customerAddress;

private String customerPhone;

public String getOrderId() {

return orderid;

}

public void setOrderId(String orderid) {

this.orderid = orderid;

}

public String getCustomerName() {

return customerName;

}

public void setCustomerName(String customerName) {

this.customerName = customerName;

}

public String getCustomerAddress() {

return customerAddress;

}

public void setCustomerAddress(String customerAddress) {

this.customerAddress = customerAddress;

}

public String getCustomerPhone() {

return customerPhone;

}

public void setCustomerPhone(String customerPhone) {

this.customerPhone = customerPhone;

}

}

* **Answer**: **Large Class**

**Task 6: In the context of the Three-tier architecture, what role does the 'Business Logic Layer' play?**

* **Answer**:
  + **Correct Answer**: 2. It processes commands from the user interface, performs validations, and implements the core functional logic.

**Task 7: What is the role of Packages in representing subsystems?**

* **Answer**:
  + **Correct Answer**: 2. Packages group related elements and can be used to modularize large systems into manageable subsystems with defined interfaces.

**Task 8: You are building a system that maintains a cache of user sessions. Which implementation is the most thread-safe and efficient?**

public class SCache {

private static volatile SCache instance;

private SCache() {}

public static SCache getInstance() {

if (instance == null) {

synchronized (SCache.class) {

if (instance == null) {

instance = new SCache();

}

}

}

return instance;

}

}

* **Answer**:
  + **Correct Answer**: 2. Uses double-checked locking Singleton, ensures lazy and thread-safe initialization.

**Task 9: Identify the code smell in the following class:**

public class Customer {

private String name;

private String address;

private String phoneNumber;

public void printCustomerDetails() {

System.out.println("Name: " + name);

System.out.println("Address: " + address);

System.out.println("Phone Number: " + phoneNumber);

}

}

* **Answer**: **Long Method**

**Task 10: What principle is violated in the payment system interface, and how would you improve it?**

* **Answer**:
  + **Correct Answer**: 4. Interface Segregation Principle is violated. Split the interface into more specific ones for better adherence to roles.

**Task 11: What major design issue exists in the following class hierarchy, and how would you refactor it?**

class Notification {

public void send(String message) {

System.out.println("Sending generic notification: " + message);

}

}

class EmailNotification extends Notification {

@Override

public void send(String message) {

System.out.println("Sending email: " + message);

}

}

class SMSNotification extends Notification {

@Override

public void send(String message) {

throw new UnsupportedOperationException("SMS not supported");

}

}

* **Answer**:
  + **Correct Answer**: 2. Violates Liskov Substitution Principle: use interfaces and split behaviors per notification type.

**Task 12: What is a key benefit of using the Facade design pattern in application architecture?**

* **Answer**:
  + **Correct Answer**: 4. It simplifies access to a complex system by providing a unified interface over a set of interfaces in a subsystem.

**Task 13: How does the Proxy Design Pattern support performance or access control?**

* **Answer**:
  + **Correct Answer**: 3. It provides a placeholder to control access to another object, often adding lazy loading, access control, or caching.

**Task 14: Which of the following best represents the "Open/Closed Principle" from the SOLID principles?**

* **Answer**:
  + **Correct Answer**: 2. Entities should be open for extension through mechanisms like inheritance or composition, but closed for modification to avoid breaking existing behavior.

**Task 15: What distinguishes the Builder pattern from the Prototype pattern in object creation?**

* **Answer**:
  + **Correct Answer**: 2. The Builder pattern separates the construction of a complex object from its representation, while Prototype allows creation of duplicate objects by copying an existing one.

**Task 16: What is the first approach to improving stability and maintainability in a legacy insurance product?**

* **Answer**:
  + **Correct Answer**: 3. Refactor classes to follow the Single Responsibility Principle and identify code smells.

**Task 17: What anti-pattern or refactoring opportunity is present in the following code?**

class UserManager {

public void processUser(String username) {

if (username.equals("admin")) {

// Admin-specific logic

} else if (username.equals("guest")) {

// Guest-specific logic

} else {

// Default logic

}

}

}

* **Answer**:
  + **Correct Answer**: 1. The method violates the Open-Closed Principle, consider using polymorphism instead of hard-coded conditions.

**Task 18: How should you model behavior in a microservice-based inventory system with dependent services?**

* **Answer**:
  + **Correct Answer**: 3. Use asynchronous messaging with Publish Subscribe to notify downstream services.

**Task 19: Which architectural model should be applied to a logistics platform with high scalability requirements?**

* **Answer**:
  + **Correct Answer**: 2. Use a 3-tier Architecture to decouple UI, Business, and Data layers.

**Task 20: What characteristic of a well-written unit test makes it valuable in Test Driven Development?**

* **Answer**:
  + **Correct Answer**: 4. It should be repeatable, focused on a single responsibility, and clearly define expected outcomes for each condition.

**Task 21: How can the test suite be improved to catch real-world issues when it mostly validates getters, setters, and trivial logic?**

* **Answer**:
  + **Correct Answer**: 2. Refactor tests to cover cases, boundary conditions, and business logic paths.

**Task 22: What pattern is most appropriate for a financial analytics platform that needs to fetch data from multiple sources?**

* **Answer**:
  + **Correct Answer**: 2. Use the Strategy Pattern to encapsulate source-specific logic and switch at runtime.

**Task 23: How should tightly coupled modules in a distributed messaging system be decoupled efficiently?**

* **Answer**:
  + **Correct Answer**: 2. Use the Publish Subscribe Pattern to decouple producers from consumers.

**Task 24: Debate on:**

* **Topic**: Large Scale Systems, Load Balancers, Async Programming

**Home Task:**

**Task 01**:

* **Objective**: Implement a SortingStrategy class with two concrete strategies:
  + Sorting alphabetically (case-insensitive)
  + Sorting by length
  + Use interfaces for sorting methods
  + Methods: setStrategyForSorting, addItems, performSort, getList

**Input**:

* Stanford
* Ankit
* Watson

**Output**:

* Alphabetical Sorting:

Top of Form

Attach

Search

Study

Voice

Bottom of Form