# **Day 27 - 23 August 2025**

**Document Name:**Day 27 - hmuvvala@ - Hari Gopal Muvvala

### Task 1 — Need for Refactoring

* Improve **readability** and **code clarity**
* Reduce **technical debt** and code **smells**
* Increase **maintainability** and **extensibility**
* Enable **faster debugging** and **easier onboarding**
* Facilitate **reusability** and **modular design**
* Keep **external behavior unchanged** while improving internals
* Prepare code for **new features** and **performance** tweaks

### Task 2 — Principles of Refactoring

* **Behavior‑preserving:** functionality must not change
* **Small, incremental changes:** one safe step at a time
* **Continuous testing:** keep tests green after each change
* **High cohesion, low coupling:** tighten related code; reduce dependencies
* **Simplicity first:** prefer clear, straightforward solutions
* **Single Responsibility:** one reason to change per module/class
* **DRY:** eliminate duplication
* **Encapsulation:** hide internals; expose clean interfaces
* **Consistent naming & style:** improve semantics and conventions

### Task 3 — Steps to Perform Code Refactoring

1. **Identify smells** (duplication, long method, large class, shotgun surgery, etc.)
2. **Protect with tests** (add/expand unit tests around current behavior)
3. **Choose a refactoring** (Extract Method/Class, Rename, Inline, Move, Introduce Parameter Object, etc.)
4. **Apply small change** (one transformation at a time)
5. **Run tests** (ensure behavior is intact; fix if red)
6. **Review & clean up** (naming, comments, formatting)
7. **Commit** (small commits with clear messages)
8. **Repeat** until smells are removed or risk outweighs benefit

### Task 4

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

1. It replaces the use of collections to store children
2. **Allows treating individual objects and compositions uniformly through a common interface.** ✅
3. It automatically serializes tree objects for persistence
4. Optimizes memory by removing duplicate nodes in the tree

**Reasoning:** Composite lets you work with leaf objects and composite objects in the same way via a common interface.

### Task 5

**Identify the code smell:**

public class Order {

private String orderid;

private String customerName;

private String customerAddress;

private String customerPhone;

public String getOrderld() {

return orderid;

}

public void setOrderld(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;

}

}

Options:

1. Long Method
2. Primitive Obsession
3. **Large Class** ✅
4. Feature Envy

**Reasoning:** The class mixes order and customer details, making it too large. Should be split into smaller classes.

### Task 6

**In the context of the Three-tier architecture, what role does the ‘Business Logic Layer’ play?**

1. It is responsible for managing physical data storage and retrieval mechanisms from database systems.
2. **It processes commands from the user interface, performs validations, and implements the core functional logic.** ✅
3. It defines how the system behaves under network traffic and handles load balancing.
4. It renders the UI elements and sends them directly to database procedures for execution.

**Reasoning:** The business layer handles processing, validations, and rules — the “brain” of the application.

### Task 7

**What is the role of Packages in representing subsystems?**

1. Packages are used only to store deprecated classes for backward compatibility.
2. **Packages group related elements and can be used to modularize large systems into manageable subsystems with defined interfaces.** ✅
3. Packages represent reusable libraries only and are not part of design architecture.
4. Packages define the runtime performance model of subsystems.

**Reasoning:** Packages provide modularization and allow structuring a large application.

### Task 8

**System maintaining a cache of user sessions, must be global, initialized once, lazy and thread-safe. Which implementation is correct?**

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;

}

}

Options:

1. Implements Command pattern for caching logic.
2. **Uses double checked locking Singleton, ensures lazy and thread-safe initialization.** ✅
3. Applies Factory pattern with static holder.
4. Uses Prototype pattern with unnecessary locking.

**Reasoning:** Double-checked locking is the efficient and thread-safe lazy initialization approach.

### Task 9

**Identify the code smell:**

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);

}

}

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);

}

}

Options:

1. Long Method
2. Primitive Obsession
3. Large Class
4. **Feature Envy** ✅

**Reasoning:** Printing logic does not belong in the Customer entity. Responsibility should be moved to a separate class (UI/Printer).

### Task 10

**Consider the following payment system. What principle is violated?**

interface PaymentService{

void makePayment();

void cancelPayment();

void generateInvoice();

}

class CreditCardPayment implements PaymentService {

@Override

public void makePayment() { /\* Implementation \*/ }

@Override

public void cancelPayment() { /\* Implementation \*/ }

@Override

public void generateInvoice() {

// Not applicable for credit card

}

}

Options:

1. Liskov Substitution Principle is violated due to missing default behavior.
2. Dependency Inversion is violated, introduce abstraction for the payment handler.
3. Open Closed Principle is violated by not supporting extension for other payment types.
4. **Interface Segregation Principle is violated — split the interface into more specific ones for better adherence to roles.** ✅

**Reasoning:** A class is forced to implement methods it does not need. Interfaces should be smaller and role-specific.

### Task 11

**Consider the following Notification hierarchy. What design issue exists?**

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");

}

}

Options:

1. Violates Interface Segregation, merge all notifications into one abstract class.
2. **Violates Liskov Substitution Principle: use interfaces and split behaviors per notification type.** ✅
3. No issue, the design is extensible and allows overriding.
4. Follows Open-Closed Principle; hence no refactoring is needed.

**Reasoning:** Subclass (SMSNotification) breaks parent contract. LSP requires substitutable behavior. Should use an interface Notification and separate valid implementations.

### Task 12

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

1. It provides a way to eliminate middle layers and reduce abstraction in software components
2. It allows access to the low-level subsystems directly for debugging and testing
3. It offers a mechanism for injecting multiple implementations into a core algorithm dynamically
4. **It simplifies access to a complex system by providing a unified interface over a set of interfaces in a subsystem** ✅

**Reasoning:** Facade hides subsystem complexity and exposes a single unified interface.

### Task 13

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

1. It executes logic inside core components without any delegation
2. It logs method calls without executing them
3. **It provides a placeholder to control access to another object, often adding lazy loading, access control, or caching** ✅
4. It permanently replaces the original object with a faster mock implementation

**Reasoning:** Proxy acts as a surrogate, adding features like lazy initialization, caching, or security checks.

### Task 14

**Which of the following best represents the Open/Closed Principle?**

1. Software components should be designed to be open for direct modification but closed to extension for maintaining rigidity
2. **Entities should be open for extension through mechanisms like inheritance or composition, but closed for modification to avoid breaking existing behavior** ✅
3. Code should be able to accept runtime parameter changes without altering any class behavior or interface
4. Code must be completely static to avoid any modification or future maintenance overhead

**Reasoning:** OCP = extend behavior without modifying existing tested code.

### Task 15

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

1. The Builder pattern focuses on shallow copying of objects while Prototype deals with constructing complex objects step by step
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** ✅
3. The Builder pattern helps clone objects quickly whereas Prototype builds objects using various helper methods
4. The Builder and Prototype serve similar purposes but Builder is used at compile time and Prototype at runtime

**Reasoning:** Builder → step-by-step construction; Prototype → cloning.

### Task 16

**Legacy insurance system with unstable modules. What’s the first approach?**

1. Merge related classes into one for tighter control
2. Rewrite all modules from scratch using latest Java frameworks
3. **Refactor classes to follow the Single Responsibility Principle and identify code smells** ✅
4. Move business logic to the frontend to reduce complexity in backend

**Reasoning:** Safe starting point is SRP + refactoring, not rewriting everything.

### Task 17

**Analyze UserManager code. What anti-pattern/refactoring opportunity exists?**

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

}

}

1. **The method violates the Open Closed Principle, consider using polymorphism instead of hard-coded conditions** ✅
2. No refactoring is required since all roles are covered
3. The method property uses polymorphism by branching based on user roles
4. The logic should be moved to the database to improve separation of concerns

**Reasoning:** Adding new roles requires modifying method → violates OCP. Use polymorphism.

### Task 18

**Microservice system where product updates must notify multiple services independently.**

1. Use a centralized database to keep all services in sync
2. Implement direct service-to-service RPC calls on update
3. **Use asynchronous messaging with Publish-Subscribe to notify downstream services** ✅
4. Add retry logic in all dependent services for error recovery

**Reasoning:** Pub-Sub enables decoupling and async notifications.

### Task 19

**Logistics company platform must scale to millions. Separate data, logic, and presentation.**

1. Use Decorator to wrap all business logic for better scaling
2. **Use a 3-tier Architecture to decouple UI, Business, and Data layers** ✅
3. Implement Singleton in each layer to reduce memory usage
4. Implement Proxy classes to replace all direct DB interactions

**Reasoning:** Three-tier architecture supports independent scaling and separation of concerns.

### Task 20

**What characteristic of a well-written unit test makes it valuable in TDD?**

1. It should test only one method but involve multiple objects and rely on external systems
2. It must execute complex test scenarios using mock networks and full integrations
3. It should be independent of the code and unrelated to the software behavior
4. **It should be repeatable, focused on a single responsibility and clearly define expected outcomes for each condition** ✅

**Reasoning:** TDD tests must be deterministic, simple, and focused.

### Task 21

**High coverage but bugs remain. Tests only validate trivial logic. How to improve?**

1. Add more assertions to the existing tests without changing test focus
2. **Refactor tests to cover edge cases, boundary conditions, and business logic paths** ✅
3. Migrate unit tests to performance tests
4. Replace unit tests with mocks to simulate data better

**Reasoning:** Tests should target meaningful business logic, not trivial getters/setters.

### Task 22

**Financial analytics platform needs to fetch from multiple sources (APIs, files, DB), extensible design.**

1. Use Singleton to manage shared resource access to these sources
2. **Use Strategy Pattern to encapsulate source-specific logic and switch at runtime** ✅
3. Use Prototype to clone existing logic for each data source
4. Use Decorator Pattern to layer additional features on top of each data source

**Reasoning:** Strategy pattern encapsulates algorithms/sources and allows easy extension.

### Task 23

**Distributed messaging system with tight coupling between producers and consumers.**

1. Introduce direct REST calls instead of asynchronous messaging
2. **Use the Publish-Subscribe Pattern to decouple producers from consumers** ✅
3. Add shared database access between both modules
4. Use Adapter Pattern to hide implementation details

**Reasoning:** Pub-Sub decouples producer from consumer at runtime.