Institute of Information Technology
University of Dhaka

Design Patterns
SE - 2215

Analysis and Case Study Report

Submitted By
Sadia Akter Snigdha (BSSE 1513)

Submitted To
Mridha Md. Nafis Fuad
Lecturer - IIT, DU

Ans to the ques no: 'Part 1'

Design patterns provide proven, reusable approaches to solving common software design problems, but their effectiveness depends heavily on specific context. When misapplied or overused, they can reduce code clarity, hurt performance, and make long-term maintenance more difficult.

1. Builder Pattern - Unnecessary Complexity Trap

Pattern intend:
Separate the construction of a complex object from representation, allowing the same construction process to create different representations.

Misuse Scenario:
context: A simple DTO(Data Transfer Object) for holding user profile data in a small REST API is built using the Builder pattern.

Problem:

- Requires multiple lines of code to create a simple object.
- Increase boilerplate with builder classess and methods for trivial data.
- Reduces readability compared to straightforward constructor or record usage.

Why it's ineffective:

Builder works well for large, optional-parameter heavy objects. For small, stable objects: it adds unnecessary ceremony.

Antipattern Name: Over-Engineering with Fluent Builder.

2. Strategy Pattern - Configuration Overload

Pattern Goal:
Define a family of algorithms, encapsulate each one, and make them interchangeable at runtime

Misuse Scenario:
Context: A small calculator app uses Strategy for basic operations (add, subtract, multiply, divide)

Problem:
- Introduces multiple small classes or Lambdas for trivial logic.
- Makes maintenance harder when all algorithms are simple enough to inline.
- Adds extra indirection, slowing down onboarding for new developers.

Why it's ineffective:
When algorithms are unlikely to change and are
.

inherently simple, strategy just bloats the code.

Antipattern name: Pattern obsession

3. Decorator Pattern - Layers of Confusion.

Pattern Goal:
Attach additional responsibilities to an object dynamically without modifying its structure.

Misuse Scenario:
Context: A basic file I/O wrapper system uses Decorators for logging, compression, encryption - even for trivial features like trimming white-space.

Problem:
- Creates deeply nested wrapper objects that are hard to debug.
- Makes stack ~~to~~ traces unreadable.
- Causes performance hits multiple small operations being wrapped repeatedly.

Why it's ineffective:
When used excessively for minor responsibilities the pattern turns ~~for~~ from elegant flexibility into a maintenance nightmare.

Antipattern name: Over-Decorating.

Core Reasons Patterns Fail:

1. Lack of context Awareness - Pattern chosen without consideration of scale, constraints, or probability of future change.

2. Premature Abstraction - Designing solutions for potential problems that may never arise.

3. Rigid Adherence - Treating patterns as rules rather than guidelines.

Answer to the question no ‘Part2’

Design Patterns Case Study : OpenRefine

Project Overview

Project: OpenRefine
Repository: https://github.com/OpenRefine/OpenRefine

Short Description of the Project

OpenRefine is a free, open-source Java tool made for handling messy data. It allows users to load, explore, clean, transform, reconcile, and enhance datasets through a web browser running locally on their computer. This setup ensures privacy and ease of use. Originally created by Metaweb Technologies, which was later acquired by Google, OpenRefine has become a community-driven project supported by Code for Science and Society (CS&S).

Identified Patterns

Based on the project's architecture and source code structure, the following design patterns are used:

• Command Pattern (Behavioral pattern)

• Observer Pattern (Behavioral pattern)

• Facade Pattern (Structural pattern)

• Memento Pattern (Behavioral pattern)

• Factory Pattern (Creational pattern)

1. Command Pattern

Primary Classes:

- Command.java (and subclasses in com.google.refine.commands.*) - Encapsulates requests as objects
- AbstractOperation.java - Represents generalizable processes for history and serialization

The Command pattern encapsulates user actions and data transformations as executable objects, used extensively for routing requests and managing history.

Comparison with Standard Command Pattern

Similarities:

- Encapsulates requests: Each command object contains all information needed to perform an action
- Undo/redo support: Commands can be stored in history for reversal
- Decouples sender from receiver: Client requests are routed without direct knowledge of implementation

Key Modifications:

- Integration with history: Commands produce Change objects for diffs, enabling ordered application/reversion
- Long-running support: Distinguishes immediate vs. queued processes for asynchronous execution
- Serialization: Commands tied to operations for reproducibility and export

Benefits:

- Flexible workflow: Supports complex, reversible data transformations
- Reproducibility: Allows exporting/applying command sequences to other datasets
- Maintainability: New commands can be added without altering core routing logic

2.Observer Pattern

Primary Classes:

- Engine configuration (client-side) - Manages facet states
- History.java and HistoryEntry.java - Tracks changes for notifications

The Observer pattern is used for reactive updates, such as facet refreshes when data changes.

Comparison with Standard Observer Pattern

Similarities:

- Subject-observer relationship: Data model notifies views (facets) of state changes
- Loose coupling: Observers register for specific events without tight integration
- Event-driven: Changes trigger updates in dependent components

Key Modifications:

- Thread safety in v4: Immutable grids ensure consistent observation across threads
- Dependency tracking: Column dependencies modeled for efficient notifications

Benefits:

- Real-time feedback: Users see immediate effects of operations in facets
- Performance: Only affected views update, avoiding full redraws
- Concurrency: Supports long-running operations without blocking UI

3.Facade Pattern

Primary Classes:

- RefineServlet.java - Unified interface for all requests
- ProjectManager.java - Simplifies project loading/saving

The Facade pattern provides a simplified interface to complex subsystems, like request handling.

Comparison with Standard Facade Pattern

Similarities:

- Single entry point: Hides subsystem complexity behind one class
- Subsystem delegation: Routes to commands or managers
- Decouples clients: UI interacts via facade without knowing internals

Key Modifications:

- Embedded in servlet: Tied to web server for local HTTP access
- State management: Facade coordinates in-memory and disk persistence
- Extensible: Allows extension injection through the facade

Benefits:

- Simplified API: Easier for clients and third-parties to interact
- Maintainability: Changes to subsystems don't affect facade users
- Performance: Central routing optimizes request handling

4.Memento Pattern

Primary Classes:

- Change.java - Stores diffs for state restoration
- History.java - Manages sequence of mementos (HistoryEntry)

The Memento pattern captures and restores object states, central to undo/redo in OpenRefine's history.

Comparison with Standard Memento Pattern

Similarities:

- State capture: Mementos (Changes) store diffs without exposing internals
- Caretaker role: History object manages memento list for apply/revert
- Originator: Project/Grid creates and restores from mementos

Key Modifications:

- Ordered dependencies: Mementos must be applied in sequence due to state reliance
- Serialization: Mementos persisted to disk for project saving
- No reverse in v4: Immutable grids simplify restoration by rolling back references

Benefits:

- Robust undo/redo: Users can experiment safely with data transformations
- Reproducibility: Export history for scripted application
- Efficiency: Diff-based storage minimizes memory for large histories

5.Factory Pattern

Primary Classes:

- Runner interfaces - Create Grid instances
- ImportingManager.java - Registers and creates importers

The Factory pattern abstracts object creation, used for runners and data importers.
Comparison with Standard Factory Pattern.

Similarities:

- Encapsulates creation: Clients request objects without knowing concrete classes
- Polymorphism: Different factories (runners) produce compatible objects (Grids)
- Configurable: Selected via command-line or config files

Key Modifications:

- Pluggable factories: Runners adapt to environments (local, distributed, testing)
- Lazy instantiation: Factories produce proxy objects for deferred computation
- Test suite integration: Common base for verifying factory outputs

Benefits:

- Flexibility: Switch execution modes without code changes
- Testability: Dedicated testing factory for in-memory validation
- Resource optimization: Choose factories based on dataset size

Design Patterns Case Study : libGDX

Project Overview

Project: libGDX
Repository: https://github.com/libgdx/libgdx

Short Description of the Project

libGDX is an open-source, cross-platform Java game development framework based on OpenGL (ES), enabling the creation of 2D and 3D games for Windows, Linux, macOS, Android, iOS, and web browsers. It emphasizes flexibility, performance, and a rich ecosystem without enforcing specific coding styles.

Identified Patterns

Based on the project's architecture and source code structure, the following design patterns are used:

• Facade Pattern (Structural pattern)

• Strategy Pattern (Behavioral pattern)

• Adapter Pattern (Structural pattern)

• Observer Pattern (Behavioral pattern)

• Command Pattern (Behavioral pattern)

• Composite Pattern (Structural pattern)

• Factory Pattern (Creational pattern)

1. Facade Pattern

Primary Classes:

- Gdx.java - Central static access point to core subsystems

The Facade pattern simplifies access to libGDX's complex subsystems (e.g., application lifecycle, graphics, input, audio, files, and networking) by providing a unified, static interface.

Comparison with Standard Facade Pattern

Similarities:

- Unified interface: Hides the complexity of multiple subsystems behind a single class.
- Decouples clients: Game developers interact with subsystems without direct knowledge of their implementations.

Key Modifications:

- Static implementation: Uses static fields (e.g., Gdx.app, Gdx.graphics) for global access, rather than instance-based, to suit the single-application context of games.

Benefits:

- Simplified API: Reduces boilerplate for common operations, improving developer productivity.
- Portability: Abstracts platform-specific details, enabling cross-platform consistency.
- Maintainability: Changes to subsystems don't affect user code.

2.Strategy Pattern

Primary Classes:

- ApplicationListener.java - Defines the strategy interface for game logic
- Lwjgl3Application.java, AndroidApplication.java - Platform-specific classes that use the listener

The Strategy pattern allows interchangeable game logic algorithms via the ApplicationListener interface, which defines methods for the game lifecycle (create, render, dispose, etc.), plugged into platform-specific application classes.

Comparison with Standard Strategy Pattern

Similarities:

- Interchangeable algorithms: Different implementations of ApplicationListener can be swapped without altering the context (application classes).
- Polymorphism: The context invokes the strategy's methods uniformly.

Key Modifications:

● Lifecycle-focused: Strategies include specific hooks for resize, pause, and resume, tailored to game loops rather than generic operations.

Benefits:

- Flexibility: Enables custom game behaviors while reusing the framework's rendering and event handling.
- Cross-platform: The same strategy works across backends without modification.
- Extensibility: Easy to compose complex games by implementing the interface.

3.Adapter Pattern

Primary Classes:

- ApplicationAdapter.java - Provides default implementations for ApplicationListener

The Adapter pattern eases implementation of the ApplicationListener interface by offering an adapter class with empty methods, allowing developers to override only necessary lifecycle hooks.

Comparison with Standard Adapter Pattern

Similarities:

- Interface adaptation: Converts the full interface into a more convenient form by providing defaults.
- Reduces effort: Users avoid implementing unused methods.

Key Modifications:

- Blank defaults: All methods are no-ops, optimized for game development where not all lifecycle events are always needed.

Benefits:

- Convenience: Lowers the barrier for beginners by avoiding verbose boilerplate.
- Customization: Developers can selectively override methods like render() or dispose().
- Clean code: Prevents clutter from empty method implementations in user classes.

4.Observer Pattern

Primary Classes:

- InputProcessor.java - Interface for observing input events
- InputMultiplexer.java - Manages and notifies multiple processors

The Observer pattern handles input events (key presses, touches, etc.) by notifying registered InputProcessor observers, with multiplexing for chaining multiple handlers.

Comparison with Standard Observer Pattern

Similarities:

- Event notification: Subjects (input system) notify observers via methods like keyDown() or touchDown().
- Loose coupling: Observers register without tight integration to the subject.

Key Modifications:

- Event propagation: Multiplexer allows events to pass through multiple observers until handled (returning true stops propagation).
- Game-specific events: Focuses on real-time input like scrolling or flings.

Benefits:

- Reactive input: Enables modular event handling, e.g., separate processors for UI and gameplay.
- Composability: Easy to combine processors for complex interactions.
- Performance: Efficient for high-frequency events in games.

5.Command Pattern

Primary Classes:

- Action.java - Encapsulates executable actions
- Actor.java - Executes added actions

The Command pattern encapsulates behaviors (e.g., move, fade, rotate) as Action objects, which are added to actors and executed over time in the scene2d UI system.

Comparison with Standard Command Pattern

Similarities:

- Encapsulates requests: Each action object contains logic to perform an operation via act().
- Undo/redo potential: Actions can be restarted or reversed in some cases.

Key Modifications:

- Time-dependent: Includes duration, interpolation, and pooling for animations, extending beyond simple execution.
- Targeting: Actions are bound to specific actors.

Benefits:

- Reusable behaviors: Compose complex animations by sequencing actions.
- Decoupled logic: Separates action definition from execution context.
- Efficiency: Supports object pooling to minimize garbage collection.

6.Composite Pattern

Primary Classes:

- Group.java - Composite that contains actors
- Actor.java - Component base class

The Composite pattern builds hierarchical scene graphs in scene2d, where Group treats individual Actors and nested Groups uniformly for drawing, updating, and transformations.

Comparison with Standard Composite Pattern

Similarities:

- Tree structure: Composites (Groups) contain leaves (Actors) and other composites.
- Uniform treatment: Methods like draw() or act() recurse through children.

Key Modifications:

- Rendering-specific: Includes culling, transformations, and hit detection propagated to children.
- Event bubbling: Events like touches bubble up the hierarchy.

Benefits:

- Hierarchical organization: Simplifies managing complex UIs or scenes (e.g., menus with buttons).
- Scalability: Easy to build nested structures without special cases.
- Maintainability: Uniform API for single and grouped elements.

7.Factory Pattern

Primary Classes:

- AssetLoader.java - Abstract base for asset factories
- AssetManager.java - Registers and uses loaders to create assets

The Factory pattern creates assets (textures, sounds, etc.) via registered AssetLoader subclasses, allowing dynamic instantiation based on asset type.

Comparison with Standard Factory Pattern

Similarities:

- Abstracts creation: Clients request assets without knowing concrete classes.
- Polymorphism: Different loaders produce compatible objects.

Key Modifications:

- Parameterized: Loaders use AssetDescriptor and parameters for customization.
- Asynchronous: Supports threaded loading with dependencies.

Benefits:

- Extensibility: Add custom loaders for new asset types without core changes.
- Resource management: Handles loading, unloading, and references automatically.
- Performance: Optimizes for game loading times with progress tracking.

Design Patterns Case Study : Activiti

Project Overview

Project: Activiti
Repository: https://github.com/Activiti/Activiti

Short Description of the Project

Activiti is an open-source, lightweight workflow and Business Process Management (BPM) platform with a core BPMN 2.0 process engine for Java. It targets business users, developers, and system admins, integrates seamlessly with Spring, and runs in standalone, clustered, or cloud environments under the Apache license.

Identified Patterns

Based on the project's architecture and source code structure, the following design patterns are used:

• Facade Pattern (Structural pattern)

• Observer Pattern (Behavioral pattern)

• Factory Pattern (Creational pattern)

• Singleton Pattern (Creational pattern)

• Command Pattern (Behavioral pattern)

• State Pattern (Behavioral pattern)

• Chain of Responsibility Pattern (Behavioral pattern)

1.Facade Pattern

Primary Classes:

- ProcessEngine.java - Central facade for accessing all engine services

The Facade pattern provides a unified, simplified interface to the complex subsystems of the BPM engine, such as runtime execution, task management, and repository operations, hiding internal complexities from users.

Comparison with Standard Facade Pattern

Similarities:

- Single entry point: Abstracts multiple subsystems (services like RuntimeService, TaskService) behind one interface.
- Decouples clients: Users interact with the engine without needing to know service implementations or dependencies.

Key Modifications:

- Thread-safe and shareable: Designed for concurrent access in clustered environments, with caching for performance.
- Configurable: Built via ProcessEngineConfiguration, allowing environment-specific setups.

Benefits:

- Ease of use: Simplifies BPM interactions for embedding in applications.
- Maintainability: Changes to internal services don't affect external APIs.
- Scalability: Supports shared instances across nodes connecting to the same database.

2.Observer Pattern

Primary Classes:

- ActivitiEventListener.java - Interface for event observers
- ActivitiEventDispatcher.java - Manages and notifies listeners

The Observer pattern enables notification of engine events (e.g., process start, task creation, job execution) to registered listeners, allowing custom reactions like logging or KPI monitoring without altering core engine logic.

Comparison with Standard Observer Pattern

Similarities:

- Subject-observer decoupling: Events are dispatched to listeners without tight coupling.
- Event-driven: Listeners react to specific event types

Key Modifications:

- Typed events: Uses ActivitiEventType enum for fine-grained event classification, including custom types.
- Global and per-process: Listeners can be engine-wide or attached to specific BPMN elements (e.g., execution listeners on activities).

Benefits:

- Extensibility: Easy to add monitoring or integrations (e.g., auditing) via listeners.
- Flexibility: Supports runtime addition/removal of listeners.
- Performance: Efficient dispatching avoids blocking core execution.

3.Factory Pattern

Primary Classes:

●ProcessEngineConfiguration.java - Abstract factory for engine creation

The Factory pattern abstracts the creation of ProcessEngine instances, allowing configuration-based instantiation for different environments (e.g., in-memory for testing, Spring-integrated).

Comparison with Standard Factory Pattern

Similarities:

● Encapsulates creation: Clients request engines without specifying concrete classes.
● Polymorphism: Subclasses handle variations (e.g., JtaProcessEngineConfiguration for JTA transactions).

Key Modifications:

●Builder-stylechaining: Supports fluent configuration (e.g.,setDatabaseSchemaUpdate()) before building.
● Environment-aware: Integrates with Spring (via SpringProcessEngineConfiguration) or JNDI datasources.

Benefits:

● Configurability: Adapts to standalone, clustered, or testing setups.
● Testability: In-memory variants simplify unit tests.
● Reusability: Cached engines via ProcessEngines registry.

4.Singleton Pattern

Primary Classes:

● ProcessEngines.java - Manages and retrieves singleton engine instances

The Singleton pattern ensures a single, shared ProcessEngine instance per configuration (e.g., default engine), caching them in a registry for reuse across the application.

Comparison with Standard Singleton Pattern

Similarities:

● Restricted instantiation: getDefaultProcessEngine() enforces single access.
● Global access: Provides thread-safe, lazy initialization.

Key Modifications:

● Registry-based: Supports multiple named singletons (not just one global), retrievable by name.
● Destruction hook: Includes unregister() for cleanup in web apps or tests.

Benefits:

● Resource efficiency: Avoids redundant engine creation in multi-threaded or clustered setups.
● Consistency: Ensures all components use the same engine instance.
● Integration: Works with servlets for proper lifecycle management.

5.Command Pattern

Primary Classes:

- Command.java - Base for executable commands
- CommandExecutor.java - Executes commands in a context

The Command pattern encapsulates BPM operations (e.g., starting processes, completing tasks) as command objects, executed transactionally with interceptors for logging, authorization, and error handling.

Comparison with Standard Command Pattern

Similarities:

- Encapsulates requests: Commands contain all needed data and logic for execution.
- Decouples invoker: Clients submit commands without knowing execution details.

Key Modifications:

- Interceptor chain: Commands pass through a chain (CommandInterceptor) for aspects like transactions or auditing.
- Contextual: Uses CommandContext for shared state (e.g., session management).

Benefits:

- Transactional safety: Ensures atomicity in database operations.
- Extensibility: Easy to add interceptors for custom behavior.
- Modularity: Separates concerns in complex BPM executions.

6.State Pattern

Primary Classes:

- ProcessInstance.java and ExecutionEntity.java - Manage process states
- Behavior classes - Handle state-specific logic

The State pattern models the lifecycle of process instances and executions (e.g., active, suspended, ended), with behaviors changing based on BPMN elements.

Comparison with Standard State Pattern

Similarities:

- State encapsulation: Different states delegate to specific behaviors.
- Transitions: Events trigger state changes.

Key Modifications:

- BPMN-driven: States align with BPMN semantics (e.g., parallel gateways for concurrent executions).
- Hierarchical: Supports nested executions (sub-processes) with parent-child relationships.

Benefits:

- Flexibility: Handles complex flows like parallelism and compensation.
- Clarity: Separates state logic from core engine.
- Robustness: Ensures correct handling of wait states and continuations.

7.Chain of Responsibility Pattern

Primary Classes:

- CommandInterceptor.java - Base for interceptor chain
- SequenceFlow.java - Handles conditional flows

The Chain of Responsibility pattern processes requests (e.g., commands or sequence flows) through a series of handlers, such as interceptors for commands or conditional evaluations for BPMN flows.

Comparison with Standard Chain of Responsibility Pattern

Similarities:

- Sequential handling: Each handler decides to process or pass to the next.
- Decoupling: Handlers are independent, configurable in order.

Key Modifications:

- Configurable chain: Interceptors are linked dynamically (e.g., for transactions, logging).
- Conditional branching: In BPMN, used for evaluating sequence flow conditions until a match.

Benefits:

- Aspect-oriented: Adds cross-cutting concerns without core changes.
- Flow control: Manages decision points in processes efficiently.
- Maintainability: Easy to insert/remove handlers.