Design Document — Nostalgic Terminal Games

Design Patterns Used

Factory Pattern

Problem it Solves: Dynamically instantiates the correct game object (Tic Tac Toe, Hangman, or Minesweeper) based on user input.
Usage: GameFactory promotes scalability by adhering to the open-closed principle.

Strategy Pattern

Problem it Solves: Allows flexible and interchangeable logic for each game type.
Usage: Each game (e.g., TicTacToeStrategy, HangmanStrategy) implements a common interface, enabling isolated game logic.

Template Method Pattern

Problem it Solves: Defines a consistent game flow while allowing variations in specific steps.
Usage: AbstractGame manages lifecycle, concrete games implement specific logic.

Observer Pattern

Problem it Solves: Enables decoupled real-time updates for scoreboards and logs.
Usage: GameEventPublisher notifies components like ScoreBoard and Logger.

Singleton Pattern

Problem it Solves: Ensures only one instance exists for central resources like the Game Engine.
Usage: GameEngine and Logger use Singleton to maintain centralized control.

Command Pattern

Problem it Solves: Decouples user input parsing from game logic execution.
Usage: Actions like MakeMove, Restart, and Exit are encapsulated in command objects.

Decorator Pattern

Problem it Solves: Adds optional features (e.g., hints) to games without altering base classes.
Usage: HintDecorator enhances Game object behavior dynamically.

Adapter Pattern

Problem it Solves: Bridges incompatible interfaces (e.g., external game components).
Usage: Converts third-party module interfaces to system-compatible formats.

Composite Pattern

Problem it Solves: Treats individual UI elements and containers uniformly (e.g., menu structures).
Usage: MenuComponent and MenuItem allow for hierarchical menu systems.

Bridge Pattern
Problem it Solves: Decouples abstraction (UI) from implementation (Console, GUI).
Usage: Enables future UI changes without affecting game logic.

Builder Pattern
Problem it Solves: Constructs complex GameSession objects in a readable and flexible manner.
Usage: Used for setting up games with specific difficulty, hints, and configurations.

Prototype Pattern
Problem it Solves: Enables game state cloning for undo/replay functionality.
Usage: Game state snapshots can be created quickly without rebuilding from scratch.

State Pattern
Problem it Solves: Manages different states (e.g., Initializing, Playing, Paused, Ended) clearly.
Usage: GameContext delegates behavior to state-specific objects.

Flyweight Pattern
Problem it Solves: Reduces memory use by sharing immutable game components.
Usage: Cell objects in board games are reused to optimize memory.

Facade Pattern
Problem it Solves: Simplifies interaction with a complex game subsystem.
Usage: GameFacade offers a unified interface to GameEngine, CommandProcessor, Logger, etc.

System Architecture
This is a terminal-based standalone Java application with modular organization.

Major Components:
- Console UI: Interface layer that interacts with users.
- Game Engine (Singleton): Manages the full game lifecycle.
- Game Factory: Chooses the correct game instance.
- Game Strategies: Game rule logic encapsulated per type.
- Command Processor: Handles parsing and executing user inputs.
- Event System: Publishes and responds to in-game events.
- Optional In-Memory Persistence: Stores score/session data.

Component Interactions:
1. User ↔ Console UI: Users interact via terminal.
2. Console UI → Command Processor: Parses user commands.
3. Command Processor → Game Engine: Routes commands for execution.
4. Game Engine → Game Factory: Instantiates the correct game.
5. Game Engine ↔ Strategy Pattern: Game logic is executed per selected strategy.
6. Game Engine ↔ Event System: Notifies ScoreBoard, Logger on specific events.

Data Flow:

User initiates interaction → Console UI captures input → Command Processor routes to appropriate module → Game logic executed via Strategy + Template Method → Responses shown via Console UI → Events pushed via Observer.

Component/Service Breakdown

Game Engine

- Purpose: Controls the game session, transition through states.
- Design Pattern(s): Singleton, Template Method, State

Console UI

- Purpose: Takes user input, displays feedback.
- Design Pattern(s): Observer, Command, Bridge

Game Factory

- Purpose: Instantiates game objects based on user choice.
- Design Pattern(s): Factory

Game Strategies

- Purpose: Encapsulates logic and rules of each game.
- Design Pattern(s): Strategy

Command Processor

- Purpose: Parses commands and delegates actions.
- Design Pattern(s): Command

ScoreBoard

- Purpose: Tracks and displays scores.
- Design Pattern(s): Observer

Logger

Purpose: Logs game events.
- Design Pattern(s): Singleton, Observer

Hint System (Optional)

Purpose: Adds optional hinting functionality.
Design Pattern(s): Decorator

Game Session Manager (Optional)

Purpose: Builds and clones session configurations.
Design Pattern(s): Builder, Prototype

Menu System (Optional):
Purpose: Handles nested menus.
Design Pattern(s): Composite

External Integrations (Optional):
Purpose: Wraps legacy modules.
Design Pattern(s): Adapter

Technology Stack

Component	Technology Used
Language	Java
Build Tool	Maven or Gradle
Testing Framework	JUnit
Version Control	Git
Console I/O	Standard Java Streams
Logging	Java Logger / Log4j