Project: 15 Tiles Game
Subject: Object Oriented Programming
Prof: Salvatore Distefano
Candidate: Shehryar Raja
Matricula: 523115

Table of Contents:

1 Introduction

This project implements a backend system for the classic **15-Puzzle game** using Java, Spring Boot, and Object-Oriented Programming principles. The application exposes a REST API that lets clients create new games, view the board, make moves, and play variants with **move** and **time limits**, with each game tracked by its own unique ID.

The core design is based on a **Puzzle** abstraction and a **Board** that manages tile positions and rules. The code is structured into controllers, services, domain classes, DTOs, and exceptions, applying **encapsulation**, **inheritance**, **composition**, and **polymorphism** to keep the architecture clear, maintainable, and easy to extend.

2 Requirements

2.1 Functional Requirements

1. **Game creation**
   The system allows clients to start a new 15-Puzzle game. When a game is created, the board is shuffled and a unique **`gameId`** is returned to identify that session.

2. **View game state**
   For any active game, the client can request the current 4×4 board associated with its **`gameId`**. The empty space is represented internally and included in the returned layout.

3. **Perform moves**
   The client can request to move a specific tile. Only tiles adjacent to the empty space are accepted; invalid moves or inputs are rejected with an

error response.

4. **Check puzzle completion**
   At any moment, the client can verify whether a given game is solved. The system reports if the tiles are in the correct order or not.

5. **Support for multiple games**
   Several games can run at the same time. Each game state is stored separately using its own **`gameId`**, ensuring no interference between different sessions.

6. **API accessibility and tools**
   The backend is accessible from browser-based clients through enabled CORS. Swagger/OpenAPI documentation is available locally to inspect and test the API endpoints.

7. **Health check and basic UI**
   A health endpoint confirms that the service is running, and a minimal static UI is provided to interact with the puzzle through a web page.

2.2 Non-Functional Requirements

1. **Performance**
   All game operations (creating games, reading the board, making moves) are handled in memory, so responses are fast and suitable for real-time interaction.

2. **Scalability**
   The system can manage multiple games at the same time using a thread-safe game registry, without requiring a database in the current scope.

3. **Reliability and Correctness**
   The board always keeps a valid 15-Puzzle configuration (tiles 1–15 and

one empty space), and tests help verify game ID uniqueness and core game behavior.

4. **Maintainability**
   The codebase is organized into clear layers (controllers, services, domain classes, DTOs, exceptions), making the logic easy to understand, modify, and extend.

5. **Security and Robustness**
   Inputs such as game IDs and moves are validated, and error responses are controlled so that internal implementation details are not exposed.

6. **Concurrency Safety**
   A **`ConcurrentHashMap`** is used for storing active games, and scheduled timers are managed safely to avoid conflicts between parallel sessions.

7. **Portability**
   The application runs on Java 17+ with Maven and Spring Boot, and can be deployed on common operating systems like Windows, macOS, and Linux.

8. **Observability**
   Standard HTTP status codes and a health-check endpoint are provided to quickly verify that the service is running correctly.

3 Game Architecture

The 15-Puzzle backend follows a modular architecture where web endpoints, game management, and puzzle logic are clearly separated into packages. This structure keeps the code simple, easy to maintain, and ready for future extensions.

3.1 Controllers (`local.controllers`)

- **GameController** – Exposes REST endpoints to create a game, get the board, make moves, and check if the puzzle is solved. It validates input and delegates all logic to the services.

- **HomeController / UiForwardController** – Provide the health check and forward `/ui` requests to the static web interface.

3.2 Services (`local.services`)

- **GameManagerService** – Manages all active games. It creates new puzzles, assigns a unique `gameId`, stores them in a thread-safe map, and applies move/time limit rules.

- **GameLogicService** – Contains helper logic such as shuffling, checking valid moves, and verifying if the puzzle is solved.

3.3 Game Package (`local.game`)

- **Board** – Stores the 4×4 grid with tiles 1–15 and the empty space, ensuring a valid configuration.

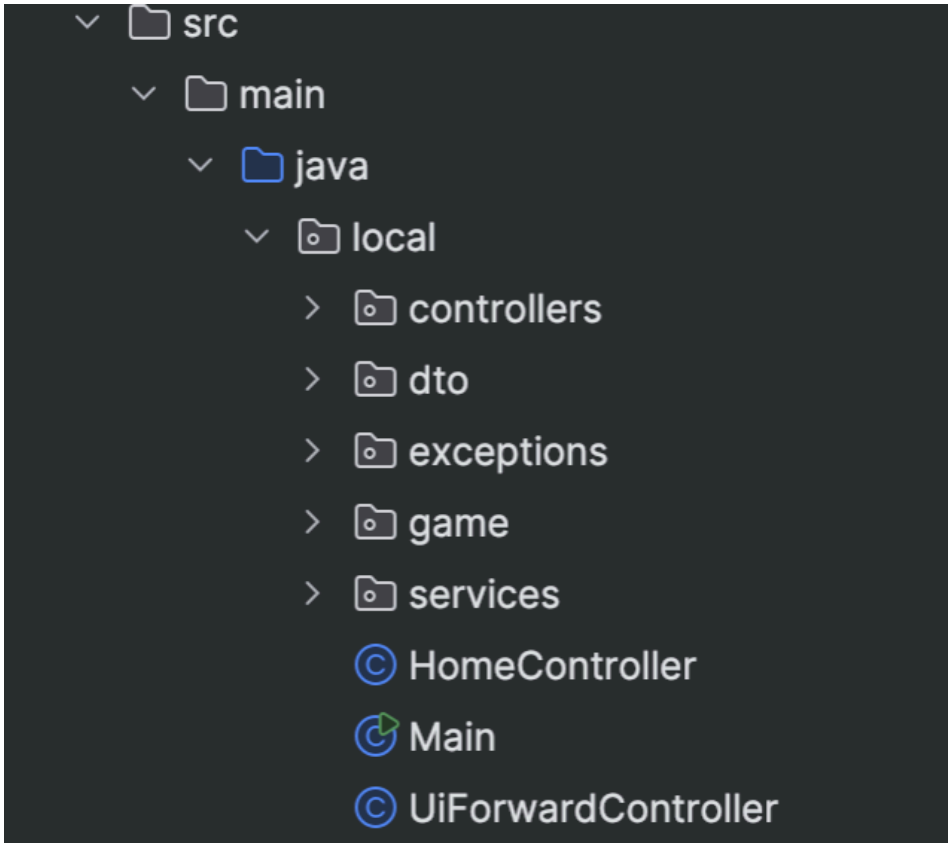
- **Puzzle (interface)** – Defines common operations (get state, move tile, check solved), allowing different puzzle variants to share the same contract.

- **MoveLimitedPuzzle / TimedPuzzle** – Implement the `Puzzle` interface by composing a `Board` and adding move or time limits without changing how clients use the puzzle.

- **Direction** – Enum representing move directions (UP, DOWN, LEFT, RIGHT).

3.4 Supporting Classes

- **MoveRequest (`local.dto`)** – Models the request body for a move operation.
- **NoSuchGameException (`local.exceptions`)** – Returned when an invalid `gameId` is used.
- **Main** – Spring Boot entry point that starts the application.



4 Classes and Relationships

This section describes the main classes that drive the 15-Puzzle logic and how they interact.

4.1 Puzzle (Interface)

Defines the common behavior of a puzzle:

- Get the current board.
- Make a move.
- Check if the puzzle is solved or lost.
- Access optional limits (moves, time).

This provides a single contract so different puzzle variants can be handled in the same way.

4.2 Board

Represents the 4×4 grid:

- Stores tiles **1–15** and one empty space.
- Ensures a valid and solvable configuration.
- Provides safe snapshots of the board for the API.
- Supports operations used to apply valid moves.

4.3 MoveLimitedPuzzle

Implements `Puzzle` and adds a move limit:

- Composes a `Board` (has-a relationship).
- Tracks how many valid moves have been made.
- Marks the game as lost when the move limit is reached.
- Keeps board details encapsulated.

4.4 TimedPuzzle

Extends `MoveLimitedPuzzle` and adds a time limit:

- Inherits all behavior from `MoveLimitedPuzzle`.
- Start timing when the game begins.
- Consider the game lost if the time limit expires or the move limit is exceeded.

4.5 Direction (Enum)

Defines movement directions:

- `UP`, `DOWN`, `LEFT`, `RIGHT`.
  Used to express moves clearly and consistently in the puzzle logic.

4.6 GameManagerService

Central manager for all games:

- Creates a new shuffled, solvable `Board`.
- Wraps it in a `TimedPuzzle` (with configured move and time limits).
- Stores games in a `Map<String, Puzzle>`.
- Provides methods to get the board, apply moves, and check solved/lost status.

4.7 GameController

Connects HTTP clients to the puzzle logic:

- Exposes REST endpoints for creating games, making moves, and reading state.

- Delegates all game operations to `GameManagerService`.

- Returns JSON responses with appropriate HTTP status codes.

4.8 Supporting Classes

- **MoveRequest (DTO)**
  Represents the request body for a move, keeping the controller input simple and separated from domain logic.

- **NoSuchGameException**
  Thrown when an invalid `gameId` is used and mapped to a clear error response, without exposing internal details.

4.9 Summary of Relationships

- **Inheritance**
    - `TimedPuzzle` extends `MoveLimitedPuzzle`.
    - Both implement the `Puzzle` interface.
- **Composition**
    - `MoveLimitedPuzzle` and `TimedPuzzle` contain a `Board`.
    - `The board` uses `Direction` for movement.
- **Aggregation**

    - `GameManagerService` manages multiple `Puzzle` instances in a `Map<gameId, Puzzle>`.

- **Layering**

    - Domain logic (`Board`, `Puzzle`, `MoveLimitedPuzzle`, `TimedPuzzle`, `Direction`) is separated from services (`GameManagerService`) and web layer (`GameController`, `MoveRequest`).

5 Object-Oriented Principles Applied

5.1 Encapsulation:

Encapsulation means putting both state (data) and behavior (methods) inside a class, and letting other parts of the program interact with that class **only through its methods**. It groups related logic in one place so that the object itself is responsible for updating its internal state.

For example, instead of letting other classes change fields directly, we:

- Keep related fields and methods together in the same class.

- Expose operations like **`moveTile(...)`**, **`isSolved()`**, or “create game” so that the class can update its own data correctly.

In this project, encapsulation is applied as follows:

The board keeps all board-related state and logic together.
It knows how to represent the 4×4 grid, how to slide tiles, how to shuffle, and how to check if the puzzle is solved. Other classes do not move tiles manually; they call methods like **`slideTile(...)`**, **`slide(Direction)`**, or **`snapshot()`** and let the board update its own state.

MoveLimitedPuzzle and **TimedPuzzle** encapsulate the rules of each game mode.
They store the current board and the move/time limits and expose operations such as **`moveTile()`**, **`isSolved()`**, **`hasLost()`**, **`movesMade()`**, and **`maxMoves()`**. The rest of the system does not modify counters or timers directly; it just asks the puzzle object to perform moves and to report its status.

Direction encapsulates the valid movement options in an enum (**`UP`**, **`DOWN`**, **`LEFT`**, **`RIGHT`**), so the code uses meaningful directions instead of arbitrary integers or strings.

GameManagerService encapsulates the game lifecycle.
Controllers do not work with maps or timers directly; they call higher-level methods like “create game”, “make move”, and “get game state”, and the service decides how to manage game objects internally.

Encapsulation in the Board class

In this implementation, the board is a self-contained unit that knows everything about the 4×4 grid. It stores the tiles, creates a solved configuration through the **`solved()`** factory method, and exposes operations such as **`tileAt(...)`** and **`snapshot()`** so that other classes can read the current state without moving tiles themselves. All board-related logic (initialization, access, and updates) stays inside the **`Board`** class, and other parts of the program interact with it only through these methods.

```java
public final class Board {  9 usages
    private static final int N = 4;  21 usages
    private final int[][] tiles = new int[N][N];  10 usages

    private Board() {  1 usage
        int k = 1;
        for (int i = 0; i < N; i++) {
            for (int j = 0; j < N; j++) {
                tiles[i][j] = (i == N - 1 && j == N - 1) ? 0 : k++;
            }
        }
    }

    public static Board solved() { return new Board(); }  1 usage
    public int size() { return N; }  no usages
    public int tileAt(int row, int col) {  no usages
        checkBounds(row, col);
        return tiles[row][col];
    }

    public int[][] snapshot() {  1 usage
        int[][] c = new int[N][N];
        for (int i = 0; i < N; i++) System.arraycopy(tiles[i],
                srcPos: 0, c[i],  destPos: 0, N);
        return c;
    }
```

5.2 Information Hiding

Information hiding means protecting a class's internal data by declaring attributes as private. This ensures that other classes cannot directly access or modify the data. Instead, access must go through public methods (getters/setters), which lets us add validation or logic if needed.

In this project, information hiding is applied as follows:

- **The board** keeps the tile grid and helper methods private. Other classes cannot access the `tiles` array directly; they only interact through methods like `solved()`, `tileAt(...)` and `snapshot()`, which returns a safe copy.

- **MoveLimitedPuzzle** and **TimedPuzzle** hide their internal counters, limits, and timing data. External code can only use methods such as `moveTile(...)`, `isSolved()`, or `hasLost()`, without knowing how the rules are implemented inside.

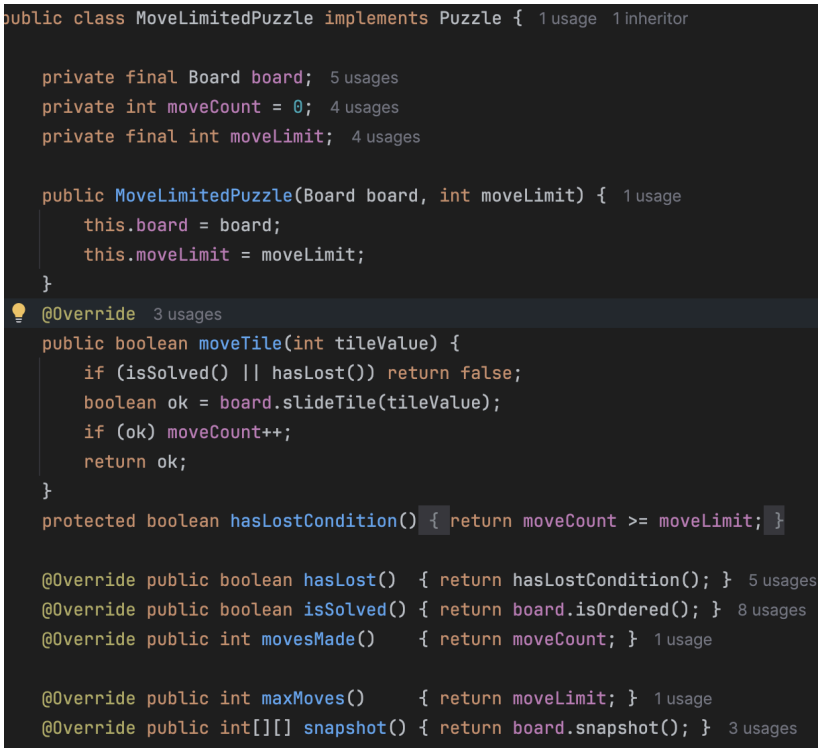
- **GameManagerService** keeps the game registry (`Map<String, Puzzle>`) private and exposes only high-level operations like create game, make move, and get game state.

- **Direction** provides named values (`UP`, `DOWN`, `LEFT`, `RIGHT`) instead of raw numbers, hiding how movement is calculated internally.

- **GameController** interacts only with the public APIs and DTOs, never with internal fields of the puzzle or board.

Information hiding in `MoveLimitedPuzzle`

In **`MoveLimitedPuzzle`**, the internal state of the game mode is hidden using private fields: the `Board` instance, the **`moveCount`**, and the **`moveLimit`**. These values cannot be changed directly from outside the class. Instead, all updates go through the **`moveTile(int tileValue)`** method, which first checks the game status and only increments the counter when a valid move is performed. Other methods like **`hasLost()`**, **`isSolved()`**, **`movesMade()`**, **`maxMoves()`**, and **`snapshot()`** expose only the necessary information, without revealing or allowing direct access to the underlying data.



5.3 Reuse

Reuse refers to the use of existing code components such as method or object, in different parts of a program without rewriting them, promoting efficiency and consistency.

In this project, reuse is achieved by:

- **Puzzle interface** – **`MoveLimitedPuzzle`** and **`TimedPuzzle`** share the same contract (**`moveTile`**, **`isSolved`**, **`hasLost`**, **`snapshot`**, etc.), so **`GameManagerService`** can work with any puzzle variant in a uniform way.

- **Inheritance** – **`TimedPuzzle`** reuses all logic from **`MoveLimitedPuzzle`** and only extends the losing rule with a time limit.

- **Board class** – Both puzzle variants delegate moves and win checks to the same **`Board`** methods, avoiding duplicated move logic.

- **Shuffle and snapshot** – The same **`shuffleByRandomMoves`** and **`snapshot()`** implementations are reused for every game, ensuring solvable boards and safe board views.

- **Shared types** – **`Direction`** and **`MoveRequest`** are reused across controllers, services, and domain classes to express moves consistently.

In this class, code reuse is achieved through inheritance. **`TimedPuzzle`** extends **`MoveLimitedPuzzle`**, so it automatically reuses all the existing move and board logic. The constructor calls **`super(board, moveLimit)`** to reuse the parent initialization, and `hasLostCondition()` calls **`super.hasLostCondition()`** and only adds the extra time check. The class introduces just **`timeLimit`**, **`startedAt`**, and simple getters, showing how new behavior is added without duplicating the original puzzle implementation.

```java
public class TimedPuzzle extends MoveLimitedPuzzle {  2 usages

    private final Duration timeLimit;  3 usages
    private final Instant startedAt;  2 usages

    public TimedPuzzle(Board board, int moveLimit, Duration timeLimit) {  1 usage
        super(board, moveLimit);
        this.timeLimit = timeLimit;
        this.startedAt = Instant.now();
    }

    private Duration elapsed() { return Duration.between(startedAt, Instant.now()); }

    @Override  2 usages
    protected boolean hasLostCondition() {
        boolean moveLimitHit = super.hasLostCondition();
        boolean timeOver = elapsed().compareTo(timeLimit) >= 0;
        return moveLimitHit || timeOver;
    }

    public Duration getTimeLimit() { return timeLimit; }  no usages
    public Duration getElapsed()   { return elapsed(); }  no usages
}
```

5.4 Inheritance

Inheritance allows a class to reuse fields and methods from another class, keeping shared logic in one place and letting subclasses only add or modify what changes.

In this project, inheritance is applied between **`MoveLimitedPuzzle`** and **`TimedPuzzle`**:

- **`MoveLimitedPuzzle`** implements **`Puzzle`** and defines the common behavior for the 15-Puzzle game: it holds the **`Board`**, counts moves, enforces the move limit, and provides methods like **`moveTile(...)`**, **`isSolved()`**, **`hasLost()`**, **`movesMade()`**, **`maxMoves()`**, and **`snapshot()`**. It also defines a protected **`hasLostCondition()`** method used to decide when the player loses.

- **`TimedPuzzle`** **extends** **`MoveLimitedPuzzle`**, inheriting all this logic without rewriting it. It adds only two new fields (**`timeLimit`** and **`startedAt`**) and overrides **`hasLostCondition()`** to include the time constraint together with the move limit.

Inheritance in `TimedPuzzle`

This class shows inheritance clearly. **`TimedPuzzle`** extends **`MoveLimitedPuzzle`**, so it automatically inherits all the puzzle behavior: board handling, move counting, and loss checking. In the constructor, it calls **`super(board, moveLimit)`** to reuse the parent initialization. It then adds only **`timeLimit`** and **`startedAt`**, and overrides **`hasLostCondition()`** to combine the original move-limit rule with an extra time-limit check. This demonstrates how new behavior is built on top of the base class without duplicating code.

```java
public class TimedPuzzle extends MoveLimitedPuzzle {  2 usages

    private final Duration timeLimit;  3 usages
    private final Instant startedAt;  2 usages

    public TimedPuzzle(Board board, int moveLimit, Duration timeLimit) {  1 usage
        super(board, moveLimit);
        this.timeLimit = timeLimit;
        this.startedAt = Instant.now();
    }

    private Duration elapsed() { return Duration.between(startedAt, Instant.now()); }

    @Override  2 usages
    protected boolean hasLostCondition() {
        boolean moveLimitHit = super.hasLostCondition();
        boolean timeOver = elapsed().compareTo(timeLimit) >= 0;
        return moveLimitHit || timeOver;
    }

    public Duration getTimeLimit() { return timeLimit; }  no usages
    public Duration getElapsed()   { return elapsed(); }  no usages
}
```

5.5 Composition

Composition is when a class is built by **having** other objects inside it (a “has-a” relationship) and using them to provide behavior, instead of inheriting from them.

In this project, composition appears clearly in:

- **MoveLimitedPuzzle has a Board**
  `MoveLimitedPuzzle` stores a **`Board`** as a private field. All moves and win checks are delegated to this **`Board`**, but the board itself is never exposed directly. This means the puzzle uses the board’s functionality without inheriting from it.

- **TimedPuzzle has time-related data**
  `TimedPuzzle` adds **`timeLimit`** and **`startedAt`** as its own fields and uses them together with the inherited logic. The time handling is composed into the puzzle behavior.

- **GameManagerService has many Puzzle instances**
  It keeps a **`Map<String, Puzzle>`** and controls how games are created, stored, and accessed. The service “has” puzzles and coordinates them.

These “has-a” relationships show composition: classes reuse behavior by containing other objects and delegating work to them.

Composition in `GameManagerService`

This code shows composition because **`GameManagerService`** is built by **having** other objects as its internal parts. It keeps a **`Map<String, Puzzle>`** to store active games, uses a **`ScheduledExecutorService`** plus **`timeouts`** and **`timedOut`** sets to manage time limits, and creates **`Board`** and **`TimedPuzzle`** instances inside **`createGame()`**. All these components are private and controlled through the service’s methods, so the class implements its behavior by composing and coordinating these objects instead of inheriting from them.

```
public class GameManagerService {

    private static final int MOVE_LIMIT = 250;  1 usage
    private static final Duration TIME_LIMIT = Duration.ofMinutes(3);  2 usages

    private final Map<String, Puzzle> games = new ConcurrentHashMap<>();  7 usages

    private final Random rnd = new Random();  1 usage

    private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 2
    private final Map<String, ScheduledFuture<?>> timeouts = new ConcurrentHashMap<>();  3 usages
    private final Set<String> timedOut = ConcurrentHashMap.newKeySet();  6 usages

    public String createGame() {  1 usage
        String id = UUID.randomUUID().toString();

        Board board = Board.solved();
        board.shuffleByRandomMoves( moves: 80, rnd);

        Puzzle game = new TimedPuzzle(board, MOVE_LIMIT, TIME_LIMIT);
        games.put(id, game);

        long seconds = TIME_LIMIT.toSeconds();
        ScheduledFuture<?> f = scheduler.schedule(() -> timedOut.add(id),
                seconds, TimeUnit.SECONDS);
        timeouts.put(id, f);

        return id;
    }
```

5.6 Subtyping

Subtyping is a relation between **types** (both classes and interfaces) where an object of a subtype can be used wherever a supertype is expected.
It enables polymorphism, because the client code can work with a general type while the concrete subtype decides the actual behaviour at runtime.

In this project:

- **`MoveLimitedPuzzle`** implements **`Puzzle`**, so it is a **subtype of the interface `Puzzle`**.

- **`TimedPuzzle`** extends **`MoveLimitedPuzzle`**, so it is a subtype of **`MoveLimitedPuzzle`, `Puzzle`, and `Object`**.

Subtyping in the client code (`GameManagerService`)

The important part is how the **client program** uses these types.
In **`GameManagerService`**, a single runtime object (**`TimedPuzzle`**) is accessed through different types of variables, including the **`Puzzle`** interface.

The map declaration:

```
private final Map<String, Puzzle> games = new ConcurrentHashMap<>();  7 usages
```

The full `createGame()` method:

```
public String createGame() {  1 usage
    String id = UUID.randomUUID().toString();

    Board board = Board.solved();
    board.shuffleByRandomMoves( moves: 80, rnd);

    TimedPuzzle timed = new TimedPuzzle(board, MOVE_LIMIT, TIME_LIMIT);
    Puzzle game = timed;
    games.put(id, game);

    long seconds = TIME_LIMIT.toSeconds();
    ScheduledFuture<?> f = scheduler.schedule(() -> timedOut.add(id),
            seconds, TimeUnit.SECONDS);
    timeouts.put(id, f);
    return id;
```

The **`GameManagerService`** declares **`games`** as **`Map<String, Puzzle>`**, creates a **`TimedPuzzle`** object, then assigns it to a **`Puzzle`** variable and stores it in the map. The same object is later accessed through the **`Puzzle`** interface, showing how one multi-type object is used through different variable types (class and interface).

5.7 Abstraction

Abstraction is the process of hiding complex implementation details and showing only the essential features of an object.

In this project, abstraction is mainly provided by the **`Puzzle` interface**:

- `Puzzle` defines what a puzzle game can do: **`moveTile(...)`**, **`isSolved()`**, **`hasLost()`**, **`movesMade()`**, **`maxMoves()`**, and **`snapshot()`**.
- It does **not** specify how the board is stored, how moves are validated, or how limits are enforced.

Classes like **`MoveLimitedPuzzle`** and **`TimedPuzzle`** implement this interface and contain the real logic, while **`GameManagerService`** and **`GameController`** work only with the **`Puzzle`** type. This keeps the rest of the code independent from implementation details and makes it easy to add new puzzle variants without changing the API.

Abstraction in `Puzzle`

The **`Puzzle`** interface defines the essential operations of the game—such as moving tiles, checking win/lose state, and reading a board snapshot—without exposing how the board is stored or how moves are implemented. Classes like **`MoveLimitedPuzzle`** and **`TimedPuzzle`** provide the internal logic, while controllers and services depend only on this abstract contract.

```
public interface Puzzle {  14 usages  2 implementations
    boolean moveTile(int tileValue);  3 usages  1 imple
    boolean isSolved();  8 usages  1 implementation
    boolean hasLost();  5 usages  1 implementation
    int movesMade();  1 usage  1 implementation
    int maxMoves();  1 usage  1 implementation
    int[][] snapshot();  3 usages  1 implementation
```

5.8 Polymorphism

Polymorphism refers to the ability of different classes to be treated as instances of a common superclass, and for methods to behave differently based on the object's actual class or parameters.
There are several forms of polymorphism in this project:

Runtime Polymorphism (Inclusion)

In this project, runtime polymorphism is achieved using the **`Puzzle`** interface and its implementations.

`MoveLimitedPuzzle` and **`TimedPuzzle`** both implement **`Puzzle`**, so they can be referenced through the common **`Puzzle`** type. In **`GameManagerService`**, each game is stored as a **`Puzzle`**:

- A new game is created as a **`TimedPuzzle`**, but saved in **`Map<String, Puzzle>`**.
- Later, methods like **`moveTile(...)`**, **`isSolved()`**, and **`hasLost()`** are called on the **`Puzzle`** reference.

At runtime, Java automatically dispatches these calls to the correct implementation based on the actual object (for example, **`TimedPuzzle`** applies both move limit and time limit). This means the service code never needs **`if`**/**`else`** on the class type; it just trusts the **`Puzzle`** interface, and different behaviors are provided by different implementations. That is runtime polymorphism by inclusion.

1. **Runtime polymorphism in `GameManagerService`**

In **`createGame()`**, a **`TimedPuzzle`** object is created but stored in the **`game's`** map using the general **`Puzzle`** type. Later, in **`makeMove(...)`**, the game is retrieved as a **`Puzzle`** and methods like **`hasLost()`**, **`isSolved()`**, and `moveTile(...)` are called on it. The actual implementation that runs depends on the real object (e.g., **`TimedPuzzle`**), showing runtime polymorphism: different puzzle implementations are used through the same **`Puzzle`** interface.

```java
private final Map<String, Puzzle> games = new ConcurrentHashMap<>();  7 usages
private final Random rnd = new Random();  1 usage

private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool(
private final Map<String, ScheduledFuture<?>> timeouts = new ConcurrentHashMap<>();
private final Set<String> timedOut = ConcurrentHashMap.newKeySet();  6 usages

public String createGame() {  1 usage
    String id = UUID.randomUUID().toString();

    Board board = Board.solved();
    board.shuffleByRandomMoves( moves: 80, rnd);

    Puzzle game = new TimedPuzzle(board, MOVE_LIMIT, TIME_LIMIT);
    games.put(id, game);

    long seconds = TIME_LIMIT.toSeconds();
    ScheduledFuture<?> f = scheduler.schedule(() -> timedOut.add(id),
            seconds, TimeUnit.SECONDS);
    timeouts.put(id, f);
    return id;
}
```

```java
public void makeMove(String gameId, int tileValue) {  1 usage
    Puzzle g = requireGame(gameId);
    if (timedOut.contains(gameId) || g.hasLost() || g.isSolved()) return;
    g.moveTile(tileValue);
    if (g.isSolved()) cancelTimeout(gameId);
}
```

2. **Ad-hoc Polymorphism (Method Overloading)**

This is a form of polymorphism where multiple methods have the same name but different parameter lists. The correct method is selected at compile time based on the arguments passed.

In this project, it is demonstrated by the **`moveTile`** operations:

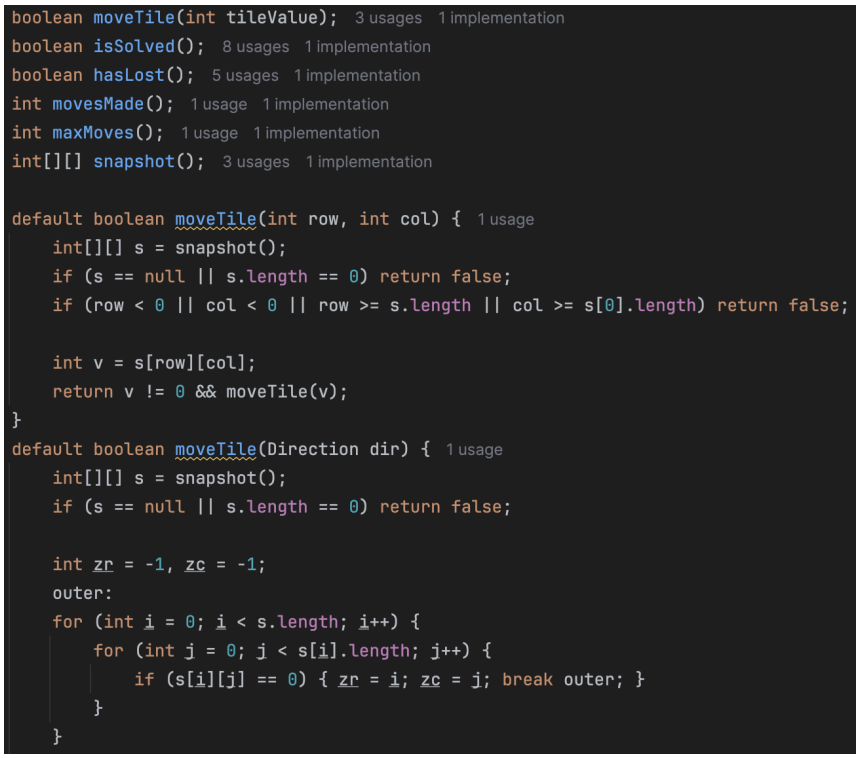
- In the **`Puzzle`** interface and its implementations, **`moveTile`** is overloaded to support:
    - **`moveTile(int tileValue)`** – move by tile number

- **`moveTile(int row, int col)`** – move by board coordinates
- **`moveTile(Direction dir)`** – move using a direction enum

All three represent “make a move”, but accept different inputs. When `GameManagerService` or the controller calls one of these, Java chooses the correct version at **compile time** based on the parameters, giving a flexible API without changing the method name.

Ad-hoc polymorphism in `Puzzle`

The **`Puzzle`** interface overloads **`moveTile`** to accept a tile value, board coordinates, or a direction. All three represent the same operation (“make a move”), and the compiler selects the correct version based on the arguments, demonstrating method overloading.



```
boolean moveTile(int tileValue);  3 usages  1 implementation
boolean isSolved();  8 usages  1 implementation
boolean hasLost();  5 usages  1 implementation
int movesMade();  1 usage  1 implementation
int maxMoves();  1 usage  1 implementation
int[][] snapshot();  3 usages  1 implementation

default boolean moveTile(int row, int col) {  1 usage
    int[][] s = snapshot();
    if (s == null || s.length == 0) return false;
    if (row < 0 || col < 0 || row >= s.length || col >= s[0].length) return false;

    int v = s[row][col];
    return v != 0 && moveTile(v);
}
default boolean moveTile(Direction dir) {  1 usage
    int[][] s = snapshot();
    if (s == null || s.length == 0) return false;

    int zr = -1, zc = -1;
    outer:
    for (int i = 0; i < s.length; i++) {
        for (int j = 0; j < s[i].length; j++) {
            if (s[i][j] == 0) { zr = i; zc = j; break outer; }
        }
    }
```

3. **Universal Polymorphism (Parametric / Generics)**

Parametric polymorphism may operate on any type, or on several types, which is specified by the user at runtime. It is often implemented in Java using Generics. Parametric polymorphisms are useful as they enable the programmer to write code with no dependencies on types. Such code will work with any data type, provided the type satisfies certain conditions.

In this project, universal (parametric) polymorphism mainly shows up through Java generics.

I used **`Map<String, Puzzle>`** to store games, so the manager can keep any **`Puzzle`** implementation keyed by a `String` without casting. For timeouts, I use **`Map<String, ScheduledFuture<?>>`** and **`Set<String> timedOut`**, which rely on generics to manage different value types safely. Classes like **`ConcurrentHashMap`** and **`ScheduledExecutorService`** are also used with type parameters so the types are checked at compile time.

I don't define my own generic classes here, but by using these generic collections and utilities, the code benefits from universal polymorphism: one generic implementation (like **`Map<K, V>`**) can work with many type combinations in a type-safe way.

Generics in `GameManagerService` – Generic types like **`Map<String, Puzzle>`**, **`Map<String, ScheduledFuture<?>>`** and **`Set<String>`** are used to manage different kinds of data with a single implementation, demonstrating parametric polymorphism in a type-safe way.

```
private final Map<String, Puzzle> games = new ConcurrentHashMap<>();  7 usages
private final Random rnd = new Random();  1 usage

private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 2
private final Map<String, ScheduledFuture<?>> timeouts = new ConcurrentHashMap<>();  3 usages
private final Set<String> timedOut = ConcurrentHashMap.newKeySet();  6 usages
```

Coercion Polymorphism:
Coercion polymorphism is when Java automatically converts one type to another to match a method's parameter.
Coercion polymorphism (not significantly used)
In Java, coercion polymorphism occurs when the language automatically converts one numeric type to another (for example, from **`int`** to **`double`**) to match a method parameter. In this 15-Puzzle backend, the logic does not rely on such implicit numeric conversions, so this form of polymorphism is only relevant in theory rather than being a key design feature.

5.9 Multithreading

A thread is a unit of execution within a program that performs a specific task. Threads allow a program to perform multiple tasks concurrently — a valuable feature for applications that require real-time interaction, such as games or servers.

In this Project:

- Spring handles **multiple HTTP requests** in parallel, so different players can play their games at the same time.
- **`GameManagerService`** uses:
    - **`ConcurrentHashMap`** and **`ConcurrentHashMap.newKeySet()`** to safely store games and timeouts across threads.
    - a **`ScheduledExecutorService`** to run **timeout tasks in the background** without blocking requests.

When a game is created, a timer task is scheduled on a separate thread. While that timer counts down, users can still make moves through normal request threads. If the timer fires, it marks the game as timed out using the shared, thread-safe data structures.

Multithreading in `GameManagerService`

A **`ScheduledExecutorService`** with its own thread pool schedules timeout tasks, which run after the configured time and mark games as timed out in the **`timedOut`** set. These background threads share state through thread-safe collections (**`ConcurrentHashMap`** / **`newKeySet()`**), allowing timeouts to run in parallel with normal HTTP requests without data races.

```
private final ScheduledExecutorService scheduler = Executors.newScheduledThreadPool( corePoolSize: 2
private final Map<String, ScheduledFuture<?>> timeouts = new ConcurrentHashMap<>();  3 usages
private final Set<String> timedOut = ConcurrentHashMap.newKeySet();  6 usages
```

```
long seconds = TIME_LIMIT.toSeconds();
ScheduledFuture<?> f = scheduler.schedule(() -> timedOut.add(id),
        seconds, TimeUnit.SECONDS);
timeouts.put(id, f);
return id;
```

5.10 Exception Handling

Java involves identifying and handling errors that occur during program execution. This allows the program to manage unexpected situations gracefully, rather than crashing with other words it allows the program to recover from errors and continue running, making it more robust and fault tolerant.

In this project:

- When a client uses an invalid **`gameId`**, the service throws an exception instead of continuing with bad data.

- The controller (or global handler) turns these exceptions into proper HTTP responses (e.g. 400/404) with a clear message.

- This keeps the API predictable and the server stable.

  Exception Handling in `makeMove`

  The **`makeMove`** endpoint wraps the service call in a **`try`** block. If the move is valid, it returns the updated board with **`200 OK`**. If **`GameManagerService`** throws an **`IllegalArgumentException`** (for example, invalid **`gameId`** or illegal move), it is caught and mapped to **`400 BAD_REQUEST`**. Any other unexpected error is caught by the second **`catch`** and returned as **`500 INTERNAL_SERVER_ERROR`**. This prevents crashes and ensures clear, consistent HTTP responses for error cases.

```java
@PutMapping(path = "/{gameId}/move", consumes = "application/json")  no usages
public ResponseEntity<int[][]> makeMove(
        @PathVariable String gameId,
        @RequestBody MoveRequest moveRequest) {
    try {
        gameManagerService.makeMove(gameId, moveRequest.getTileValue());
        int[][] updatedGameState = gameManagerService.getGameState(gameId);
        if (updatedGameState == null) {
            return new ResponseEntity<>(HttpStatus.NOT_FOUND);
        }
        return new ResponseEntity<>(updatedGameState, HttpStatus.OK);
    } catch (IllegalArgumentException ex) {

        return new ResponseEntity<>(HttpStatus.BAD_REQUEST);
    } catch (Exception e) {

        return new ResponseEntity<>(HttpStatus.INTERNAL_SERVER_ERROR);
    }
}
```

5.11 Extensibility

Extensibility is the ability of a software system to support future growth with minimal changes to the existing codebase. It allows new features, behaviors, or components to be added without rewriting or breaking existing logic. In this project, extensibility is achieved through careful use of abstraction, polymorphism, and modular design.

In this project, extensibility is mainly achieved through abstraction, polymorphism, and composition:

Programming to the `Puzzle` interface

`GameManagerService` and **`GameController`** work with the **`Puzzle`** interface instead of concrete classes. This means new puzzle variants (e.g. different rules, scoring, or modes) can be introduced by creating new implementations of **`Puzzle`** and plugging them in, without changing the calling code.

Extension via `hasLostCondition()`

`MoveLimitedPuzzle` defines a protected **`hasLostCondition()`** method, used by **`hasLost()`**. **`TimedPuzzle`** overrides this hook to add a time limit on top of the move limit. New loss rules can be added in the same way, by subclassing **`MoveLimitedPuzzle`** and overriding **`hasLostCondition()`**, while keeping the existing move logic intact.

Composition with `Board`

All puzzle variants reuse the same **`Board`** class for tile storage and move validation. New features can extend puzzle behavior (limits, scoring, timing, etc.) without modifying how the board works, because the board logic is encapsulated and shared through composition.

The class header:

```
public class MoveLimitedPuzzle implements Puzzle {
```

The board fields:

```
private final Board board;  5 usages
private int moveCount = 0;  4 usages
private final int moveLimit;  4 usages
```

The hook and its use:

```
protected boolean hasLostCondition() { return moveCount >= moveLimit; }
```

```
@Override public boolean hasLost()  { return hasLostCondition(); }
```

`MoveLimitedPuzzle` as the base puzzle class.

`MoveLimitedPuzzle` implements **`Puzzle`**, keeps the **`Board`**, **`moveCount`**, and **`moveLimit`** as private fields, and defines the protected **`hasLostCondition()`** method, which is used by **`hasLost()`**. Subclasses such as **`TimedPuzzle`** can override **`hasLostCondition()`** to add extra losing rules without changing this base class.

```
public class TimedPuzzle extends MoveLimitedPuzzle {  3 usages

    private final Duration timeLimit;  3 usages
    private final Instant startedAt;  2 usages

    public TimedPuzzle(Board board, int moveLimit, Duration timeLimit) {
        super(board, moveLimit);
        this.timeLimit = timeLimit;
        this.startedAt = Instant.now();
    }
```

```java
@Override  2 usages
protected boolean hasLostCondition() {
    boolean moveLimitHit = super.hasLostCondition();
    boolean timeOver = elapsed().compareTo(timeLimit) >= 0;
    return moveLimitHit || timeOver;
}
```

Extending the base puzzle with a time limit.

`TimedPuzzle` extends **`MoveLimitedPuzzle`**, adds the time-related fields (**`timeLimit`**, **`startedAt`**), and overrides **`hasLostCondition()`** to combine the move limit from the superclass with a time limit. This shows how new behaviour can be added without modifying the base class.

Conclusion:

This 15-Puzzle project provides a simple but solid backend that properly applies core Object-Oriented Programming concepts in a real Spring Boot application. Responsibilities are clearly split: controllers handle HTTP requests, services manage game lifecycle and rules orchestration, and the domain layer (**`Board`**, **`Puzzle`**, **`MoveLimitedPuzzle`**, **`TimedPuzzle`**) represents the puzzle state and behavior.

Encapsulation, inheritance, composition, abstraction, and polymorphism work together to make the system modular and extensible. The **`Puzzle`** interface offers a stable contract, while concrete variants reuse the same **`Board`** and extend only what changes (move limits, time limits). Generics and thread-safe collections allow multiple games and timeouts to run safely in parallel, and exception handling in the controller turns errors (like invalid **`gameId`** or moves) into clear HTTP responses instead of crashes.

Although it uses in-memory storage and a fixed 4×4 board, the architecture is ready for future enhancements such as different board sizes, scoring, persistence, authentication, or a richer UI without major redesign. Overall, the project is playable, robust, and neatly demonstrates practical OOP design in a clean, maintainable way.