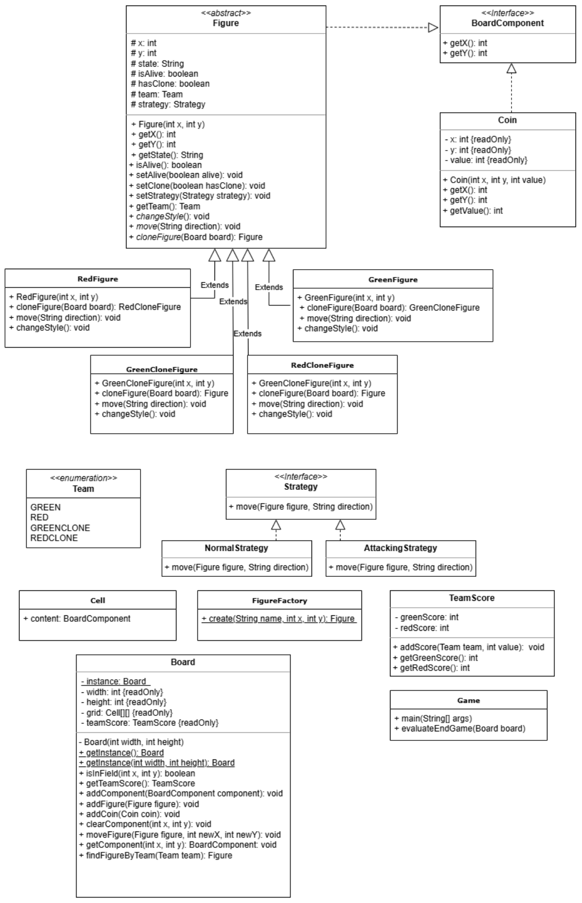
<<abstract>>
Figure
# x: int
# y: int
# state: String
# isAlive: boolean
# hasClone: boolean
# team: Team
# strategy: Strategy
+ Figure(int x, int y)
+ getX(): int
+ getY(): int
+ getState(): String
+ isAlive(): boolean
+ setAlive(boolean alive): void
+ setClone(boolean hasClone): void
+ setStrategy(Strategy strategy): void
+ getTeam(): Team
+ changeStyle(): void
+ move(String direction): void
+ cloneFigure(Board board): Figure
<<Interface>>
BoardComponent
+ getX(): int
+ getY(): int
Coin
- x: int {readOnly}
- y: int {readOnly}
- value: int {readOnly}
+ Coin(int x, int y, int value)
+ getX(): int
+ getY(): int
+ getValue(): int
RedFigure
+ RedFigure(int x, int y)
+ cloneFigure(Board board): RedCloneFigure
+ move(String direction): void
+ changeStyle(): void
Extends
Extends
Extends
Extends
GreenFigure
+ GreenFigure(int x, int y)
+ cloneFigure(Board board): GreenCloneFigure
+ move(String direction): void
+ changeStyle(): void
GreenCloneFigure
+ GreenCloneFigure(int x, int y)
+ cloneFigure(Board board): Figure
+ move(String direction): void
+ changeStyle(): void
RedCloneFigure
+ GreenCloneFigure(int x, int y)
+ cloneFigure(Board board): Figure
+ move(String direction): void
+ changeStyle(): void
<<enumeration>>
Team
GREEN
RED
GREENCLONE
REDCLONE
<<Interface>>
Strategy
+ move(Figure figure, String direction)
NormalStrategy
+ move(Figure figure, String direction)
AttackingStrategy
+ move(Figure figure, String direction)
Cell
+ content: BoardComponent
FigureFactory
+ create(String name, int x, int y): Figure
TeamScore
- greenScore: int
- redScore: int
+ addScore(Team team, int value): void
+ getGreenScore(): int
+ getRedScore(): int
Game
+ main(String[] args)
+ evaluateEndGame(Board board)
Board
- instance: Board
- width: int {readOnly}
- height: int {readOnly}
- grid: Cell[][] {readOnly}
- teamScore: TeamScore {readOnly}
- Board(int width, int height)
+ getInstance(): Board
+ getInstance(int width, int height): Board
+ isInField(int x, int y): boolean
+ getTeamScore(): TeamScore
+ addComponent(BoardComponent component): void
+ addFigure(Figure figure): void
+ addCoin(Coin coin): void
+ clearComponent(int x, int y): void
+ moveFigure(Figure figure, int newX, int newY): void
+ getComponent(int x, int y): BoardComponent: void
+ findFigureByTeam(Team team): Figure

Figure 1: UML diagram of my code

In my project I used 4 patterns: 2 creational patterns - Singleton and Factory Method, 1 structural - Composite, and 1 behavioral - Strategy.

Creational patterns

1. The Singleton pattern is used as a single instance: a board should inherently be unique within a single running game. It is illogical to have multiple independent boards. It also eliminates the possibility of accidentally creating multiple boards, which could lead to unpredictable game behavior and complicate state management. The Singleton pattern ensures that the Board class will have only one instance, which is accessible globally via the static getInstance() method.

2. Factory Method reates different types of figures (GreenFigure, RedFigure, GreenCloneFigure, RedCloneFigure) which has similar logic, but leads to the creation of objects of different classes. The factory method encapsulates this process in one place (FigureFactory), hiding from the client the details of creating each specific figure type. In addition, if in the future it is necessary to add new figure types, it will be enough to add a new case branch to the create() method of the factory and create the corresponding figure class. The code using the factory will remain unchanged.

Structural pattern

3. Using the Strategy pattern, the behavior of figures when moving can change depending on their state (NORMAL or ATTACKING). The pattern allows you to define movement strategies and make them interchangeable during execution. This allows figures to change their movement strategy without changing their class because the logic of moving figures is separated from the Figure class itself and encapsulated in separate strategy classes. It is implemented through the Strategy interface,

the NormalStrategy and AttackingStrategy strategy classes.

Behavioral pattern

4. The game board (Board) contains various objects (Figure and Coin). The Composite pattern allows working with both individual objects and compositions of objects in a uniform manner. The Board class can uniformly manage all elements on the board via the BoardComponent interface, without worrying about their specific type when performing common operations (placing on the board, getting an element by coordinates). Adding new types of elements to the board only requires implementing the BoardComponent interface, and the Board will be able to work with them without changing the core logic of element management. It is implemented via the BoardComponent interface, the "leaf" classes Figure and Coin, and the "composite" class Board.