Statement of completion

The project implements Numerical Tic-Tac-Toe inside an extensible board game framework. Below is a summary of what has and has not been completed.

Completed:

- **Extensible Framework**: Interfaces such as IBoard, IGameRules, IGameHelp, IGameCommand, and IPiece are in place. These make the design reusable so that other games could be plugged in.
- **Numerical Tic-Tac-Toe game**: Implemented as NumericalTicTacToeGame, with its own rules, help, pieces, NumericalComputerPlayer, and command factory.
- **Rules**: Encapsulated in NumericalTicTacToeRules. This checks valid moves, detects wins, and detects draws.
- **Players**: Human vs Human and Human vs Computer are supported. The computer can automatically generate random moves.
- **Undo/Redo**: Moves can be undone and redone, supported through commands and tracked in the HistoryManager. User input ('undo', 'redo').
- **Save/Load**: Implemented through GameStorage. User input ('save ', 'load ').
- **Help**: Implemented in NumericalTicTacToeHelp which implements the IGameHelp interface, lists all available gameplay commands. User input ('help').
- **Rules Display**: Overview of Numerical Tic-Tac-Toe Rules displayed via ShowWelcome() and accessed via user input ('rules').

Not completed / Limitations:

- **Integer-based representation:** The framework uses integers to represent board state and pieces. This was a deliberate choice for simplicity, but it also works for other games that can map pieces to codes (e.g. Connect Four: 0 = empty, 1 = P1 disc, 2 = P2 disc). As a result of this design:
  - Board: stores integers in its cells.
  - Pieces: expose only a numeric value.
  - Rules: validate moves and detect wins using integers.
  - Save/Load: serialises values using integers.
- **Game catalogue:** Only NTTT implemented, other menu entries are placeholders.

All other basic requirements of the assignment have been met.

Summary of Libraries and classes used:

- **System**: Console, Environment, Random
- **System.Collections.Generic**: List, Stack
- **System.IO**: StreamWriter, StreamReader
- **System.Linq**: Enumerable
- **System.Text**: StringBuilder

Design Overview

The final design for the extensible two-player board game framework, demonstrated with Numerical Tic-Tac-Toe, builds on the preliminary Assignment 1 design with significant refinements to improve reusability, maintainability, and extensibility. The preliminary design outlined 10 core classes and 1 interface, focusing on basic responsibilities like rules, state management and move tracking. The final implementation expands this to 21 classes and interfaces, introducing abstractions and design patterns to address assignment requirements, implementation challenges, and the need for polymorphic behaviour to support additional games. These changes were motivated by the feedback received from Assignment 1, which emphasised extracting further commonalities while reducing coupling.

1) Structural Refinements and Interface Additions

Changes: Core components were formalised behind interfaces**:** IBoard, IPiece, IGameRules, IGameHelp, IGameCommand, plus an AbstractCommandFactory for command creation.

- The planned "Board" became GridBoard implementing IBoard
- "Help" became NumericalTicTacToeHelp via IGameHelp
- The preliminary "Move" class was replaced by commands (PlaceNumberCommand implementing IGameCommand).

Why: The preliminary design relied on concrete classes and implicit move handling, which tightly coupled logic. Interfaces promote polymorphism and loose coupling, so other games can plug in alternative boards, rules, help, and commands without changes to the engine.

2) Class Refinements and Additions

Changes:

- History renamed to HistoryManager with stacks for commands.
- Refactored the standalone "Move" concept into IGameCommand implementations.
- Introduced NumericalTicTacToeGame as a BoardGame
- Player base gained AvailablePieces list
- Added NumericalComputerPlayer as a subclass of ComputerPlayer

Why:

- HistoryManager reflects its invoker role
- Refactoring "Move" into commands enabled consistent redo/undo via Execute/Undo
- NumericalTicTacToeGame & NumericalComputerPlayer for game-specific behaviour
- Adding AvailablePieces to Player base decouples resource management from engine

3) Incorporation of Design Patterns

Changes: Please refer to the Design Principles & Patterns section (discussed extensively).
Why: The Preliminary design lacked patterns, limiting flexibility. The integration of multiple patterns reduced coupling and enabled extensibility.

- Template & Factory allowed games to define their specifics without altering engine
- Command enabled undo/redo and consistent move encapsulation
- Strategy (IGameRules already existed) but subclassing ComputerPlayer now leaves room for alternative strategies (note NumericalCP currently only inherits random)
- Abstract Factory keeps BoardGame independent of concrete command types

Design Principles & Patterns

Single Responsibility Principle (SRP)

The design follows SRP by giving each class a single clear role. Examples include:

- GridBoard: stores grid and displays it to console (no knowledge of rules or players).
- NumericalTicTacToeRules: defines game rules (doesn't handle storage or I/O).
- NumericalTicTacToeHelp: prints available commands (contains no game logic).

Open/Closed Principle (OCP)

The framework is closed for modification but open for extension. The abstract BoardGame class defines the game loop and depends primarily on interfaces, making it easy to plug in new game-specific classes. To create a new game, we do not change the engine but provide classes such as rules, help, pieces, a command factory, a board and optionally a computer player. For example, different computer player strategies can be introduced by subclassing ComputerPlayer, as shown with NumericalComputerPlayer.

Liskov Substitution Principle (LSP)

The framework follows LSP because subclasses can be used wherever their base type is expected without breaking the game. For example, HumanPlayer and ComputerPlayer are both valid Player objects, and NumericalComputerPlayer can substitute for ComputerPlayer. Similarly, any IGameRules implementation can be dropped in as long as it respects the contract.

Interface Segregation Principle (ISP)

ISP states that classes should not be forced to depend on methods they do not use. The framework follows ISP by using small, focused interfaces. Examples include:

- IGameRules: only defines methods related to game rules (valid moves, win/draw checks)
- IGameHelp: only defines a single method to display help text
- IBoard: only defines board operations (get/set cell, display)

None of these interfaces force implementing classes to carry unused methods.

Dependency Inversion Principle (DIP)

The framework follows DIP because the BoardGame engine depends primarily on abstractions (IGameRules, IGameHelp, AbstractCommandFactory, IBoard) rather than concrete classes, which allows different games to plug in their own implementations, without changing the engine. For example, NumericalTicTacToeGame supplies NumericalTicTacToeRules for IGameRules etc. HistoryManager and GameStorage are used directly as concretes, but these could also be abstracted (e.g. IHistoryManager, IStorage) if multiple implementations (IStorage for local and cloud) were needed later.

Don’t Repeat Yourself (DRY)

The design avoids duplication by centralising responsibilities in single classes.

Keep It Simple, Stupid (KISS)

The framework emphasises simplicity by representing the board and pieces as integers. This makes tasks such as win checks, move validation and save/load straightforward. Although this limits extensibility for more complex games, it keeps the implementation easy to understand and avoids unnecessary complexity for the assignment requirements.

YAGNI

The implementation focuses only on what was required for the assignment. For example, GameStorage was added to the BoardGame constructor so that save and load functionality is available without hardcoding storage directly into the engine. This also makes it possible for different storage types in the future (e.g., cloud storage), but only required local save/load feature was implemented.

Template Method

BoardGame defines the overall game loop (Play( )) and the fixed steps (DisplayBoard, CheckWin, IsDraw, ProcessHumanInput and ComputerPlayerTurn). It leaves hooks for game-specific behaviour: ShowWelcome, CreatePiece, GetStartingOrderOptions, CreateComputerPlayer, and InitialisePieces. NumericalTicTacToeGame overrides these to supply Numerical Tic-Tac-Toe specifics.

Factory Method

Object creation is deferred to overridable methods in BoardGame:

- Pieces via CreatePiece(int) (abstract)
- ComputerPlayer via CreateComputerPlayer(int) (virtual)

NumericalTicTacToeGame provides NumericalTicTacToePiece & NumericalComputerPlayer

Strategy

- Rules strategy: The engine uses the IGameRules interface for validity, move generation, and win/draw checks; NumericalTicTacToeRules supplies one concrete behaviour, but any other rules class could be swapped in to change the logic while keeping the same algorithm steps. (See Diagram).
- ComputerPlayer strategy: ComputerPlayer selects a move; the numerical variant subclasses it (NumericalComputerPlayer) and can override GenerateMove(…) to supply a different strategy.

Command

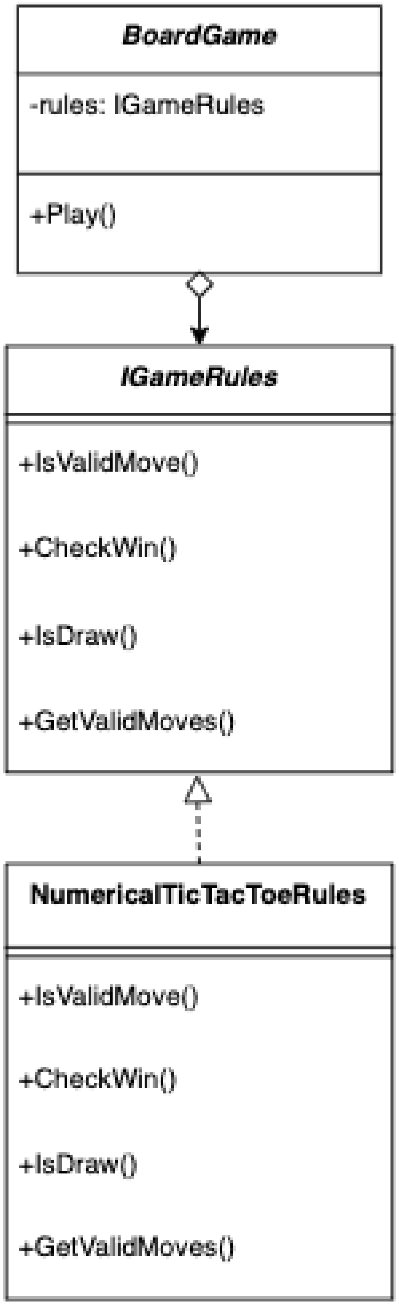
Moves are encapsulated as commands. IGameCommand defines Execute/Undo; PlaceNumberCommand writes/undoes the cell value and updates piece pools; HistoryManager (the invoker) executes and tracks undo/redo. The engine creates a command and executes it during MakeMove.

Abstract Factory

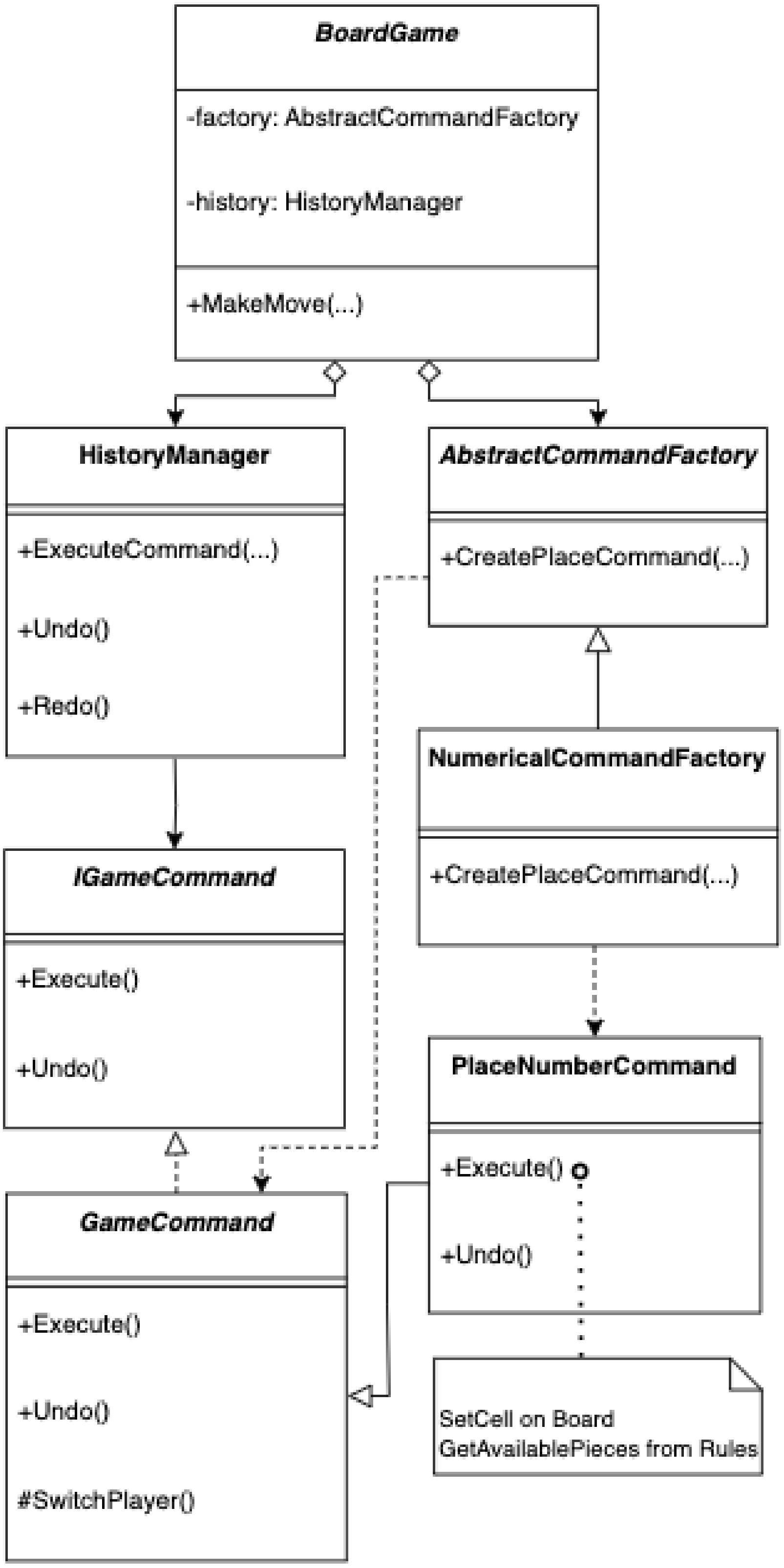
BoardGame (the engine) delegates command creation to an abstract factory. In MakeMove, the factory provides an IGameCommand, which is then executed by the HistoryManager. The abstract method is implemented by NumericalCommandFactory, returning a PlaceNumberCommand, keeping BoardGame independent of concrete command types. This promotes extensibility for new games by subclassing the factory and command.

Diagrams:

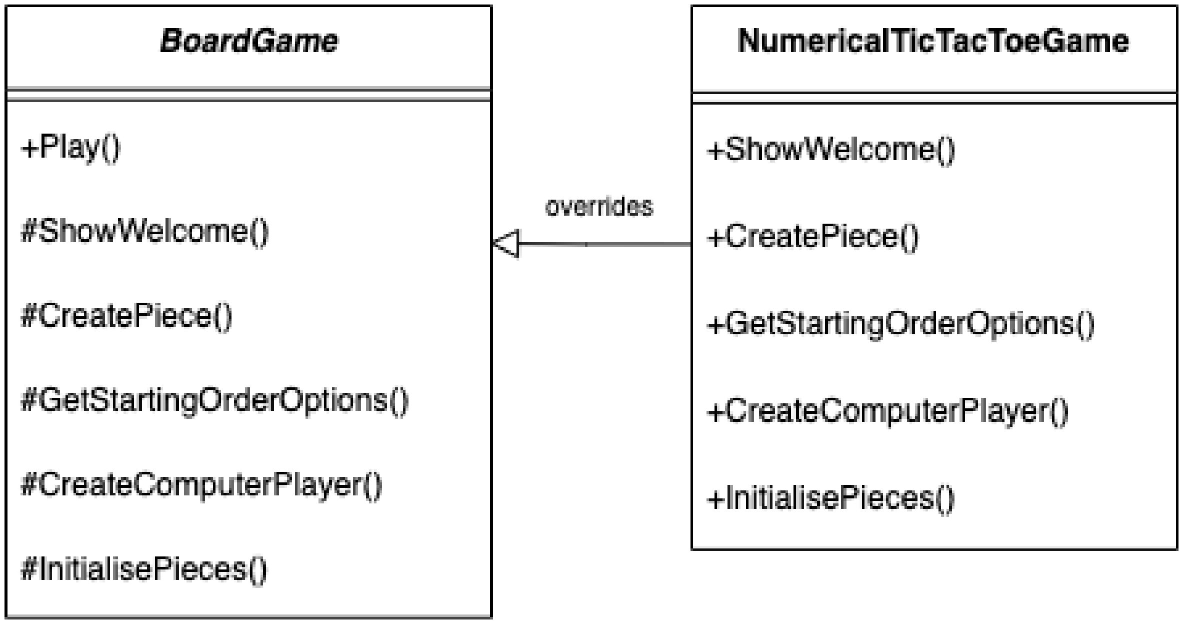
Strategy Pattern:
Game Rules



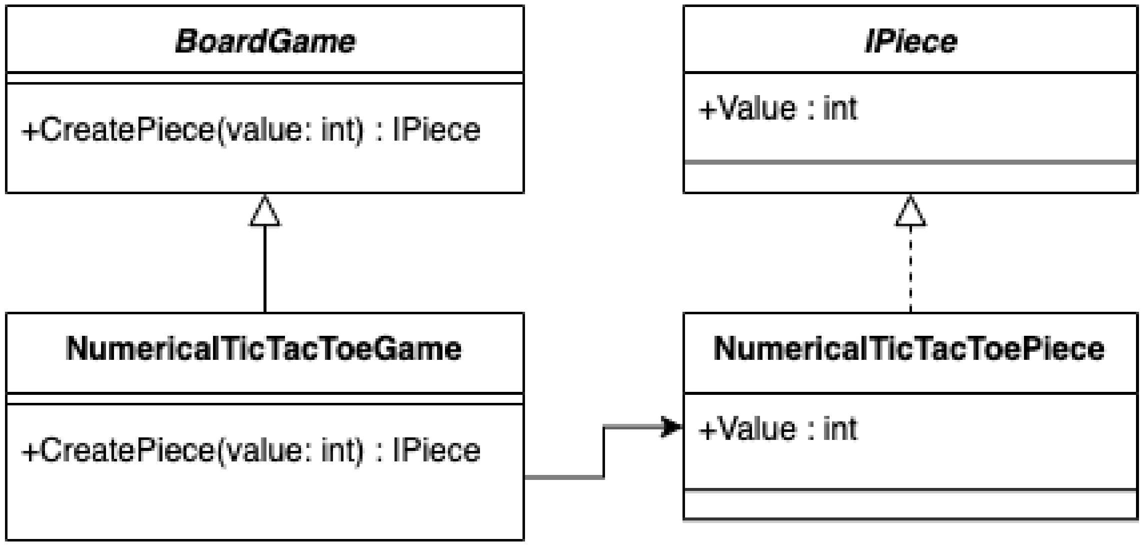
Abstract Factory + Command Pattern:
Command Creation and Execution



Template Method Pattern



Factory Method Pattern



Execution Instructions

Prerequisites:

- .NET 6 SDK installed.
- UTF-8 support (for board display).

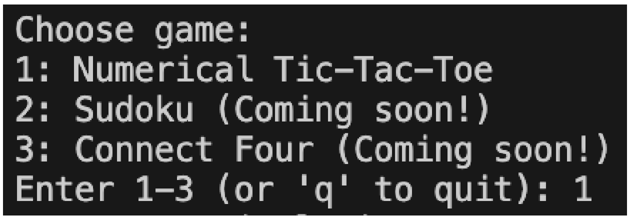
Getting Started:

1. Open a terminal.
2. Navigate to the project folder that contains BoardGames.csproj.
3. Input to terminal
   a. dotnet build
   b. dotnet run

Choose game:

1. Numerical Tic-Tac-Toe
2. Sudoku (n/a)
3. Connect Four (n/a)

Enter 1-3 (or ‘q’ to quit):
Enter 1 to play Numerical Tic-Tac-Toe or ‘q’ to quit. Invalid choices re-prompt



```
Choose game:
1: Numerical Tic-Tac-Toe
2: Sudoku (Coming soon!)
3: Connect Four (Coming soon!)
Enter 1-3 (or 'q' to quit): 1
```

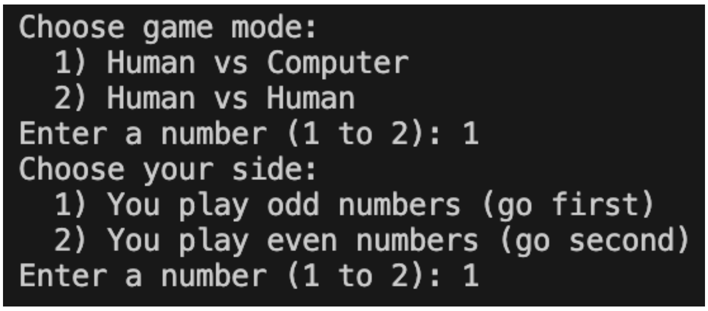
Read Rules:

- The game will show a brief welcome/rules summary (typing rules shows this again).

Choose mode:

1. Human vs Computer
2. Human vs Human

Enter 1 or 2. If HvC (1) is selected, the user will also choose who starts.



```
Choose game mode:
  1) Human vs Computer
  2) Human vs Human
Enter a number (1 to 2): 1
Choose your side:
  1) You play odd numbers (go first)
  2) You play even numbers (go second)
Enter a number (1 to 2): 1
```

Making a Move:

Enter: place <row> <column> <number>
rows/columns are 1-based
Example: place 1 1 3 (places 3 top left cell)

Game Commands:

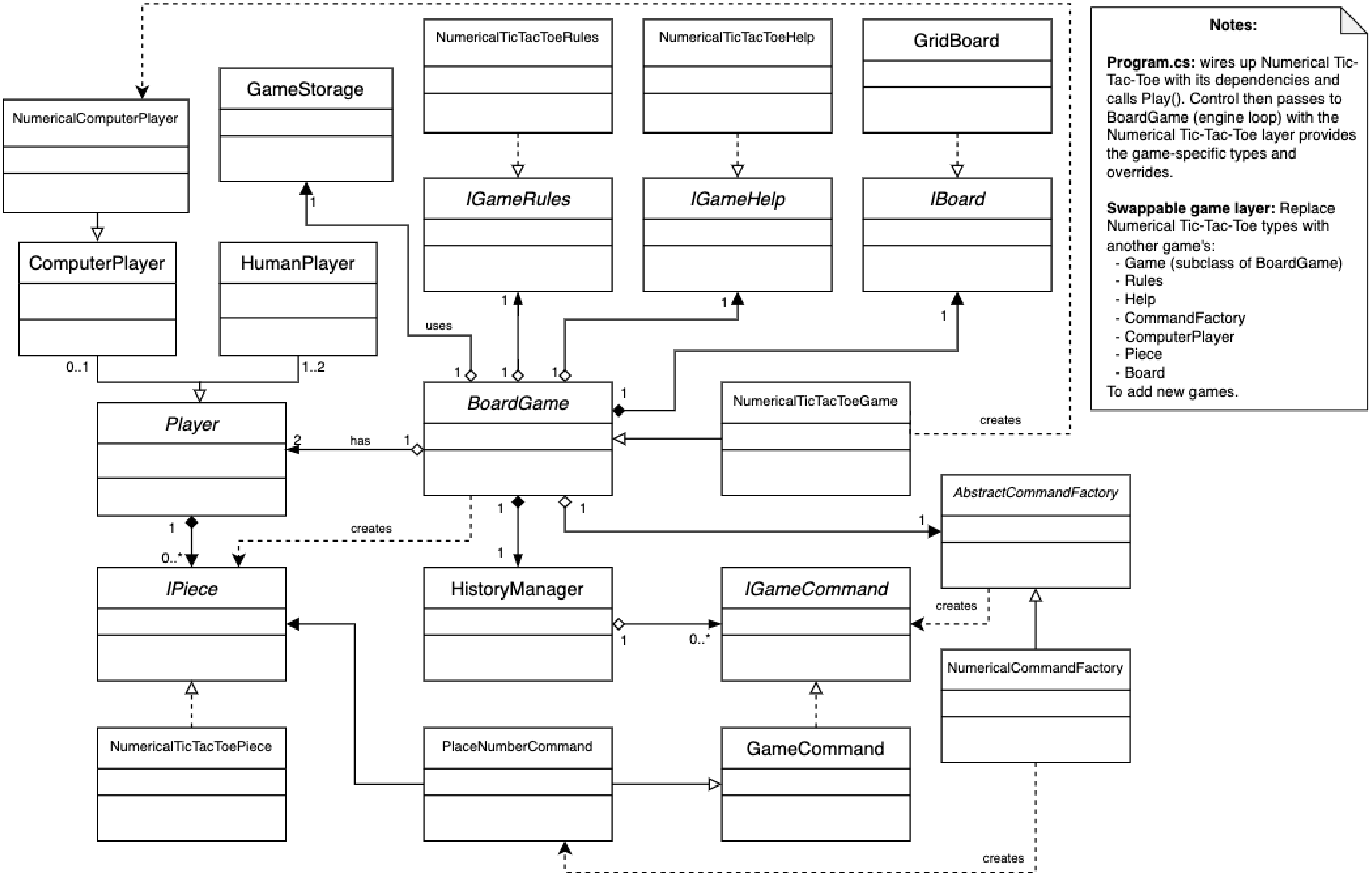
Below is a list of extra commands (available to view by inputting ‘help’)

- Undo last move: undo
- Redo move: redo
- Reroll computer move (HvC): reroll
- Show help: help
- Show rules: rules
- Save game: save <name>
- Load game: load <name>
- Quit: quit (not q unlike menu to safeguard unintended exit)

Notes:

- Player 1 uses odds (1,3,5,7,9). Player 2 uses evens (2,4,6,8).
- Any line of three numbers (excludes empty (0)) summing to 15 wins.
- The board updates display after each place/undo/redo/reroll.
- Files are saved/loaded in the current working directory.
- Program returns to Game Menu after game ends (press 1 to play again).

Class Diagram – Numerical Tic-Tac-Toe



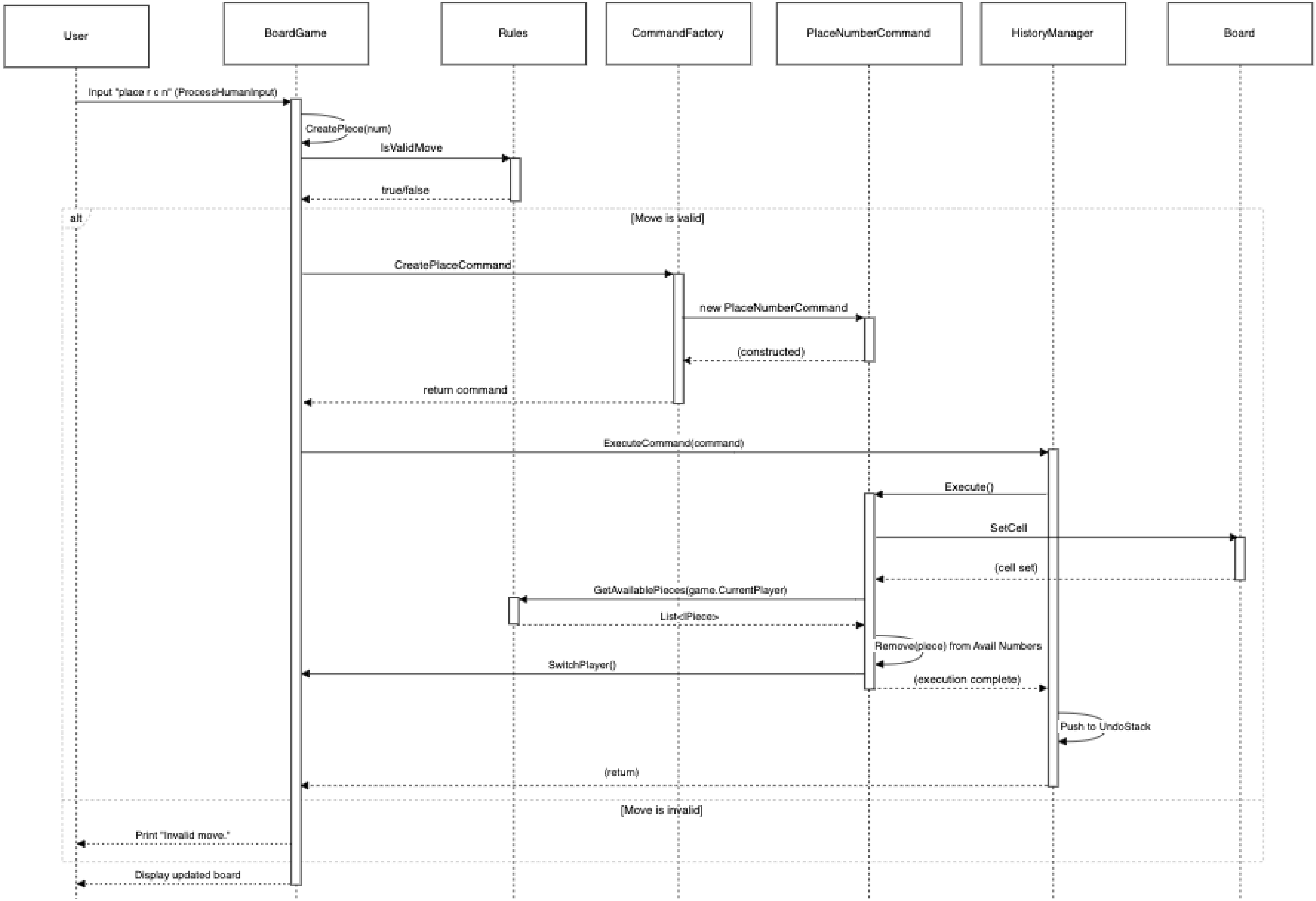
Notes:

Program.cs: wires up Numerical Tic-Tac-Toe with its dependencies and calls Play(). Control then passes to BoardGame (engine loop) with the Numerical Tic-Tac-Toe layer provides the game-specific types and overrides.

Swappable game layer: Replace Numerical Tic-Tac-Toe types with another game's:
- Game (subclass of BoardGame)
- Rules
- Help
- CommandFactory
- ComputerPlayer
- Piece
- Board

To add new games.

Sequence Diagram – Numerical Tic-Tac-Toe (Human player makes a move)



User

BoardGame

Rules

CommandFactory

PlaceNumberCommand

HistoryManager

Board

Input "place r c n" (ProcessHumanInput)

CreatePiece(num)

IsValidMove

true/false

alt

[Move is valid]

CreatePlaceCommand

new PlaceNumberCommand

(constructed)

return command

ExecuteCommand(command)

Execute()

SetCell

(cell set)

GetAvailablePieces(game.CurrentPlayer)

List<IPiece>

Remove(piece) from Avail Numbers

SwitchPlayer()

(execution complete)

Push to UndoStack

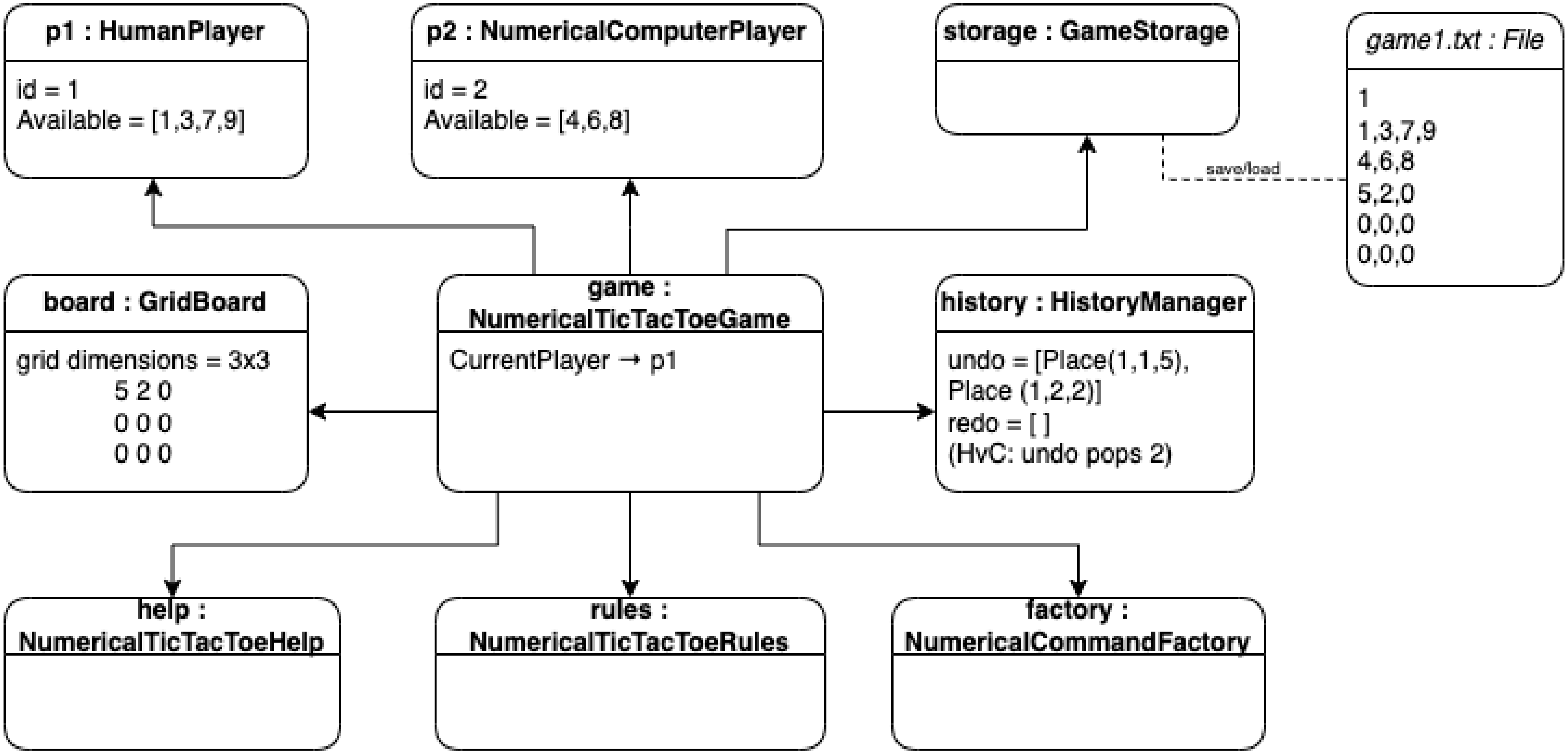
(return)

[Move is invalid]

Print "Invalid move."

Display updated board

Object Diagram – Numerical Tic-Tac-Toe (Human vs Computer, two moves played)



p1 : HumanPlayer
id = 1
Available = [1,3,7,9]

p2 : NumericalComputerPlayer
id = 2
Available = [4,6,8]

storage : GameStorage

game1.txt : File
1
1,3,7,9
4,6,8
5,2,0
0,0,0
0,0,0

board : GridBoard
grid dimensions = 3x3
5 2 0
0 0 0
0 0 0

game : NumericalTicTacToeGame
CurrentPlayer → p1

history : HistoryManager
undo = [Place(1,1,5), Place (1,2,2)]
redo = [ ]
(HvC: undo pops 2)

help : NumericalTicTacToeHelp

rules : NumericalTicTacToeRules

factory : NumericalCommandFactory

Snapshot after two moves (HvC, human first). NumericalTicTacToeGame composes the engine parts (board, rules, help, factory, history, storage) and aggregates two players. The board shows the live grid state (5 at [1,1], 2 at [1,2]. Each player keeps their remaining pieces (P1: odds; P2: evens). History is LIFO; in HvC, undo pops two to return to the previous human turn, while reroll pops only the last NumericalComputerPlayer move. GameStorage persists the state to game1.txt.