Alexandria University
Faculty of Engineering
CSED 2025


# Paradigms Project Phase 1

# Report

Submitted to

**Dr. Mohamed M. Saad**

**Eng. Mohamed Tarek El Habiby**

**Eng. Menna El Kammah**

**Eng. Maram Aly Attia**

Submitted by

| | |
|---|---|
| **Islam Yasser Mahmoud** | **20010312** |
| **Mohamed Ehab Mabrouk** | **20011524** |
| **Mohamed Hassan Youssef** | **20011544** |
| **Mkario Michel Azer** | **20011982** |

Faculty of engineering Alex. University

May 2023

# Table of Contents

## Problem Statement:

Implementation of a Board drawing engine for drawing game boards 2 times (one using JavaScript and the other using Scala). The engine will support drawing six games: Tic-Tac-Toe, Connect-4, Checkers, Chess, Sudoku, and 8-Queens. And it is extensible to draw any other board game.

The engine has only two responsibilities:

1. Drawing the board and pieces.
2. Enforce the rules of moving pieces.

The engine will NOT:

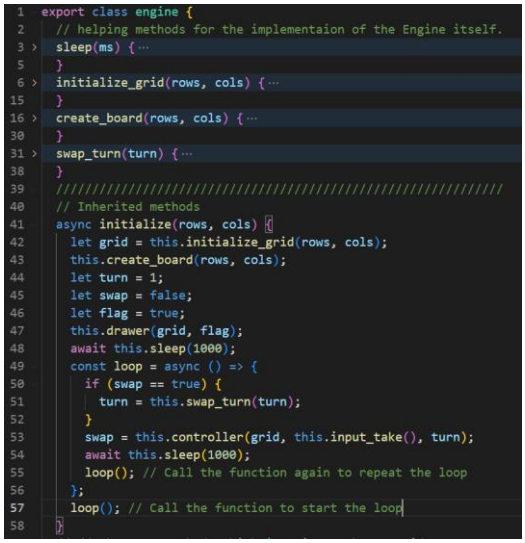
- Have intelligence for playing against the player .
- Check the winner or calculate the scores .
- Save/Load the game status.

## Main Features:

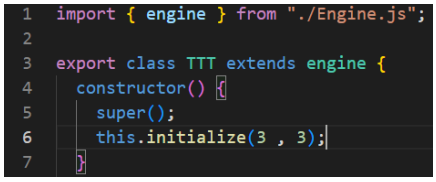
### Object Oriented Programming Features:

#### Inheritance:

Inheritance was used to avoid any repetition in the code as it contains all the repeated code including:




The Class engine shows us the methods that are inherited and the in the TTT class – as an example – we can see that TTT extends engine and in the constructor, it uses super() which is a special keyword used to call a method or constructor from the parent class within a subclass. It allows access and invoke the parent class's implementation of a method or constructor. On other words we can say that: super() is typically used in the constructor of a subclass to invoke the constructor of the parent class. By doing so, you can initialize the inherited members of the subclass and perform any additional operations specific to the subclass. Then we call initialize method from the parent class to start initializing the game.




#### Encapsulation:

The Games' classes encapsulate related data and behavior within themselves. They encapsulate the game logic, board state, and methods for validating and controlling as each game knows nothing about the implementation of any another one.

## Polymorphism: (Dynamic Method Selection)(overriding)

In the whole implementation method overriding was properly used as we can see in the game engine the abstract methods that all the games' classes override providing their own implementations based on the specific game logic.

```
// // Abstract methods which is going to be Overriden
drawer(grid) {
  throw new Error("Engine call");
}
controller(grid, input, turn) {
  throw new Error("Engine call");
}
cell_additions() {
  throw new Error("Engine call");
}
board_additions() {
  throw new Error("Engine call");
}
input_take() {
  throw new Error("Engine call");
}
input_validation(grid, input, turn) {
  throw new Error("Engine call");
}
}
```

If we take a look on the TTT class – as an example – we will find the following:

- It overrides drawer(grid) method to draw the game state in its unique way besides the two helping methods for the drawer which focus on the styling:
  - The board_additions() method.
  - The cell_additions() method.

```
drawer(grid) {
  this.board_additions();
  this.cell_additions();
  const cells = document.querySelectorAll(".cell");
  for (let i = 0; i < cells.length; i++) {
    const row = cells[i].dataset.row;
    const col = cells[i].dataset.col;
    console.log(row , col);
    cells[i].textContent = grid[row][col];
  }
}
```

- It overrides the input_take() method to take the user's input in its way.
- It overrides the controller(grid , input , turn) to put its rules and validate the user's input besides its helping method which make the input validation:
  - The input_validataion(grid , input , turn) to validate the input according to the game's rules.

```
controller(grid , input , turn) {
  let valid = this.input_validation(grid , input , turn);
  if (valid == true) {
    this.drawer(grid);
    return true;
  } else {
    alert("Invalid input");
  }
}
}
```

# Functional Programming Features:

## Higher Order Functions:

We used higher order functions to send a function as a parameter to another function or to return a function as an output from another function.

- Higher order functions that take functions as parameter:

```
def gameEngine(drawer: (Array[Array[String]], JFrame, JPanel) => Array[Array[String]], controller: (String, Array[Array[String]], Int) => Any,
               initializer: () => Array[Array[String]], switchFunc: Int => Int, frame: JFrame, panel: JPanel): Unit = {
  @tailrec
  def gameLoop(drawer: (Array[Array[String]], JFrame, JPanel) => Array[Array[String]], controller: (String, Array[Array[String]], Int) => Any,
               initializer: () => Array[Array[String]], switchFunc: Int => Int, frame: JFrame, panel: JPanel, grid: Array[Array[String]], player: Int): Unit = {

    controller(scala.io.StdIn.readLine(), grid, player) match {
      case grid: Array[Array[String]] =>
        gameLoop(drawer, controller, initializer, switchFunc, frame, panel, drawer(grid, frame, panel), switchFunc(player))
      case n: Int =>
        gameLoop(drawer, controller, initializer, switchFunc, frame, panel, grid, switchFunc(n))
    }
  }
  switchFunc(2)
  gameLoop(drawer, controller, initializer, switchFunc, frame, panel, drawer(initializer(), frame, panel), player = 1)
}
```

Here gameEngine() as well as gameLoop() functions take four functions as parameter drawer(), controller(), initializer() and switchFunc()

- Higher order functions that return functions as output:

```
def TicTacToeValidation(move: String, grid: Array[Array[String]]): Int => Any = {
  Option(move).filter(_.length == 2)
```

```
def canMove(piece: String, grid: Array[Array[String]], color: Char): (Int, Int, Int, Int) => Boolean = piece match {
  case "\u265F" => checkPawn(grid, pawnDirection(color), _, _, _, _)
  case "\u265C" => checkCastle(grid, _, _, _, _)
  case "\u265E" => checkKnight
  case "\u265D" => checkBishop(grid, _, _, _, _)
  case "\u265B" => checkQueen(grid, _, _, _, _)
  case "\u265A" => checkKing
}
```

```
    .filter(_ => "abcdefg".toList.contains(move.charAt(0)))
    .filter(_ => playsInCol(grid.map(row => row(map(move.charAt(0)))).toList) < 6)
    .map(_ => Connect4UpdateGrid(playsInCol(grid.map(row => row(map(move.charAt(0)))).toList), map(move.charAt(0)), grid, _))
    .getOrElse(invalid2Players)
}
```

TicTacToeValidation(), Connect4Validation() both return a function that accepts an Int as parameter.

canMove() is a function in Chess that returns the right function to be checked corresponding to the piece to be checked for its validity of move

```
def initializers(number: Int): () => Array[Array[String]] = HashMap(
  1 -> TicTacToeInitializeGrid _,
  2 -> ChessInitializeGrid _,
  3 -> CheckersInitializeGrid _,
  4 -> Connect4InitializeGrid _,
  5 -> SudokuInitializeGrid _,
  6 -> Queens8InitializeGrid _
)(number)
```

```
def drawers(number: Int): (Array[Array[String]], JFrame, JPanel) => Array[Array[String]] = HashMap(
  1 -> TicTacToeDrawer,
  2 -> ChessDrawer,
  3 -> CheckersDrawer,
  4 -> Connect4Drawer,
  5 -> SudokuDrawer,
  6 -> Queens8Drawer
)(number)
```

```
def controllers(number: Int): (String, Array[Array[String]], Int) => Any = HashMap(
  1 -> TicTacToeController,
  2 -> ChessController,
  3 -> CheckersController,
  4 -> Connect4Controller,
  5 -> SudokuController,
  6 -> Queens8Controller
)(number)
```

Initializers(), controllers() and drawers() functions return the initializer, controller, and drawer, of the corresponding desired game that was chosen by the user.

## Pattern matching:

We used pattern matching to match certain patterns to their desired actions to be performed.

```
controller(scala.io.StdIn.readLine(), grid, player) match {
  case grid: Array[Array[String]] =>
    gameLoop(drawer, controller, initializer, switchFunc, frame, panel, drawer(grid, frame, panel), switchFunc(player))
  case n: Int =>
    gameLoop(drawer, controller, initializer, switchFunc, frame, panel, grid, switchFunc(n))
}
```

Pattern matching is useful here to check if the returned type is Array[Array[String]] or Int.

```
def CheckersIsValid(move: String, grid: Array[Array[String]], player: Int, inputPattern: Regex): Boolean = move match {
  case inputPattern(row1, col1, row2, col2) =>
    (grid(row1(0).asDigit - 1)(map(col1(0))), player, grid(row2(0).asDigit - 1)(map(col2(0))), row2(0) - row1(0), map(col2(0)) - map(col1(0))) match {
      case ("w\u23FA", 1, " ", 1, 1 | -1) | ("b\u23FA", 2, " ", -1, -1 | 1) => true
      case ("w\u23FA", 1, " ", 2, 2 | -2) | ("b\u23FA", 2, " ", -2, -2 | 2) =>
        (grid(min(row2(0).asDigit, row1(0).asDigit))(min(map(col2(0)), map(col1(0))) + 1), player) match {
          case ("w\u23FA", 2) | ("b\u23FA", 1) => true
          case _ => false
        }
      case _ => false
        }
  case _ => false
}
if(CheckersIsValid(move, grid, player, """([1-9])([a-i]) ([1-9])([a-i])""".r))
  CheckersUpdateGrid(move(0).asDigit-1, map(move(1)), move(3).asDigit-1, map(move(4)), grid)
else invalid2Players(player)
```

Here we check if user input matches some regex expression also to check if a tuple matches some criteria related to the validation logic of the Checkers game.

```
def SudokuIsValid(move: String, grid: Array[Array[String]], insertPattern: Regex, removePattern: Regex): Boolean = move match {
  case insertPattern(row, col, num) => grid(row(0).asDigit - 1)(map(col(0))) match
    case "b " => isValidPlacement(grid, row(0).asDigit - 1, map(col(0)), num(0))
    case _ => false
  case removePattern(row, col) => !grid(row(0).asDigit - 1)(map(col(0)))(0).equals('r')
    && !grid(row(0).asDigit - 1)(map(col(0)))(1).equals(' ')
  case _ => false
}
SudokuIsValid(move, grid, """([1-9])([a-i]) ([1-9])""".r, """([1-9])([a-i])""".r) match {
  case false => invalid1Player(player)
  case true => SudokuUpdateGrid(move.charAt(0).asDigit - 1, map(move.charAt(1)), grid, if(move.length == 4) move.substring( beginIndex = 3) else " ")
}
```

Also pattern matching was useful in Sudoku validation logic  to check if the user move matches the regex expression for insert or the regex expression for remove.

```
def ChessIsValid(move: String, grid: Array[Array[String]], player: Int, inputPattern: Regex): Boolean = move match {
  case inputPattern(row1, col1, row2, col2) => grid(row1(0).asDigit - 1)(map(col1(0))) match {
    case " " => false
    case piece => piece(0).equals(ChessColor(player)) &&
      !grid(row2(0).asDigit - 1)(map(col2(0)))(0).equals(ChessColor(player)) &&
      canMove(piece.substring( beginIndex = 1), grid, ChessColor(player))(row1(0).asDigit - 1, map(col1(0)), row2(0).asDigit - 1, map(col2(0)))
  }
  case _ => false
}
ChessIsValid(move, grid, player, """([1-9])([a-i]) ([1-9])([a-i])""".r) match
  case false => invalid2Players(player)
  case true => ChessUpdateGrid(move.charAt(0).asDigit - 1, map(move.charAt(1)), move.charAt(3).asDigit - 1, map(move.charAt(4)), grid)
```

In Chess we check if the user move matches the regex expression for the valid Chess input move.

```
def checkPawn(grid: Array[Array[String]], pawnDirection: Int, row1: Int, col1: Int, row2: Int, col2: Int): Boolean = (col1 == col2, row2 - row1) match {
  case (true, `pawnDirection`) => grid(row2)(col2).equals(" ")
  case (true, n) if n == 2 * pawnDirection && row1 == pawnStartRow(pawnDirection) =>
    noPieceVerticallyBetween(row1, col1, row2, col2, grid) && grid(row2)(col2).equals(" ")
  case (false, `pawnDirection`) if Math.abs(col1 - col2) == 1 => grid(row2)(col2).startsWith(if (pawnDirection == 1) "b" else "w")
  case _ => false
}
```

In checkPawn() function we use pattern matching to match each tuple to certain required action to check the validity of the pawn move.

## Regex Expressions:

It is very useful feature in functional programming to check if a certain if something matches with a specified expression.

```
"""([1-9])([a-i]) ([1-9])""".r, """([1-9])([a-i])""".r
```

In sudoku we check if the user's move matches any of those regex expressions.

```
"""([1-9])([a-i]) ([1-9])([a-i])""".r
```

In Chess and Checkers, we check if the user move matches this regex expression.

## Pure functions:

All functions access only their parameters, and no global variables exist at all. Also, local variables don't exist at all except the drawers some of them contain three val of List while others contain only one val of List.

## Recursion:

instead of using for loops and while loops we use tail recursion in the main gameEngine() function.

```
@tailrec
def randomCell(grid: Array[Array[String]], exist: Array[Array[Array[Boolean]]], i: Int, j: Int, num: Int): Unit = {
  exist(2)(i / 3 * 3 + j / 3)(num) match {
    case true => randomCell(grid, exist, i, j, between(1, 10))
    case false => grid(i).update(j, "r".concat(num.toString))
      exist(0)(i).update(num, true)
      exist(1)(j).update(num, true)
      exist(2)(i / 3 * 3 + j / 3).update(num, true)
  }
}
```

Tail recursion also was used in generation the random cells for Sudoku drawing board.

## Collections utilities

Map, flatMap, foreach, forall, flatten, zip, zipWithIndex, filter, slice all of them are much powerful utilities used for iterating through collections of data and perform certain type of action also their usage avoids the use of for loops.

# Main differences between the two paradigms:

## Object Oriented Paradigm:

- Classes and inheritance are the key for extending the code.

- Encapsulation can be done by putting functions in classes and by private membership.
- Operates on data with the ability to modify it.
- OOP provides variables.
- OOP emphasizes mutable state, where the game engine's state (e.g., player turn, game board) can be modified directly.
- OOP provides a natural way to model complex interactions between game entities through classes and inheritance.
- OOP does contain For loops.

### Functional Paradigm:

- Higher order functions are the key to extending the code.
- Encapsulation can be done by putting a function inside another function so in this way the inner function is encapsulated.
- Operates on data without modifying it.
- Functions always produce the same output for the same input. (pure)
- No classes and objects and no inheritance while the implementation logic is composed of several functions where each function output is an input for another function.
- It doesn't contain For loops.

## Pros and Cons of each paradigm:

### Object Oriented Paradigm:

#### Pros:

- **Inheritance:** Code reusing is easily done through implementing common logic in a single class (e.g., game engine) and allowing the remaining classes to inherit from it.
- **Polymorphism** can be done to create specialized entities or behaviors by extending or overriding existing classes.
- **Encapsulation:** it provides encapsulation, where each class has its own implementation for some methods.
- **Modularity:** The code promotes modularity by organizing related code into classes and methods. Each class is responsible for specific functionality, making it easier to understand, maintain, and reuse.
- **Abstraction:** The engine class defines abstract methods such as cell_additions(), board_additions(), input_take(), input_validation(), drawer(), and controller(). These methods are meant to be overridden by the derived classes (such as TTT) to provide concrete implementations.
- **Easy Collaboration:** OOP facilitates team collaboration and code development by providing a clear and standardized way to define and interact with objects. Objects can be designed and implemented independently, allowing multiple team members to work on different components simultaneously.
- Better **organization** of code than other paradigms.

#### Cons:

- **Complexity:** OOP can introduce additional complexity, The interdependence between objects and classes can make it challenging to understand the flow of control and data within the system. Proper design and planning are necessary to mitigate this complexity.
- **Performance Overhead:** OOP languages often come with additional overhead compared to lower-level languages. Features like dynamic dispatch and runtime type checking can impact performance, especially in performance-critical applications such as games or simulations.

- **Overuse of Inheritance:** Improper use of inheritance, such as excessive subclassing or deep inheritance hierarchies, can lead to code that is difficult to maintain and understand. Inheritance should be used judiciously, focusing on the "is-a" relationship, and avoiding unnecessary coupling.
- **Mutable State and Side Effects:** OOP allows for mutable state within objects, which can introduce complexities related to managing and synchronizing shared state. Mutable state can also lead to subtle bugs and make it harder to reason about program behavior.
- **Coupling:** May result in tight coupling between objects, which can make it harder to change one part of the code without affecting other parts.

## Functional Paradigm:

### Pros:

- There are many built in functions that are so powerful that **you need to specify what to do rather than how to do it,** for example **takeWhile()** is used to calculate the right row to play in Connect4 game.

```
def playsInCol(column: List[String]): Int =
  column.takeWhile(cell => !cell.equals(" ")).length
```

- Using **slice(), flatMap(), flatten()** to check if there is the same number in row, column or the subgrid in Sudoku makes it very simple.

```
def isValidPlacement(grid: Array[Array[String]], row: Int, col: Int, num: Char): Boolean = {
  !List(grid(row).toList.flatten, grid.flatMap(row => row(col)).toList, grid
    .slice(row / 3 * 3, row / 3 * 3 + 3)
    .flatMap(row => row.slice(col / 3 * 3, col / 3 * 3 + 3)).flatten.toList)
    .flatten.contains(num)
}
```

- The concept of **pure function** makes every function as a single unit of execution and independent of any other outer states which makes the code more readable and make it more testable.
- The concept of **higher order function** makes the code extensible as you can implement another game with its controller and drawer, and it will be adapted to the code.
- **Pattern matching** is a powerful tool that enables you to match complex cases such as checking for the type, tuples, lists, regex expressions.
- **Immutable Data:** Functional programming promotes immutability, where data is treated as immutable and cannot be modified once created. This eliminates many issues related to shared mutable state, making the code more reliable, predictable, and easier to reason about.
- **Readability and Maintainability:** Functional programming emphasizes writing code in a declarative style, focusing on what needs to be achieved rather than how to achieve it. This leads to concise, self-explanatory code that is easier to read, understand, and maintain.
- **Concurrency and Parallelism:** Functional programming favors the use of immutable data and pure functions, which inherently support concurrent and parallel execution. With no shared mutable state, multiple computations can be performed independently, making it easier to leverage multi-core processors and distributed systems.

- **Avoidance of State Bugs:** By minimizing mutable state and side effects, functional programming reduces the likelihood of state-related bugs, such as race conditions, deadlocks, and inconsistent data. This can result in more robust and bug-free codes.

**Cons:**

- **Limited Mutability:** While immutability is beneficial for many scenarios, there are cases where mutable state is necessary or more efficient. Functional programming restricts the use of mutable state, which can lead to more complex workarounds or performance trade-offs in certain situations. As the implementation of the games would become easier if the mutability was allowed as it requires changing player turns, game board content etc.…
- It is **not possible to store a flag array** for example to indicate something in the game logic since it is required that each function is stateless and pure so it may lead to a less efficient performance.
- **Complex transformation** between functions without using global state leads to complex understanding in some parts of the code.
- **Performance Overhead:** Functional programming languages often prioritize immutability and pure functions, which can introduce some performance overhead. Immutable data structures require copying or updating entire data structures instead of modifying in-place, which can impact performance in certain scenarios.
- **Integration Challenges:** Functional programming languages and paradigms may not seamlessly integrate with existing imperative or object-oriented codebases. Mixing different paradigms can result in code that is harder to understand and maintain.
- **Lack of Familiarity:** Functional programming may not be as widely adopted or supported in certain domains or industries. Finding skilled functional programmers or libraries specific to functional programming paradigms might be more challenging compared to mainstream imperative or object-oriented languages.