Namita Raghuvanshi A02310449

HW4-Alternative B – Leveraging Reuse with Template Methods

Introduction

In this project, we are developing a Sudoku Solver which is open to extension and closed to modification. Our focus is to keep the solver simple, modular, reusable and implement template pattern in different strategies that solve the puzzle.

Aim

Our aim to is to become familiar with Template pattern.

UML Diagrams

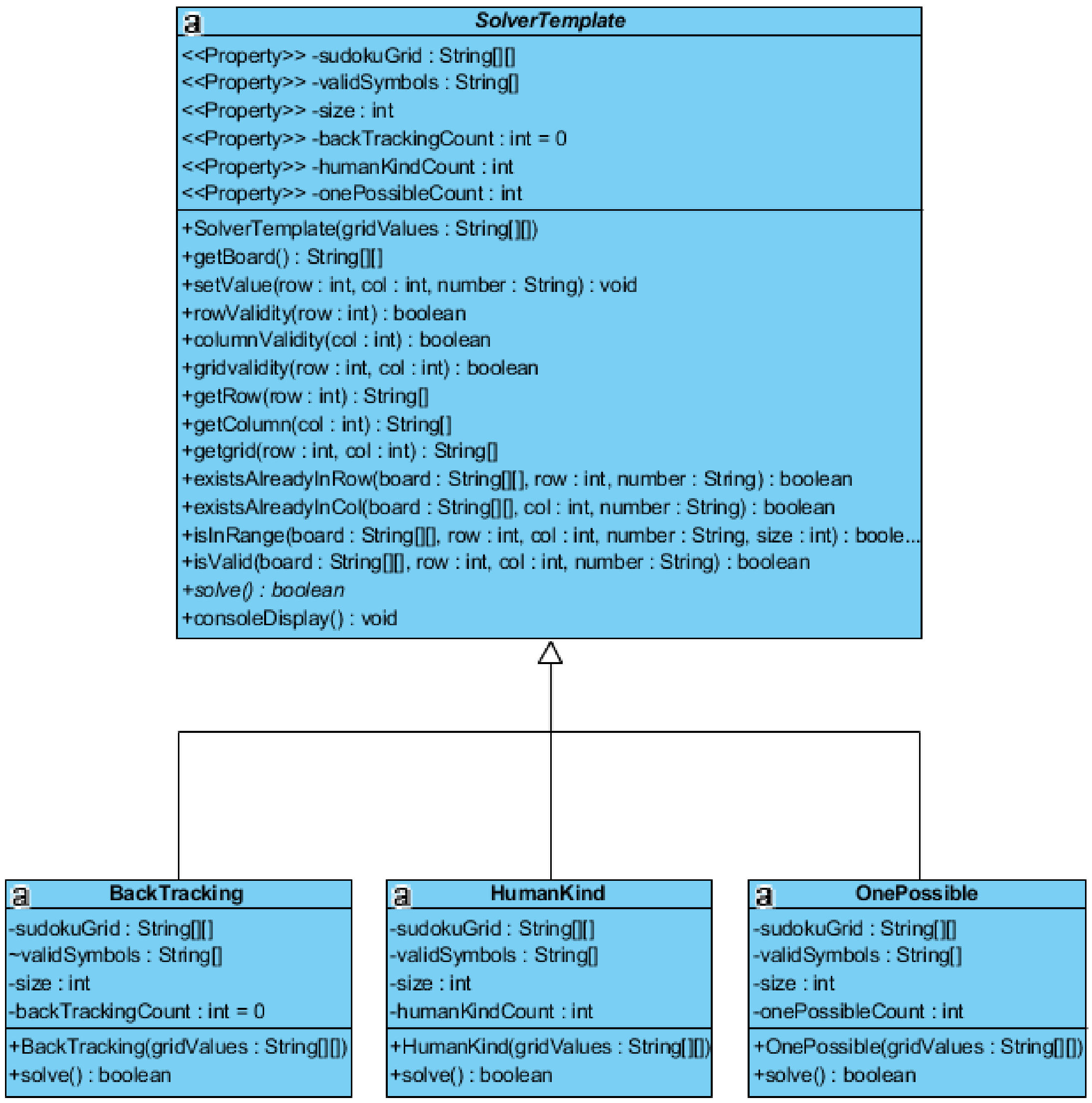
SolverTemplate
<<Property>> -sudokuGrid : String[][]
<<Property>> -validSymbols : String[]
<<Property>> -size : int
<<Property>> -backTrackingCount : int = 0
<<Property>> -humanKindCount : int
<<Property>> -onePossibleCount : int
+SolverTemplate(gridValues : String[][])
+getBoard() : String[][]
+setValue(row : int, col : int, number : String) : void
+rowValidity(row : int) : boolean
+columnValidity(col : int) : boolean
+gridvalidity(row : int, col : int) : boolean
+getRow(row : int) : String[]
+getColumn(col : int) : String[]
+getgrid(row : int, col : int) : String[]
+existsAlreadyInRow(board : String[][], row : int, number : String) : boolean
+existsAlreadyInCol(board : String[][], col : int, number : String) : boolean
+isInRange(board : String[][], row : int, col : int, number : String, size : int) : boole...
+isValid(board : String[][], row : int, col : int, number : String) : boolean
+solve() : boolean
+consoleDisplay() : void
BackTracking
-sudokuGrid : String[][]
~validSymbols : String[]
-size : int
-backTrackingCount : int = 0
+BackTracking(gridValues : String[][])
+solve() : boolean
HumanKind
-sudokuGrid : String[][]
-validSymbols : String[]
-size : int
-humanKindCount : int
+HumanKind(gridValues : String[][])
+solve() : boolean
OnePossible
-sudokuGrid : String[][]
-validSymbols : String[]
-size : int
-onePossibleCount : int
+OnePossible(gridValues : String[][])
+solve() : boolean

Fig. 1.1 Class Diagram of template pattern

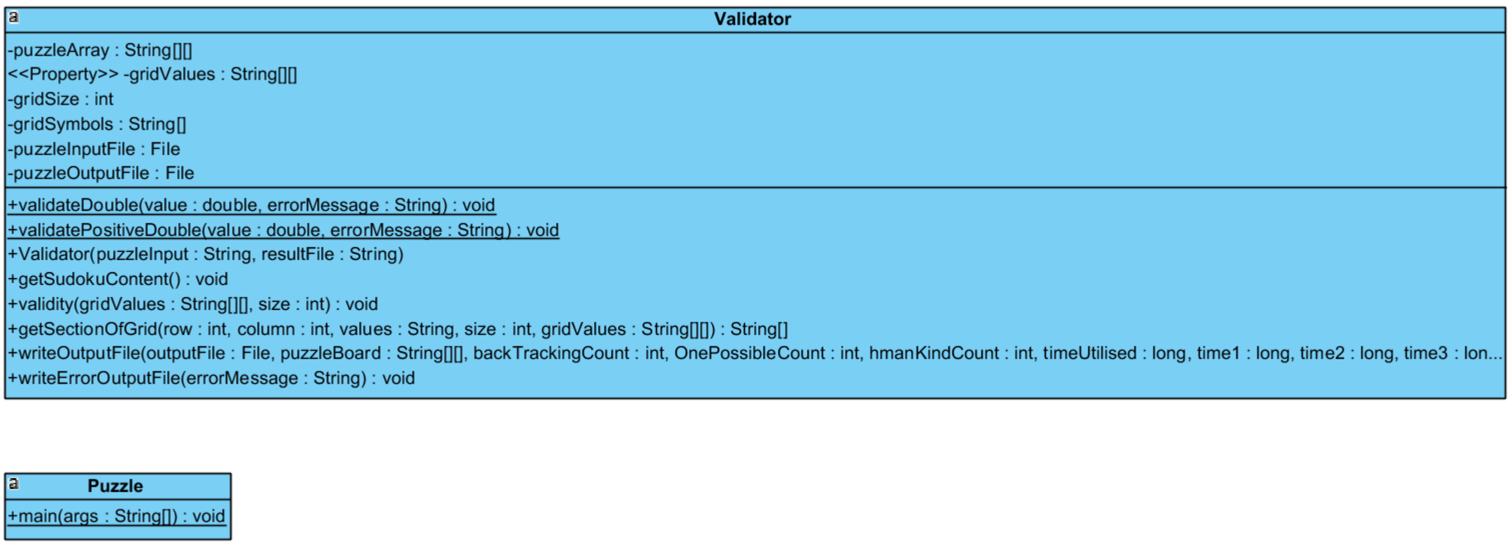
Validator
-puzzleArray : String[][]
<<Property>> -gridValues : String[][]
-gridSize : int
-gridSymbols : String[]
-puzzleInputFile : File
-puzzleOutputFile : File
+validateDouble(value : double, errorMessage : String) : void
+validatePositiveDouble(value : double, errorMessage : String) : void
+Validator(puzzleInput : String, resultFile : String)
+getSudokuContent() : void
+validity(gridValues : String[][], size : int) : void
+getSectionOfGrid(row : int, column : int, values : String, size : int, gridValues : String[][]) : String[]
+writeOutputFile(outputFile : File, puzzleBoard : String[][], backTrackingCount : int, OnePossibleCount : int, hmanKindCount : int, timeUtilised : long, time1 : long, time2 : long, time3 : lon...
+writeErrorOutputFile(errorMessage : String) : void
Puzzle
+main(args : String[]) : void

Fig. 1.2 Class Diagram of validator

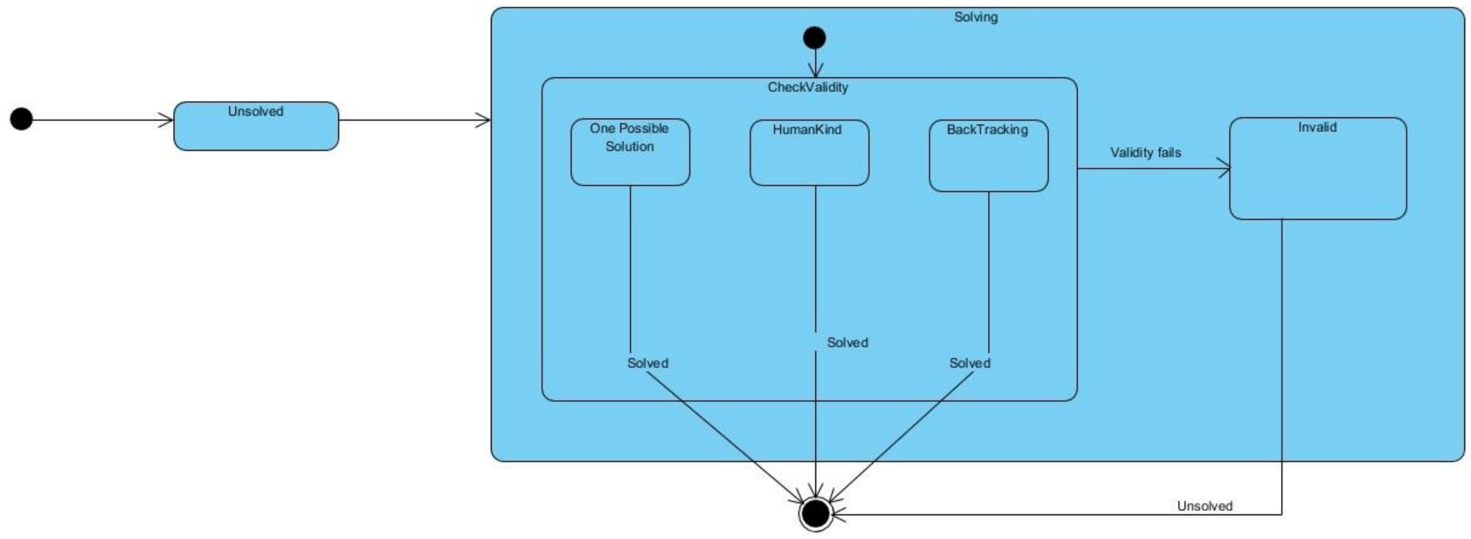

Fig 2.1 State Chart Diagram.

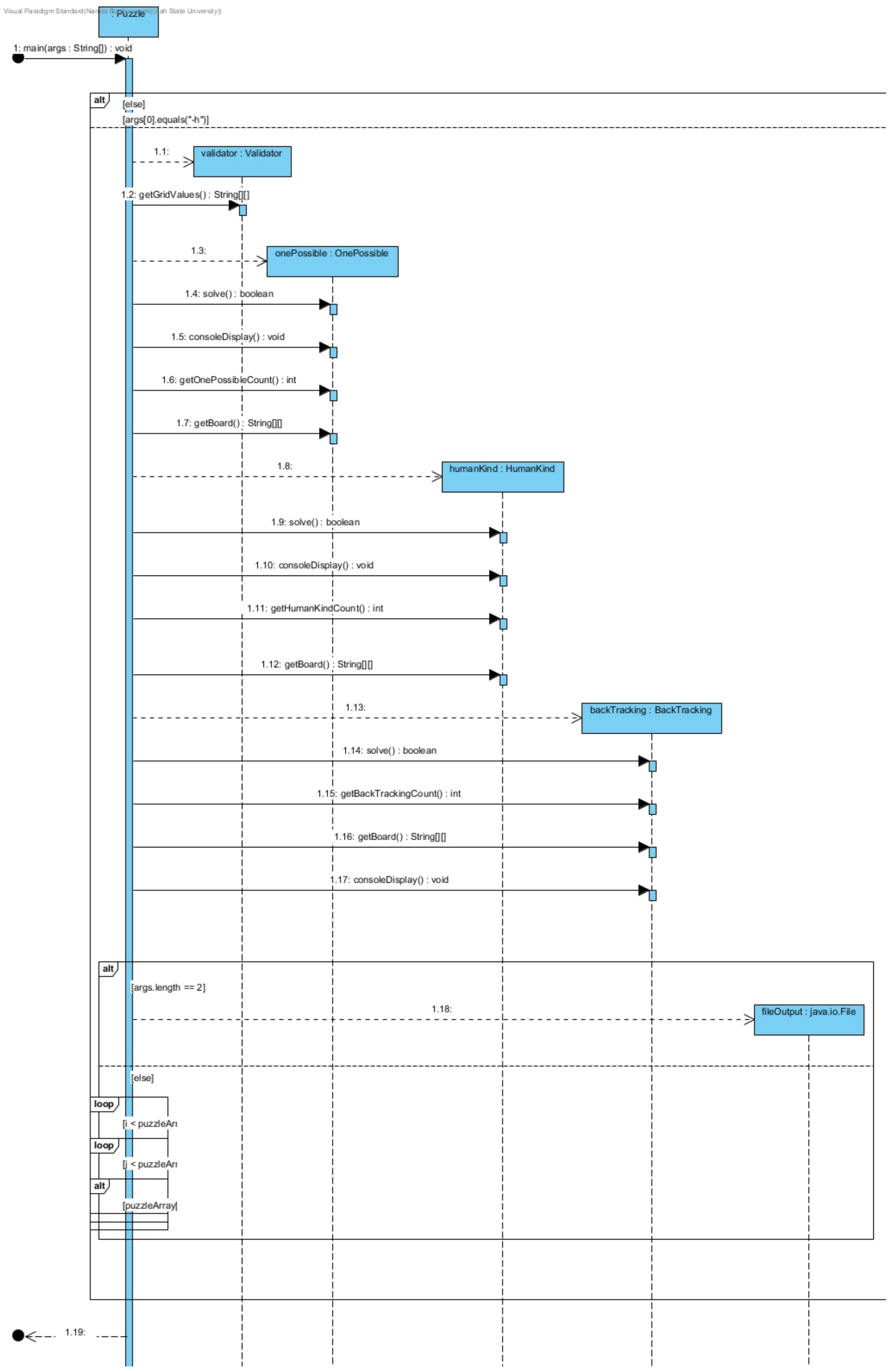
Visual Paradigm Standard(Namita Raghuvanshi(Utah State University))
: Puzzle
1: main(args : String[]) : void
alt
[else]
[args[0].equals("-h")]
1.1:
validator : Validator
1.2: getGridValues() : String[][]
1.3:
onePossible : OnePossible
1.4: solve() : boolean
1.5: consoleDisplay() : void
1.6: getOnePossibleCount() : int
1.7: getBoard() : String[][]
1.8:
humanKind : HumanKind
1.9: solve() : boolean
1.10: consoleDisplay() : void
1.11: getHumanKindCount() : int
1.12: getBoard() : String[][]
1.13:
backTracking : BackTracking
1.14: solve() : boolean
1.15: getBackTrackingCount() : int
1.16: getBoard() : String[][]
1.17: consoleDisplay() : void
alt
[args.length == 2]
1.18:
fileOutput : java.io.File
[else]
loop
[i < puzzleArr
loop
[j < puzzleArr
alt
[puzzleArray]
1.19:

Fig 3.1 Interaction Diagram

Insights Uncovered

-While doing the project I have learned a lot about template pattern and I also know it now that implementing a code with this pattern saves a lot of duplicate code because we code for the template once which is common to all the strategies and the part which is different for all the strategies is done separately in their classes .

-While doing the project I have learned a lot about testing and I also know it now that creating a code with better modularity, encapsulation and abstraction leads to easily testable cases.

-While trying to figure out how can we check the validity of a puzzle before actually starting to solve it, I learned that we should create a comprehensive validation check which checks the puzzle in not just one way but which checks all the aspects of the puzzle. This is not just the case for sudoku puzzle but it should be the same for all programs where there is some king of validity check required

- Also I have faced the fact that every time we should try to figure out new methods and ways rather than coding just with the pre-existing knowledge.

-one of my bad practice that I tackled while doing this project is that we should always be open to new ideas and strategy rather than thinking in one direction. Because sometimes the only solution is to change the whole plan and get started again from a new point. I have wasted a lot of time trying to figure out things where all I needed was to think differently about a new solution.

-I have also figured out that the naming conventions are not to be taken lightly as it could make the code look messy and also it won't be readable for someone else. Also it is time wasting to debug a code which does not have proper names.

- I also learned that we should always code in such a way that further add-ons and changes should not be a cumbersome task.