Assignment 1 — ADT Specification — *Solution*

ADT Specification

```
type DisplayType
declare newDisplayType() → DisplayType
end DisplayType.

type InputType
declare newInputType() → InputType
end InputType.

type IndexType
import Boolean
declare newIndexType() → IndexType
        increment(IndexType) → IndexType
        decrement(IndexType) → IndexType
        lessThan(IndexType,IndexType) → Boolean
        greaterThan(IndexType,IndexType) → Boolean
        equals(IndexType,IndexType) → Boolean
        clone(IndexType) → IndexType
end IndexType.

type ColourType
import DisplayType, Boolean
declare newColourType() → ColourType
        invert(ColourType) → ColourType
        equals(ColourType,ColourType) → Boolean
        showColour(ColourType,DisplayType,LocationType) → ColourType
end ColourType.

type DimensionType
import IndexType, Boolean
declare newDimensionType(IndexType) → DimensionType
        lessThan(DimensionType,DimensionType) → Boolean
        greaterThan(DimensionType,DimensionType) → Boolean
        equals(DimensionType,DimensionType) → Boolean
        clone(DimensionType) → DimensionType
end DimensionType.

type WorthType
import Boolean
declare newWorthType() → WorthType
        increment(WorthType) → WorthType
        decrement(WorthType) → WorthType
        lessThan(WorthType,WorthType) → Boolean
        greaterThan(WorthType,WorthType) → Boolean
        equals(WorthType,WorthType) → Boolean
        clone(WorthType) → WorthType
end WorthType.

type LevelNumberType
import Boolean
declare newLevelNumberType() → LevelNumberType
```

```
        increment(LevelNumberType) → LevelNumberType
        decrement(LevelNumberType) → LevelNumberType
        lessThan(LevelNumberType,LevelNumberType) → Boolean
        greaterThan(LevelNumberType,LevelNumberType) → Boolean
        equals(LevelNumberType,LevelNumberType) → Boolean
        clone(WorthType) → WorthType
end LevelNumberType.

type LocationType [IndexType, IndexType]
import IndexType
declare newLocationType() → LocationType
        setRow(LocationType,IndexType) → LocationType
        getRow(LocationType) → IndexType
        setColumn(LocationType,IndexType) → LocationType
        getColumn(LocationType) → IndexType
        clone(LocationType) → LocationType
end LocationType.

type SymbolType [Boolean, ColourType]
import ColourType, Boolean, DisplayType, LocationType
declare newSymbolType() → SymbolType
        newSymbolType(ColourType) → SymbolType
        setColour(SymbolType,ColourType) → SymbolType
        getColour(SymbolType) → ColourType
        makeEmpty(SymbolType) → SymbolType
        isEmpty(SymbolType) → Boolean
        equals(SymbolType,SymbolType) → Boolean
        clone(SymbolType) → SymbolType
        showSymbol(SymbolType,DisplayType,LocationType) → SymbolType
end SymbolType.

type PlayerType [SymbolType]
import SymbolType, Boolean
declare newPlayerType(SymbolType) → PlayerType
        setSymbol(PlayerType,SymbolType) → PlayerType
        getSymbol(PlayerType) → SymbolType
        equals(PlayerType,PlayerType) → Boolean
        opponent(PlayerType,PlayerType,PlayerType) → PlayerType
        clone(PlayerType) → PlayerType
end PlayerType.

type SquareType [SymbolType, LocationType]
import SymbolType, LocationType, Boolean, DisplayType
declare newSquareType(LocationType) → SquareType
        newSquareType(LocationType,SymbolType) → SquareType
        setLocation(SquareType,LocationType) → SquareType
        getLocation(SquareType) → LocationType
        setSymbol(SquareType,SymbolType) → SquareType
        getSymbol(SquareType) → SymbolType
        isEmpty(SquareType) → Boolean
        clone(SquareType) → SquareType
        showSquare(SquareType, DisplayType) → SquareType
end SquareType.

type GridType [SquareType, …, SquareType, DimensionType, WorthType]
import DimensionType, Boolean, SquareType, LocationType, WorthType,
                  SymbolType, DisplayType
```

```
declare newGridType() → GridType
        newGridType(DimensionType) → GridType
        newGridType(DimensionType,LocationType,SymbolType) → GridType
        squareOccupied(GridType,LocationType) → Boolean
        occupySquare(GridType,LocationType,SymbolType) → GridType
        setSquare(GridType,SquareType) → GridType
        getSquare(GridType,LocationType) → SquareType
        setDimension(GridType,DimensionType) → GridType
        getDimension(GridType) → DimensionType
        setWorth(GridType,WorthType) → GridType
        getWorth(GridType) → WorthType
        getSymbol(GridType,LocationType) → SymbolType
        validMove(GridType,LocationType) → Boolean
        fullGrid(GridType) → Boolean
        gameOver(GridType) → Boolean
        draw(GridType) → Boolean
        win(GridType) → SymbolType
        diagWin(GridType) → SymbolType
        horizWin(GridType) → SymbolType
        vertWin(GridType) → SymbolType
        evaluateGrid(GridType,PlayerType) → WorthType
        evaluateRows(GridType,PlayerType) → WorthType
        evaluateColumns(GridType,PlayerType) → WorthType
        evaluateDiagonals(GridType,PlayerType) → WorthType
        equals(GridType,GridType) → Boolean
        clone(GridType) → GridType
        showGrid(GridType,DisplayType) → GridType
end GridType.

type GameTreeType [GridType, GameTreeType, GameTreeType, GameTreeType,
                   LevelNumberType]
import Boolean, GridType, LevelNumberType
declare newGameTreeType() → GameTreeType
        newGameTreeType(GridType,LevelNumberType) → GameTreeType
        newGameTreeType(GridType,LevelNumberType,GameTreeType) →
                   GameTreeType
        isEmpty(GameTreeType) → Boolean
        setGrid(GameTreeType,GridType) → GameTreeType
        getGrid(GameTreeType) → GridType
        setParent(GameTreeType,GameTreeType) → GameTreeType
        getParent(GameTreeType) → GameTreeType
        setChild(GameTreeType,GameTreeType) → GameTreeType
        getChild(GameTreeType) → GameTreeType
        setSibling(GameTreeType,GameTreeType) → GameTreeType
        getSibling(GameTreeType) → GameTreeType
        setLevel(GameTreeType,LevelNumberType) → GameTreeType
        getLevel(GameTreeType) → LevelNumberType
        buildGame(GameTreeType,GridType,PlayerType,PlayerType,LevelNum
                   berType) → GameTreeType
        generateLevel(GameTreeType,PlayerType,PlayerType) →
                   GameTreeType
        chooseBest(GameTreeType) → WorthType
        findBest(GameTreeType,WorthType) → GameTreeType
        findMove(GameTreeType) → GameTreeType
        adjustLevel(GameTreeType) → GameTreeType
```

```
end GameTreeType.

type Connect4Type [InputType, DisplayType, GameTreeType, PlayerType,
                   PlayerType, LevelNumberType]
import GridType, LevelNumberType, InputType, DisplayType,
                   GameTreeType, PlayerType, DimensionType,
                   LocationType, Boolean
declare newConnect4Type(InputType,DisplayType) → Connect4Type
        setInput(Connect4Type) → Connect4Type
        getInput(Connect4Type) → InputType
        setDisplay(Connect4Type,DisplayType) → Connect4Type
        getDisplay(Connect4Type) → DisplayType
        setGameTree(Connect4Type,GameTreeType) → Connect4Type
        getGameTree(Connect4Type) → GameTreeType
        setPlayer1(Connect4Type) → Connect4Type
        getPlayer1(Connect4Type,PlayerType) → PlayerType
        setPlayer2(Connect4Type) → Connect4Type
        getPlayer2(Connect4Type,PlayerType) → PlayerType
        setLevelNumber(Connect4Type,LevelNumberType) → Connect4Type
        getLevelNumber(Connect4Type) → LevelNumberType
        whoStarts(Connect4Type) → PlayerType
        whatSizeGrid(Connect4Type) → DimensionType
        howSmartIsComputer(Connect4Type) → LevelNumberType
        whatIsHumanMove(Connect4Type) → LocationType
        movePossible(Connect4Type,LocationType) → Boolean
        makeHumanMove(Connect4Type,LocationType) → Connect4Type
        makeComputerMove(Connect4Type) → Connect4Type
        gameOver(Connect4Type) → Boolean
        play(Connect4Type) → Connect4Type
        showConnect4(Connect4Type) → Connect4Type
end Connect4Type.
```

Function Description

`DisplayType` defines the destination for program output; it is a link from the program to the user. The details are left unspecified and are dependent upon the implementation.

`newDisplayType() → DisplayType`

Used to create a display value.

`InputType` defines the origin of information for the game; it is a link from the user to the program. The details are left unspecified and are dependent upon the implementation.

`newInputType() → InputType`

Used to create an input value.

`IndexType` defines the possible row/column numbers that comprise directions to a square on a grid. The range of values are whole numbers from 1 upwards.

`newIndexType() → IndexType`

Used to create an index value. By default, a value of 1 is generated. *Can be implemented in JAVA directly.*

`increment(IndexType) → IndexType`

Used to update the given index value by increasing it by one. *Can be implemented in JAVA directly.*

`decrement(IndexType) → IndexType`

Used to update the given index value by decreasing it by one. *Can be implemented in JAVA directly.*

`lessThan(IndexType,IndexType) → Boolean`

Used to compare two `IndexType` values returning `true` if the first is numerically less than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`greaterThan(IndexType,IndexType) → Boolean`

Used to compare two `IndexType` values returning `true` if the first is numerically greater than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`equals(IndexType,IndexType) → Boolean`

Used to compare two `IndexType` values returning `true` if the first is numerically equivalent to the second and `false` otherwise. *Can be implemented in JAVA directly.*

`clone(IndexType) → IndexType`

Used to copy the given index value returning the copy. *Can be implemented in JAVA directly.*

`ColourType` defines the possible colours for a square on the grid. The only values are the colours 'red' and 'blue'.

`newColourType() → ColourType`

Used to create a colour. By default, the colour is 'red'.

`invert(ColourType) → ColourType`

Used to swap between colour values, i.e. if a 'red' value is provided 'blue' is returned and vice-versa.

`equals(ColourType,ColourType) → Boolean`

Used to compare two `ColourType` values returning `true` if the first is equivalent to the second and `false` otherwise. *Can be implemented in JAVA directly.*

`showColour(ColourType,DisplayType,LocationType) → ColourType`

Used to display the given colour on the given display at the given logical location.

`DimensionType` defines the possible horizontal and vertical capacity of a grid by specifying the side length. The range of numbers are whole numbers from 4 upwards.

`newDimensionType() → DimensionType`

Used to create a dimension value. By default a value of 4 is returned. *Can be implemented in JAVA directly.*

`lessThan(DimensionType,DimensionType) → Boolean`

Used to compare the two given dimension values and returns `true` if the first is numerically less than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`greaterThan(DimensionType,DimensionType) → Boolean`

Used to compare the two given dimension values and returns `true` if the first is numerically greater than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`equals(DimensionType,DimensionType) → Boolean`

Used to check whether the two given dimension variables have the same values, returning `true` if so `false` otherwise. *Can be implemented in JAVA directly.*

`clone(DimensionType) → DimensionType`

Used to copy the given dimension value, returning the copy. *Can be implemented in JAVA directly.*

`WorthType` defines the possible numeric values for the worth of a move on the grid, i.e. how optimistic the outcome of the game appears. Nothing is specified about the range of values other than that they are whole numbers.

`newWorthType() → WorthType`

Used to create a worth. By default a value of 0 is returned. *Can be implemented in JAVA directly.*

`increment(WorthType) → WorthType`

Used to update the given worth value by increasing it by one. *Can be implemented in JAVA directly.*

`decrement(WorthType) → WorthType`

Used to update the given worth value by decreasing it by one. *Can be implemented in JAVA directly.*

`lessThan(WorthType,WorthType) → Boolean`

Used to compare the two given worth values and returns `true` if the first is numerically smaller than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`greaterThan(WorthType,WorthType) → Boolean`

Used to compare the two given worth values and returns `true` if the first is numerically larger than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`equals(WorthType,WorthType) → Boolean`

Used to check whether the two given variables have the same numeric value, returning `true` if so `false` otherwise. *Can be implemented in JAVA directly.*

`clone(WorthType) → WorthType`

Used to copy the given worth value returning the copy. *Can be implemented in JAVA directly.*

`LevelNumberType` defines the possible numeric values for the labels of the levels of the game-tree nodes. The range of values are whole numbers from 0 upwards.

`newLevelNumberType() → LevelNumberType`

Used to create a level number. By default a value of 0 is returned. *Can be implemented in JAVA directly.*

`increment(LevelNumberType) → LevelNumberType`

Used to update the given level number value by increasing it by one. *Can be implemented in JAVA directly.*

`decrement(LevelNumberType)` → `LevelNumberType`

Used to update the given level number value by decreasing it by one. *Can be implemented in JAVA directly.*

`lessThan(LevelNumberType, LevelNumberType)` → `Boolean`

Used to compare the two given level number values and returns `true` if the first is numerically smaller than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`greaterThan(LevelNumberType, LevelNumberType)` → `Boolean`

Used to compare the two given level number values and returns `true` if the first is numerically larger than the second and `false` otherwise. *Can be implemented in JAVA directly.*

`equals(LevelNumberType, LevelNumberType)` → `Boolean`

Used to check whether the two given variables have the same numeric value, returning `true` if so `false` otherwise. *Can be implemented in JAVA directly.*

`clone(LevelNumberType)` → `LevelNumberType`

Used to copy the given level number value returning the copy. *Can be implemented in JAVA directly.*

`LocationType` defines the possible square locations on a grid. It consists of a row number and column number value.

`newLocationType()` → `LocationType`

Used to create a location value with default row and column number values.

`setRow(LocationType,IndexType)` → `LocationType`

Used to update the row component of the given location to the given index value, returning the updated location.

`getRow(LocationType)` → `IndexType`

Used to extract the row component from the given location, returning its value.

`setColumn(LocationType,IndexType)` → `LocationType`

Used to update the column component of the given location to the given index value, returning the updated location.

`getColumn(LocationType)` → `IndexType`

Used to extract the column component from the given location, returning its value.

`clone(LocationType)` → `LocationType`

Used to copy the given location value returning the copy. *Calls `newLocationType()`, `getRow()`, `setRow()`, `getColumn()`, `setColumn()`, and `IndexType.clone()`.*

`SymbolType` defines the concept of players of the game. It consists of a 'flag' to indicate whether there is a symbol in place and (if so) what its colour is.

`newSymbolType()` → `SymbolType`

Used to create a symbol value. By default, the symbol is empty and has no defined colour.

`newSymbolType(ColourType)` → `SymbolType`

Used to create a non-empty symbol value with the given colour.

`setColour(SymbolType,ColourType)` → `SymbolType`

Used to set the colour of the given variable to the given value, returning the updated variable.

`getColour(SymbolType)` → `ColourType`

Used to return the colour of the given variable, returning the colour (which may be undefined).

`makeEmpty(SymbolType)` → `SymbolType`

Used to empty the symbol from a square (if one was present) leaving an empty square (of undefined colour) which is returned.

`isEmpty(SymbolType)` → `Boolean`

Used to return whether or not the given variable is a square with no symbol, returns `true` if so `false` otherwise.

`equals(SymbolType,SymbolType)` → `Boolean`

Used to return whether or not the two given variables are either both empty or of the same symbol value, returns `true` if so `false` otherwise. *Calls* `isEmpty()`, `getColour()` *and* `ColourType.equals()`.

`clone(SymbolType)` → `SymbolType`

Used to copy the given variable, returning the copy. *Calls* `newSymbolType()`, `getColour()`, `ColourType.clone()`, `setColour()`, `isEmpty()` *and* `makeEmpty()`.

`showSymbol(SymbolType,DisplayType,LocationType)`

Used to provide a screen representation of a symbol value on the given screen at the given logical location. *Calls* `getColour()` *and* `showColour()`.

`PlayerType` defines the concept of players of the game. It consists of a symbol.

`newPlayerType()` → `PlayerType`

Used to create a player value. By default, the player has an empty symbol.

`setSymbol(PlayerType,SymbolType)` → `PlayerType`

Used to set the symbol of the given variable to the given value, returning the updated variable.

`getSymbol(PlayerType)` → `SymbolType`

Used to return the symbol of the given variable, returning the symbol.

`opponent(PlayerType,PlayerType,PlayerType)` → `PlayerType`

Used to check whether the first given parameter is the same as the second or the third. If the same as the second then the third is returned, if the same as the third then the second is returned. *Calls* `equals()`.

`equals(PlayerType,PlayerType)` → `Boolean`

Used to check whether the two given variables have the same symbols, returning `true` if so `false` otherwise. *Calls* `getSymbol()` *and* `SymbolType.equals()`.

`clone(PlayerType)` → `PlayerType`

Used to copy the given variable, returning the copy. *Calls* `getSymbol()`, `SymbolType.clone()`, *and* `setSymbol()`.

`SquareType` defines the squares on a grid. It consists of a symbol and the logical location of the square on the grid.

`newSquareType(LocationType) → SquareType`

Used to create a square value (conceptually located at the given location), returning the variable. By default, the square is empty. *Calls* `SymbolType.newSymbolType()`.

`newSquareType(LocationType,SymbolType) → SquareType`

Used to create a square value (conceptually located at the given location and consisting of the given symbol), returning the variable.

`clone(SquareType) → SquareType`

Used to copy the given square variable, returning the copy. *Calls* `getLocation()`, `LocationType.clone()`, `getSymbol()`, `SymbolType.clone()` *and* `newSquareType()`.

`setLocation(SquareType,LocationType) → SquareType`

Used to set the location of the given variable to the given value, returning the updated variable.

`getLocation(SquareType) → LocationType`

Used to return the location of the given variable, returning the location.

`setSymbol(SquareType,SymbolType) → SquareType`

Used to set the symbol of the given variable to the given value, returning the updated variable.

`getSymbol(SquareType) → SymbolType`

Used to return the symbol of the given variable, returning the symbol.

`isEmpty(SquareType) → Boolean`

Used to return whether or not the given variable is an empty square, returns `true` if so `false` otherwise. *Calls* `getSymbol()` *and* `SymbolType.isEmpty()`.

`showSquare(SquareType,DisplayType)`

Used to provide a screen representation of the given square value on the given display. *Calls* `getSymbol()`, `getLocation()` *and* `showSymbol()`.

`GridType` defines the representation of a grid (i.e. the state of the game). It consists of a collection of squares, a dimension, and the numeric worth of the board from a player's perspective.

`newGridType() → GridType`

Used to create a grid of empty squares, with default worth and dimension. *Calls* `newDimensionType()`, `setDimension()`, `newLocationType()`, `LocationType.getRow()`, `LocationType.getColumn()`, `IndexType.increment()`, `LocationType.setRow()`, `LocationType.setColumn()`, `newSquareType()`, `setSquare()`, `newWorthType()`, *and* `setWorth()`.

`newGridType(DimensionType) → GridType`

Used to create a grid (with given grid dimension) of empty squares with default worth. *Calls* `setDimension()`, `newLocationType()`, `LocationType.getRow()`, `LocationType.getColumn()`, `IndexType.increment()`, `LocationType.setRow()`, `LocationType.setColumn()`,

newSquareType(), setSquare(), newWorthType(), and setWorth().

`newGridType(DimensionType,LocationType,SymbolType) → GridType`

Used to create a variable of this type (with given grid dimension and symbol at the given location), returning the variable. *Calls* `SquareType.newSquareType()`, `setSquare()`, `validMove()`, `occupySquare()`, `newWorthType()`, *and* `setWorth()`

`squareOccupied(GridType,LocationType) → Boolean`

Used to discern whether the indicated square on the given grid is occupied, returns `true` if so, `false` otherwise. *Calls* `getSquare()`, *and* `SquareType.isEmpty()`.

`occupySquare(GridType,LocationType,SymbolType) → GridType`

Used to set the square of given location on the given grid to the given symbol, returning the updated grid. *Calls* `getSquare()`, `SquareType.isEmpty()`, *and* `setSquare()`.

`setSquare(GridType,SquareType) → GridType`

Used to set the indicated square on the given grid to the given square value. The resulting grid is returned. *Calls* `SquareType.getLocation()`, `LocationType.getRow()` *and* `LocationType.getColumn()`.

`getSquare(GridType,LocationType) → SquareType`

Used to obtain the symbol in the given location of the given grid. This symbol is returned. *Calls* `LocationType.getRow()` *and* `LocationType.getColumn()`.

`setDimension(GridType,DimensionType) → GridType`

Used to set the dimension of the given grid to the given value. The updated grid is returned.

`getDimension(GridType) → DimensionType`

Used to obtain the dimension of the given grid. This value is returned.

`setWorth(GridType,WorthType) → GridType`

Used to set the worth of the given grid to the given value. The updated grid is returned.

`getWorth(GridType) → WorthType`

Used to obtain the worth of the given grid. This value is returned.

`validMove(GridType,LocationType) → Boolean`

Used to discern whether the indicated square is on the given grid, returns `true` if so, `false` otherwise. *Calls* `getDimension()`, `LocationType.getRow()`, `LocationType.getColumn()`, `IndexType.lessThan()` *and* `IndexType.greaterThan()`.

`fullGrid(GridType) → Boolean`

Used to discern whether the indicated grid is completely occupied by non-empty symbols, returns `true` if so, `false` otherwise. *Calls* `newLocationType()`, `LocationType.getRow()`, `LocationType.getColumn()`,`IndexType.increment()`, `IndexType.lessThan()`, `getSquare()`, `SquareType.getSymbol()` *and* `SymbolType.isEmpty()`.

`gameOver(GridType) → Boolean`

Used to discern whether the game on the indicated grid is ended due to either a draw or a win, returns `true` if so, `false` otherwise. *Calls* `draw()` *and* `win()`.

`draw(GridType) → Boolean`

Used to discern whether the game on the indicated grid is ended due to a draw, returns `true` if so, `false` otherwise. *Calls* `fullGrid()` *and* `win()`.

`win(GridType) → SymbolType`

Used to discern whether the game on the indicated grid is ended due to a win, returns `true` if so, `false` otherwise. *Calls* `diagWin()`, `horizWin()` *and* `vertWin()`.

`diagWin(GridType) → SymbolType`

Used to discern whether the game on the indicated grid is ended due to a win in a diagonal sequence, returns `true` if so, `false` otherwise. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`horizWin(GridType) → SymbolType`

Used to discern whether the game on the indicated grid is ended due to a win in a vertical sequence, returns `true` if so, `false` otherwise. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`vertWin(GridType) → SymbolType`

Used to discern whether the game on the indicated grid is ended due to a win in a horizontal sequence, returns `true` if so, `false` otherwise. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`evaluateGrid(GridType,PlayerType) → WorthType`

Used to calculate the worth of the grid from the given player's perspective. *Calls* `evaluateRows()`, `evaluateColumns()` *and* `evaluateDiagonals()`.

`evaluateRows(GridType,PlayerType) → WorthType`

Used to calculate the worth of the grid's rows from the given player's perspective. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`evaluateColumns(GridType,PlayerType) → WorthType`

Used to calculate the worth of the grid's columns from the given player's perspective. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`evaluateDiagonals(GridType,PlayerType)` → `WorthType`

Used to calculate the worth of the grid's diagonals from the given player's perspective. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.greaterThan()`, `IndexType.increment()` *and* `IndexType.decrement()`.

`equals(GridType,GridType)` → `Boolean`

Used to discern whether the two grids are identical sizes and contain identical symbols, returns `true` if so, `false` otherwise. *Calls* `getSquare()`, `SquareType.isEmpty()`, `SquareType.getSymbol()`, `SymbolType.equals()`, `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`,`IndexType.lessThan` *and* `IndexType.increment()`.

`clone(GridType)` → `GridType`

Used to copy the given variable, returning the copy. *Calls* `getDimension()`, `newGridType()`, `getWorth()`, `WorthType.clone()`, `setWorth()`, `occupySquare()`, `getSquare()`, `SquareType.clone()` *and* `squareOccupied()`.

`showGrid(GridType,DisplayType)`

Used to provide a screen representation of the given grid on the given display. *Calls* `getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.increment()`, `getSquare()` *and* `showSquare()`.

`GameTreeType` defines the representation of a game-tree. It consists of a grid, 'parent' (previous move) , 'sibling' (alternative move), and 'child' (next move) trees, and a number indicating the depth of the root node of the tree (its level).

`newGameTreeType()` → `GameTreeType`

Used to create an empty tree with no sub-trees, and undefined grids and level.

`newGameTreeType(GridType,LevelNumberType)` → `GameTreeType`

Used to create a a game-tree at the given level with an empty parent and the given grid, returning the variable. *Calls* `setGrid()` *and* `setLevel()`.

`newGameTreeType(GridType,LevelNumberType,GameTreeType)` → `GameTreeType`

Used to create a game-tree at the given level with the given parent and the given grid), returning the variable. *Calls* `setGrid()`, `setLevel()` *and* `setParent()`.

`isEmpty(GameTreeType)` → `Boolean`

Used to test whether the given game-tree is the empty tree; `true` is returned if it is, `false` is returned otherwise.

`setGrid(GameTreeType,GridType)` → `GameTreeType`

Used to set the grid component of the root node to the given grid, returning the resultant tree. *Calls* `isEmpty()`.

`getGrid(GameTreeType)` → `GridType`

Used to return the grid component of the root node of the given game-tree. *Calls* `isEmpty()`.

`setParent(GameTreeType,GameTreeType)` → `GameTreeType`

Used to set the parent of the root node as the given game-tree, returning the resultant tree. *Calls* `isEmpty()`.

`getParent(GameTreeType)` → `GameTreeType`

Used to return the parent component of the root node of the given game-tree. *Calls* `isEmpty()`.

`setChild(GameTreeType,GameTreeType)` → `GameTreeType`

Used to set the child component of the root node of the given game-tree to be the given sub-tree. The updated tree is returned. *Calls* `isEmpty()`.

`getChild(GameTreeType)` → `GameTreeType`

Used to return the child sub-tree of the root node of the given game-tree. *Calls* `isEmpty()`.

`setSibling(GameTreeType,GameTreeType)` → `GameTreeType`

Used to set the sibling component of the root node of the given game-tree to be the given sub-tree. The updated tree is returned. *Calls* `isEmpty()`.

`getSibling(GameTreeType)` → `GameTreeType`

Used to return the sibling sub-tree of the root node of the given game-tree. *Calls* `isEmpty()`.

`setLevel(GameTreeType,LevelNumberType)` → `GameTreeType`

Used to set the level number component (i.e. label) of the root node of the given game-tree to be the given value. The updated tree is returned. *Calls* `isEmpty()`.

`getLevel(GameTreeType)` → `LevelNumberType`

Used to return the level number (i.e. label) of the root node of the given game-tree. *Calls* `isEmpty()`.

`buildGame(GameTreeType,GridType,PlayerType,PlayerType)` → `GameTreeType`

Used to generate a game-tree of the specified depth with moves alternately generated for the first and second player value on the specified starting grid. *Calls* `generateLevel()`, `isEmpty()`, `getLevel()`, `LevelNumberType.equals()` *and* `buildGame()`.

`generateLevel(GameTreeType,PlayerType,PlayerType)` → `GameTreeType`

Used to add an additional level to the specified game-tree with the level number incremented from the given game-tree with the next move to be made by the first player value (if their

turn, or the second player otherwise). *Calls* `getLevel()`, `LevelNumberType.increment()`, `getGrid()`, `GridType.getDimension()`, `LocationType.newLocationType()`, `IndexType.newIndexType()`, `IndexType.lessThan()`, `IndexType.increment()`. `GridType.gameOver()`, `newGameTreeType()`, `GridType.clone()`, `evaluateGrid()`, `GridType.setWorth()` *and* `GridType.occupySquare()`.

`chooseBest(GameTreeType)` → `WorthType`

Used to search the given game-tree for the best move. The value of the grid containing the best move is returned. *Calls* `getGrid()`, `GridType.getWorth()`, `getSibling()`, `isEmpty()`, `getLevel()` *and* `WorthType.lessThan()`.

`findBest(GameTreeType,WorthType)` → `GameTreeType`

Used to locate the best move from an existing game-tree. The game-tree with this best move as its root node is returned. *Calls* `getChild()`, `getGrid()`, `GridType.getWorth()`, `getSibling()`, `isEmpty()` *and* `WorthType.equals()`.

`findMove(GameTreeType,GridType)` → `GameTreeType`

Used to locate the given grid within the existing game-tree so that the user's move may be found within the game-tree. The game-tree with this grid as its root node is returned. *Calls* `getChild()`, `getGrid()`, `GridType.equals()`, `getSibling()` *and* `isEmpty()`.

`adjustLevel(GameTreeType)` → `GameTreeType`

Used to adjust the level numbers within the tree so that the root node has level 0, its child nodes have level 1, and so on. *Calls* `getChild()`, `getLevelNumber()`, `LevelNumberType.decrement()`, `setLevelNumber()`, `getSibling()` *and* `isEmpty()`.

`Connect4Type` defines the application of the preceding ADTs to the game of Connect-4. It consists of input and output mechanisms, a game-tree, two players (one computer and one human), and an indication of the desired number of levels for the game-tree (the look-ahead horizon).

`newConnect4Type(InputType,DisplayType)` → `Connect4Type`

Used to create a Connect-4 game with given input and output mechanisms, and a default game-tree, players, and look-ahead horizon governed by the user. Calls `setInput()`, `setDisplay()`, `whoStarts()`, `whatSizeGrid()`, `howSmartIsComputer()`, `newGameTreeType()`, `newGridType()`, `GameTreeType.setGrid()`, `newLevelNumberType()` *and* `newPlayerType()`.

`setInput(Connect4Type,InputType)` → `Connect4Type`

Used to set the input mechanism to the given value, returning the updated game.

`getInput(Connect4Type)` → `InputType`

Used to return the input mechanism component of the given game.

`setDisplay(Connect4Type,DisplayType)` → `Connect4Type`

Used to set the display mechanism to the given value, returning the updated game.

`getDisplay(Connect4Type) → DisplayType`

Used to return the component of the given game.

`setGameTree(Connect4Type,GameTreeType) → Connect4Type`

Used to set the game-tree to the given value, returning the updated game.

`getGameTree(Connect4Type) → GameTreeType`

Used to return the game-tree component of the given game.

`setPlayer1(Connect4Type,PlayerType) → Connect4Type`

Used to set player 1 to the given value, returning the updated game.

`getPlayer1(Connect4Type) → PlayerType`

Used to return the player 1 component of the given game.

`setPlayer2(Connect4Type,PlayerType) → Connect4Type`

Used to set player 2 to the given value, returning the updated game.

`getPlayer2(Connect4Type) → PlayerType`

Used to return the player 2 component of the given game.

`setLevelNumber(Connect4Type,LevelNumberType) → Connect4Type`

Used to set the level number to the given value, returning the updated game.

`getLevelNumber(Connect4Type) → LevelNumberType`

Used to return the level number component of the given game.

`whoStarts(Connect4Type) → PlayerType`

Used to determine which player makes the first move.

`whatSizeGrid(Connect4Type) → DimensionType`

Used to determine what sized grid the game should be played on.

`howSmartIsComputer(Connect4Type) → LevelNumberType`

Used to determine the look-ahead horizon for the computer's moves.

`whatIsHumanMove(Connect4Type) → LocationType`

Used to determine the user's desired move.

`movePossible(Connect4Type,LocationType) → Boolean`

Used to check if the user's desired move is possible. *Calls* `getGameTree()`, `getGrid()`, `validMove()`, *and* `squareOccupied()`.

`makeHumanMove(Connect4Type,LocationType) → Connect4Type`

Used to enter the user's given (valid) move onto the grid within the game-tree component of the game. *Calls* `getGameTree()`, `getGrid()`, `PlayerType.getSymbol()`, `occupySquare()`, `setGrid()` *and* `setGameTree()`.

`makeComputerMove(Connect4Type) → Connect4Type`

Used to determine and select the computer's move using the game-tree component of the game. *Calls* `getGameTree()`, `buildGame()`, `chooseBest()`, `findBest()` *and* `setGameTree()`.

`gameOver(Connect4Type) → Boolean`

Used to determine whether the game has finished because of a win, loss, or draw; returns `true` if so and `false` otherwise. *Calls* `getGameTree()`, `getGrid()` *and* `GridType.gameOver()`.

`play(Connect4Type) → Connect4Type`

Used to play Connect-4 as outlined in the following section. *Calls* `Connect4.gameOver()`, `showConnect4()`,

whatIsHumanMove(), makeHumanMove(), movePossible(), getGameTree(), getChild(), GameTreeType.isEmpty(), findMove(), buildGame(), chooseBest(), findBest(), adjustLevel(), setGameTree(), getGrid(), draw(), win() and showConnect4().

`showConnect4(Connect4Type)` → `PlayerType`

Used to provide a screen representation of the grid component of the game on the display component of the game. *Calls* `getGameTree()`, `getGrid()`, `getDisplay()` *and* `showGrid()`.

Algorithm Outline

Assuming the game is to be played on a 4 by 4 grid, a screen, an input device, and a game-tree consisting of a single empty grid of the required dimension is generated (*via* `newConnect4Type()`). The user may indicate a different grid dimension (*via* `whatSizeGrid()`), whether or not they wish to go first (*via* `whoStarts()`), and how many moves they would like the computer to 'look ahead' (*via* `howSmartIsComputer()`).

The following cycle (from `play()`) is then repeated until the game is over (detected *via* `Connect4Type.gameOver()`):

- the current grid is displayed on the screen (*via* `showConnect4()`)
- when it is the user's turn, the user indicates a move (*via* `whatIsHumanMove()`) which is added to the grid (via `makeHumanMove()`) after it is ascertained that the move is valid (*via* `movePossible()`). If the current game-tree possesses subsequent levels (*via* `getGameTree()`, `getChild()` and `GameTreeType.isEmpty()`), then the child node with the same grid as that indicated is located and is made the root of the tree (using `findMove()`) and all other subtrees are discarded;
- when it is the computer's turn — and provided the game is not over (again detected *via* `Connect4Type.gameOver()`) — then the game-tree is expanded (*via* `buildGame()`) to contain as many alternating levels of possible grids as specified. This tree is then walked over (using `chooseBest()`) to discover the best achievable outcome and then the corresponding grid in the game-tree is located (using `findBest()` which sets the root of the tree to that grid discarding all other sub-trees except those beneath this node). This grid contains the computer's move and is the new state of the game. The resulting game-tree has all the levels adjusted to ensure the count of levels from the root node is accurate (using `adjustLevel()`) and is placed back into the game (*via* `setGameTree()`) and the cycle is repeated.

When the game is over and the above cycle ends, there is little left to do except say who won. If it is a draw (discerned *via* `getGameTree()`, `getGrid()` and `draw()`) this information is displayed on the screen, otherwise the winner (discerned *via* `getGameTree()`, `getGrid()` and `win()`) is displayed. The results are shown using `showConnect4()`.