Università di Bologna

Elaborato di Programmazione ad oggetti

Java software to set and play different polyomino based puzzles

Author:

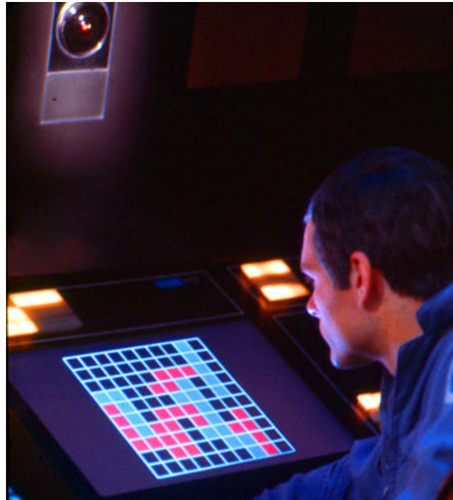
Di Luigi William

18 maggio 2014

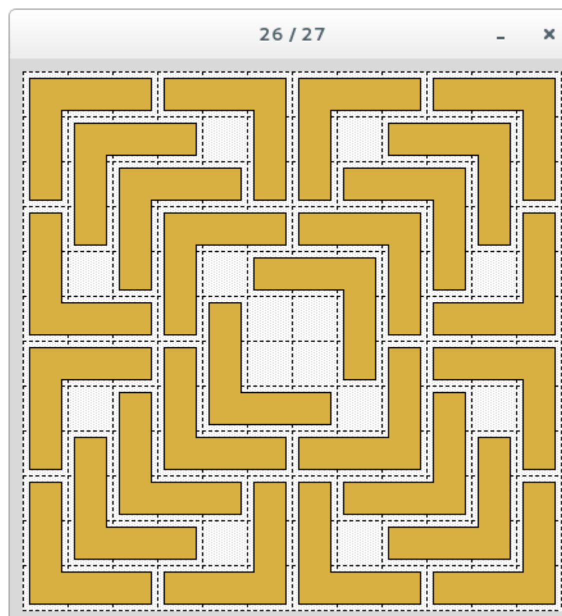
1 Introduction

What is a polyomino based puzzle?

Polyomino based puzzles are just like normal puzzles, but they are played using polyominoes. They've been around since at least 1907. Sir Arthur C. Clarke was fascinated by them. They came close to being featured in the movie *2001: A Space Odyssey* (1968) in the form of a variation of the board game "Pentominoes", but were replaced at the last minute by chess.



Like many puzzles in recreational mathematics, polyominoes raise many combinatorial problems. The first one I encountered was the problem of determining the maximum number of V-pentominoes (or the densest packing) covering the cells of the square $N \times N$. I had the opportunity to tackle for some time a real wooden version of this puzzle where $N = 12$ (for such a board it's possible to pack as much as 27 pieces)[1]. Later I decided to clone this as a Python game[2] in order to try to solve it. Obviously, I didn't manage to do so. Here is what that little program looked like:



[1] `http://oeis.org/A214294`
[2] `https://github.com/wil93/pentomino-cover-game`

Later on, I decided to actually build it with my uncle's help:

Once you empty the box it's funny to see people struggling to put all 27 pieces back in. The complexity of this puzzle comes from the fact that, for $N = 12$, the solution is unique under rotation and reflection.

Another somewhat interesting problem regarding pentominoes (the one featured in the picture about *2001: A Space Odyssey*) is called, without much imagination, "Pentominoes". It is a nim-like board game where two or more players alternate placing a pentomino on the board, avoiding overlapping any piece placed earlier. The one that can't make a move loses.

The two-player version of "Pentominoes" has been weakly solved in 1996 by Hilarie Orman. It was proved to be a first player win by examining around 22 billion board positions[3].

Why did you rewrite the existing software?

The Python clone I wrote for the first puzzle turned out to be just fine for what I wanted to achieve. The problem is that, since the program wasn't designed with OOP principles in mind, if now I want to play a slightly different pentomino based puzzle then I would need to heavily change the codebase.

In particular, one could easily identify the following design flaws:

No modularity

The program is literally just a single file containing lots of routines: this reflects badly on code maintenance. Some time ago I was demonstrating that puzzle to some high school students and I needed to quickly add a temporary "cheat code" that automatically solved the puzzle. Not so difficult but, needless to say, with a modular codebase I would have probably done it faster.

Poor code reusing

For example, in four different routines (*checkFree*, *doPaint*, *setBusy* and *addPentomino*) you can find this exact piece of code:

```
for i in xrange(self.numPieces):
    new_x = x + self.pos[self.rotation][i][0]
    new_y = y + self.pos[self.rotation][i][1]
    # Do stuff with (new_x, new_y) ...
```

[3] Hilarie K. Orman. Pentominoes: A First Player Win (Pdf).

Which could be rewritten in a much nicer way using iterators. Let's not talk about the heavy code duplication found in the *drawCell* routine.

No encapsulation
Every routine has access to the full game state. For example: the `checkFree` routine, which only purpose is to check whether a cell may or not be a valid spot to place an entire pentomino, has the faculty to change the window title.

In addition to plain design flaws, another motivation for an OOP rewrite of the software could be to allow multiplayer gaming (for example, in "Pentominoes"), possibly through a local network.

2 Overall analysis

We want the new software to make it easy to code different types of games. Strictly speaking, we want a Java framework for implementing polyomino based puzzles. This framework should be written as much modular as possible, in order to allow further customization. In particular, we detail the following requirements for such a framework:

- It shall be easy to code different polyominoes, thus each one of them shall specify its shape, color and name in a simple ASCII text file.
- It shall be possible to customize the shape and behavior of the grid.
- The game-player interaction shall be mediated by an entity which shall support, if needed, more than one player.

In order to demonstrate the framework, I decided to support with the minimal amount of effort per game every one of the following games (upon selection from a dedicated "main menu"):

◇ **Twenty-seven**
This is the first puzzle described in the introduction. It is characterized by:
  - The only available polyomino is the V-pentomino (27 copies of it).
  - The game is played on a rectangular 12×12 grid.
  - There is only one (local) player.

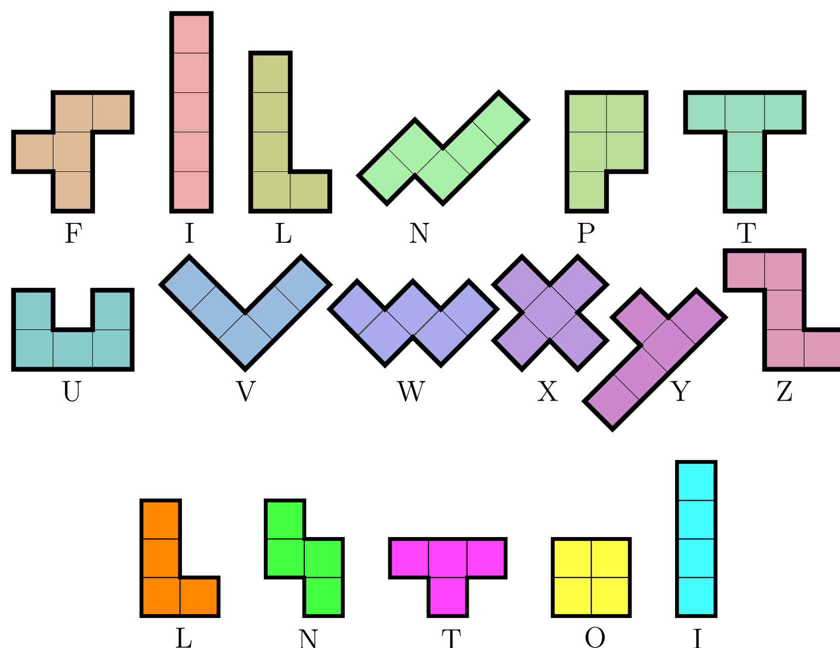
◇ **Crucitetris**
This is similar to the first puzzle, but instead of a "best fit", the user has to find a perfect fit, given that some of the cells are not usable. It is characterized by:
  - A subset of all tetrominoes and pentominoes are available (in a limited quantity).
  - The game is played on a rectangular grid, in which some of the cells are not usable.
  - There is only one (local) player.

◇ **Pentominoes**
As described in the introduction, this is a two player game. It is characterized by:
  - All pentominoes are available (not limited in quantity).
  - The game is played on a rectangular 8×8 grid.
  - There are two players, one of which can be remotely connected via LAN.

Also, I decided to support every pentomino and tetromino. If another polyomino type is needed, it will have to be prepared as described in the next section.

3 Implementation choices

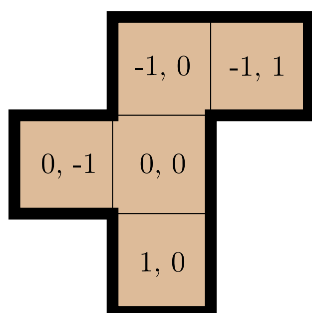
The naming scheme I followed for polyominoes is the one that pairs each piece with the letter that most closely resemble it[4], the same I did for tetrominoes:



Each polyomino is identified by two files. For example, let's consider the F-pentomino. Its descriptor files are `F.png` which is a visual representation of the piece, and `F.txt` which defines its actual structure. In general, the `*.txt` file is a simple ASCII text file consisting of $3+N$ lines, where N is the polyomino's area (that is, the number of blocks of which it is made):

1. The first line is the polyomino's name. Any ASCII line will do.
2. The second line is the polyomino's color, any CSS compliant[5] color string will do.
3. The third line is the polyomino's area N, it must be a positive integer.
4. Each one of the next N lines contains two space-separated non-negative integers i and j. These are the coordinates of one of the N blocks of which the polyomino, centered in $(0,0)$, is made.

Here is what the F-pentomino looks like:



Here is what there's inside `F.txt`:

```
F
#ddbb99
5
-1 0
-1 1
0 -1
0 0
1 0
```

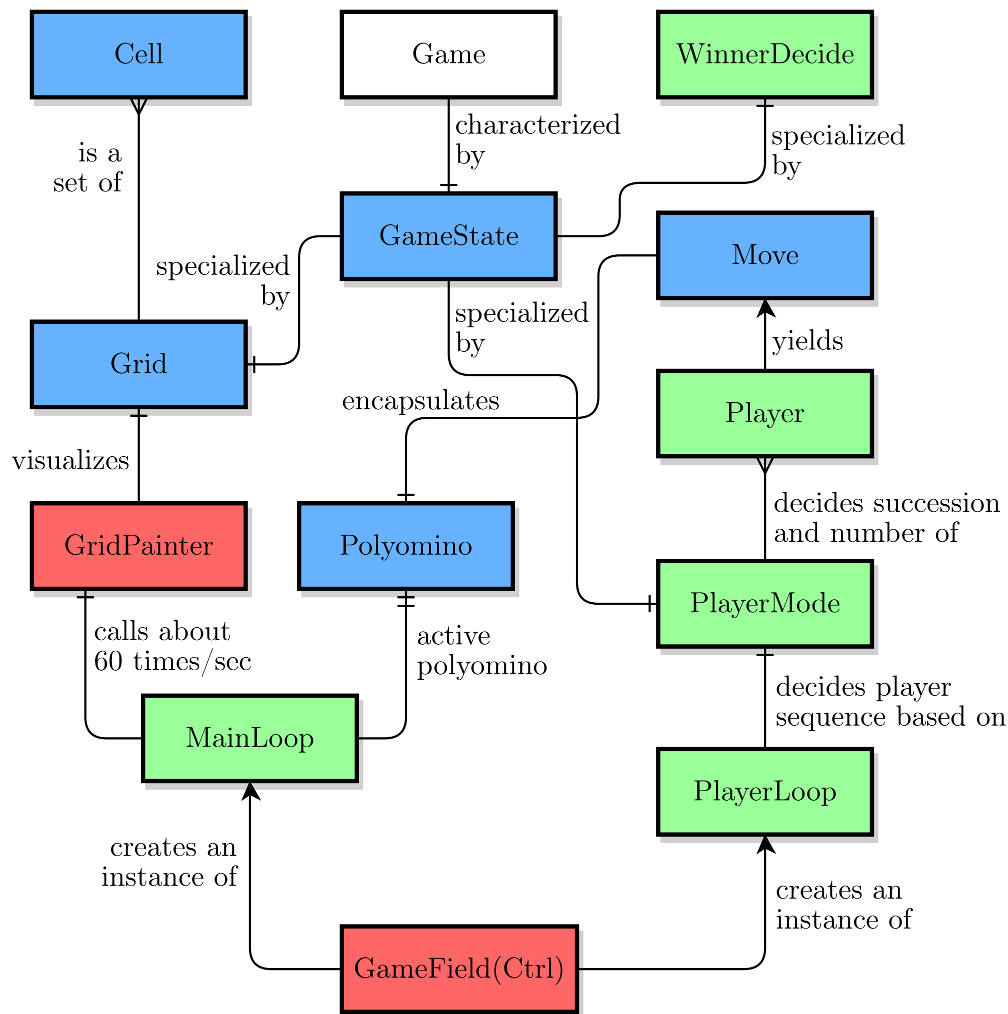
[4] (Gardner 1960, Golomb 1995)
[5] `http://www.w3.org/TR/css3-color/`

4 Architectural design

The main design concept I chose to follow was to encode every game type in a specific instance of a "GameState" class. This class would hold all of the information needed, like: polyominoes available, grid type and so on. Also, the "view" of the game (which is made basically of *GameField*, *GameFieldCtrl*, *GridPainter*) is decoupled from the "model" (which is the *Grid*). I didn't make a sharp distinction between MVC components in the project setup, though I might do it in the future.

4.1 Overall architecture

Follows an abstract representation of the overall software architecture. Red, blue and green boxes stand for "view", "model" and "controller" respectively.



4.2 Design patterns adopted

Whenever possible, the following software design patterns have been used:

⊙ **Strategy pattern**
The *GameState* class is "the strategy" by which any game type is characterized. It defines what grid, polyomino set and player mode will be used. It also encapsulates *WinnerDecide*, another class which follows the strategy pattern, used to determine if the current user has won the match.

⊙ **Template method pattern**
The *AbstractGrid* implements most of Grid interface's methods by calling appropriately a few abstract methods (the **template methods**). When a concrete Grid implementation will be needed, the developer will extend this class and will only have to implement those missing methods.

⊙ **Decorator pattern**
The *ObstacledGrid* class is a Grid decorator (its constructor takes a Grid instance, which will be internally referenced). It adds new supported methods (in this case, just the *addObstacle* method) while every other call to Grid's methods will be "redirected" to the internally referenced Grid. Another example of decorator is the *BrightAnimatedColor* class, which decorates *javafx.paint.Color*. This class internally stores the "decoration parameters" only and, when a Color has to be decorated, it must be passed as a parameter to its *animateColor* method.

⊙ **Observer pattern**
The Grid interface is "observable", since it allows any instance of *GridObserver* to be registered as one of its observers. Whenever a change is made to the grid (i.e. a polyomino is removed) an event is emitted to all observers. The *GameFieldCtrl* class implements *GridObserver*, thus it is an "observer".

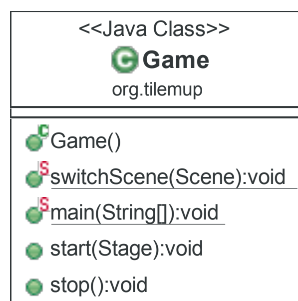
⊙ **Singleton pattern**
I don't really like this pattern, as it tends to introduce global state and causes classes to be difficult to test in isolation (as it is necessary to first initialize the singleton). However, I did use it for the *GameContainer* class, which keeps references to the main loop, player loop, edit/undo buttons and so on. I plan to reduce the content of this class (or even to completely delete it).

4.3 Packages' organization

The project is composed of different packages. Each package wraps up related classes (i.e. the grid interface and all concrete grids are in the same package). Follows a description for each package.

4.3.1 org.tilemup

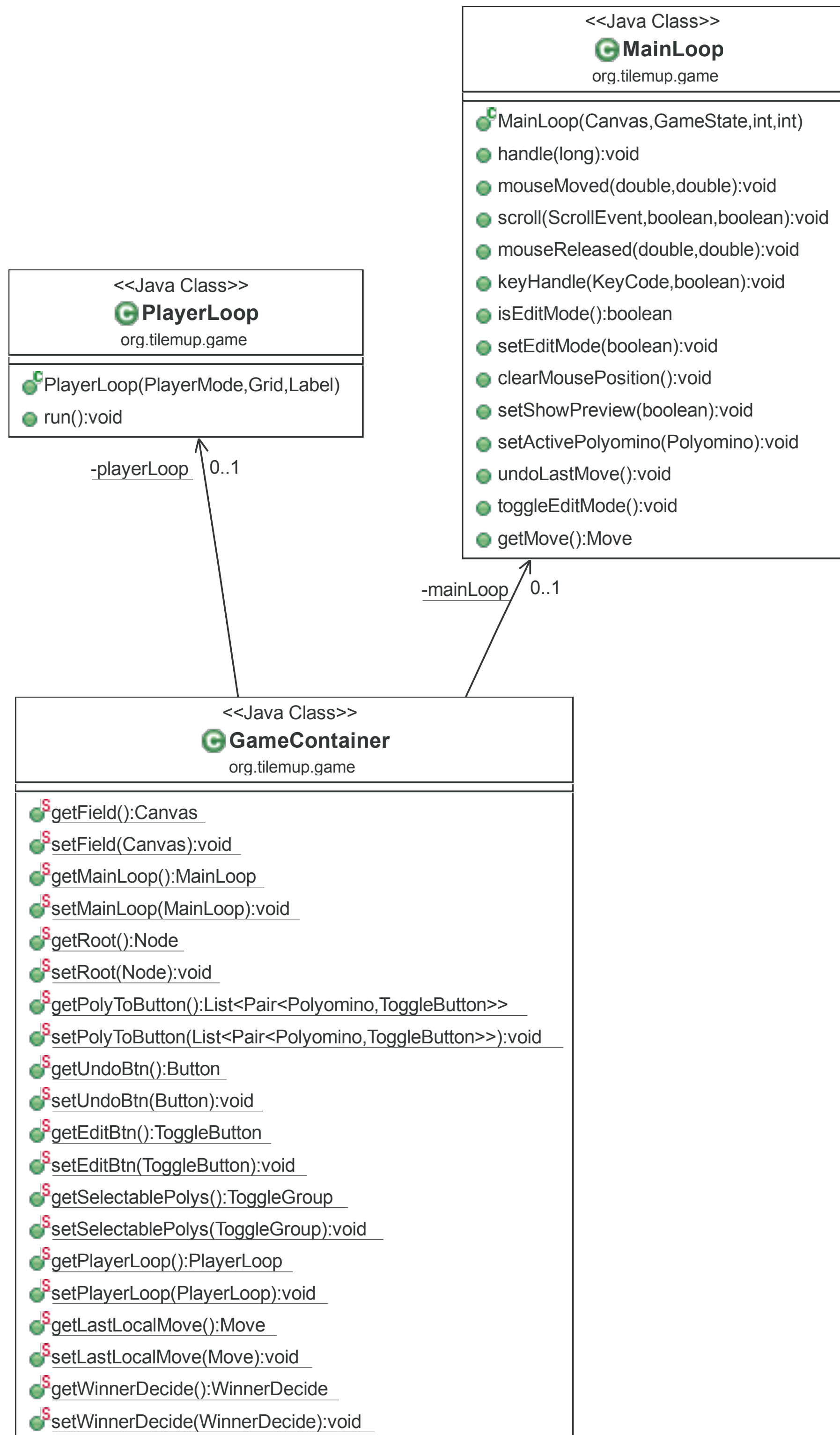
The main package. It only contains the *Game* class, from which the application is instantiated.



Game
This class is the entry point of the game. It extends the *javafx.application.Application* class. Its *switchScene* method is used to replace the current scene with a given new one.

4.3.2 org.tilemup.game

This package contains basic constructs such as *MainLoop* and *PlayerLoop*, which together form the game loop.



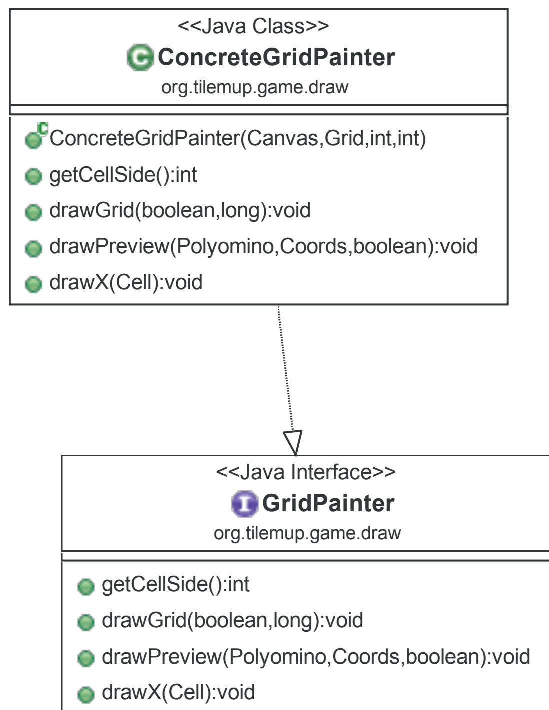
MainLoop
: This class is responsible for user interaction with the puzzle: it handles the drawing of the grid on the screen. It extends the *javafx.animation.animationTimer* class.

PlayerLoop
: It's responsible for the alternation of players during a match's life cycle.

GameContainer
: It's a **singleton** whose job is to keep references to useful properties. Some of these properties may be moved away, in the future.

4.3.3 org.tilemup.game.draw

It contains the *ConcreteGridPainter* class for drawing grid cells and polyominoes to the screen. In the future, if another style of drawing will be needed, it will suffice to make another class implement the *GridPainter* interface.

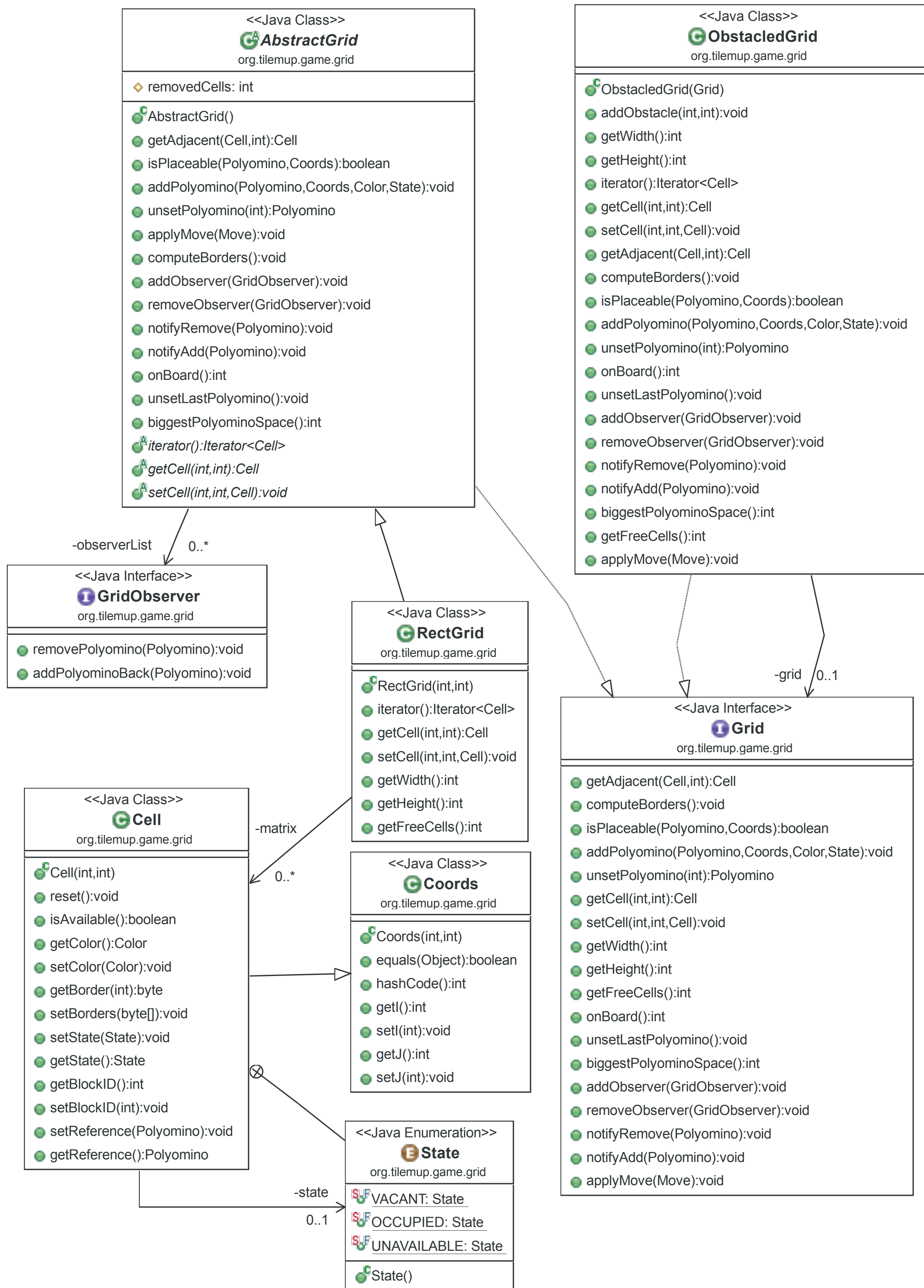


GridPainter
: This interface defines the contract a class must fulfill in order to be a candidate for drawing the game field to the screen.

ConcreteGridPainter
: It's the "reference" *GridPainter* implementation.

4.3.4 org.tilemup.game.grid

This package abstracts the concept of grid by defining the *Grid* interface, which says that a grid must respond to *getWidth()*, *getCell(i, j)* and things like that. Also it contains all "concrete" grids (for now, only *RectGrid*) as well as a grid decorator called *ObstacledGrid* which enhances any *Grid* by adding "obstacles", which are just unusable grid cells.


<<Java Class>>
AbstractGrid
org.tilemup.game.grid
removedCells: int
AbstractGrid()
getAdjacent(Cell,int):Cell
isPlaceable(Polyomino,Coords):boolean
addPolyomino(Polyomino,Coords,Color,State):void
unsetPolyomino(int):Polyomino
applyMove(Move):void
computeBorders():void
addObserver(GridObserver):void
removeObserver(GridObserver):void
notifyRemove(Polyomino):void
notifyAdd(Polyomino):void
onBoard():int
unsetLastPolyomino():void
biggestPolyominoSpace():int
iterator():Iterator<Cell>
getCell(int,int):Cell
setCell(int,int,Cell):void
<<Java Class>>
ObstacledGrid
org.tilemup.game.grid
ObstacledGrid(Grid)
addObstacle(int,int):void
getWidth():int
getHeight():int
iterator():Iterator<Cell>
getCell(int,int):Cell
setCell(int,int,Cell):void
getAdjacent(Cell,int):Cell
computeBorders():void
isPlaceable(Polyomino,Coords):boolean
addPolyomino(Polyomino,Coords,Color,State):void
unsetPolyomino(int):Polyomino
onBoard():int
unsetLastPolyomino():void
addObserver(GridObserver):void
removeObserver(GridObserver):void
notifyRemove(Polyomino):void
notifyAdd(Polyomino):void
biggestPolyominoSpace():int
getFreeCells():int
applyMove(Move):void
-observerList
0..*
<<Java Interface>>
GridObserver
org.tilemup.game.grid
removePolyomino(Polyomino):void
addPolyominoBack(Polyomino):void
<<Java Class>>
RectGrid
org.tilemup.game.grid
RectGrid(int,int)
iterator():Iterator<Cell>
getCell(int,int):Cell
setCell(int,int,Cell):void
getWidth():int
getHeight():int
getFreeCells():int
-grid
0..1
<<Java Interface>>
Grid
org.tilemup.game.grid
getAdjacent(Cell,int):Cell
computeBorders():void
isPlaceable(Polyomino,Coords):boolean
addPolyomino(Polyomino,Coords,Color,State):void
unsetPolyomino(int):Polyomino
getCell(int,int):Cell
setCell(int,int,Cell):void
getWidth():int
getHeight():int
getFreeCells():int
onBoard():int
unsetLastPolyomino():void
biggestPolyominoSpace():int
addObserver(GridObserver):void
removeObserver(GridObserver):void
notifyRemove(Polyomino):void
notifyAdd(Polyomino):void
applyMove(Move):void
<<Java Class>>
Cell
org.tilemup.game.grid
Cell(int,int)
reset():void
isAvailable():boolean
getColor():Color
setColor(Color):void
getBorder(int):byte
setBorders(byte[]):void
setState(State):void
getState():State
getBlockID():int
setBlockID(int):void
setReference(Polyomino):void
getReference():Polyomino
-matrix
0..*
<<Java Class>>
Coords
org.tilemup.game.grid
Coords(int,int)
equals(Object):boolean
hashCode():int
getI():int
setI(int):void
getJ():int
setJ(int):void
<<Java Enumeration>>
State
org.tilemup.game.grid
VACANT: State
OCCUPIED: State
UNAVAILABLE: State
State()
-state
0..1

Grid

This interface defines the contract a class must fulfill in order to be a grid.

AbstractGrid

This abstract class follows the **template method** pattern, so it defines the grid's behavior depending on a few abstract methods: *iterator*, *getCell*, *setCell*. One could decide to specialize this class, for example, by using a hash map to store the grid, and just these three methods should be overridden. The default implementation, though, uses just a simple matrix.

RectGrid

As outlined, this is the default grid implementation. It stores cells using a simple matrix. As the name hints, it allows to create any (not too big) rectangular grid.

Cell

Represents a single grid cell. It has: state, color, borders (that can be thin or thick), a reference to the polyomino which covers it (or null if there isn't any).

Cell.State

Represents a cell's state, which can be vacant, occupied or unavailable.

Coords

Is a very basic class, essentially a pair of integers.

ObstacledGrid

This class follows the **decorator** pattern, since it decorates any *Grid* by adding "obstacles" to it. To add an obstacle, the *addObstacle* method shall be called (every other call will be dispatched to the underlying grid).

4.3.5 org.tilemup.game.players

In this package there are three main concepts wrapped: the *Player*, which must always respond to *getMove*; the *Move*, returned by the player; the *PlayerMode*, which is how the *PlayerLoop* interacts with the player(s). Among other things, here is implemented the logic for LAN connection between players.

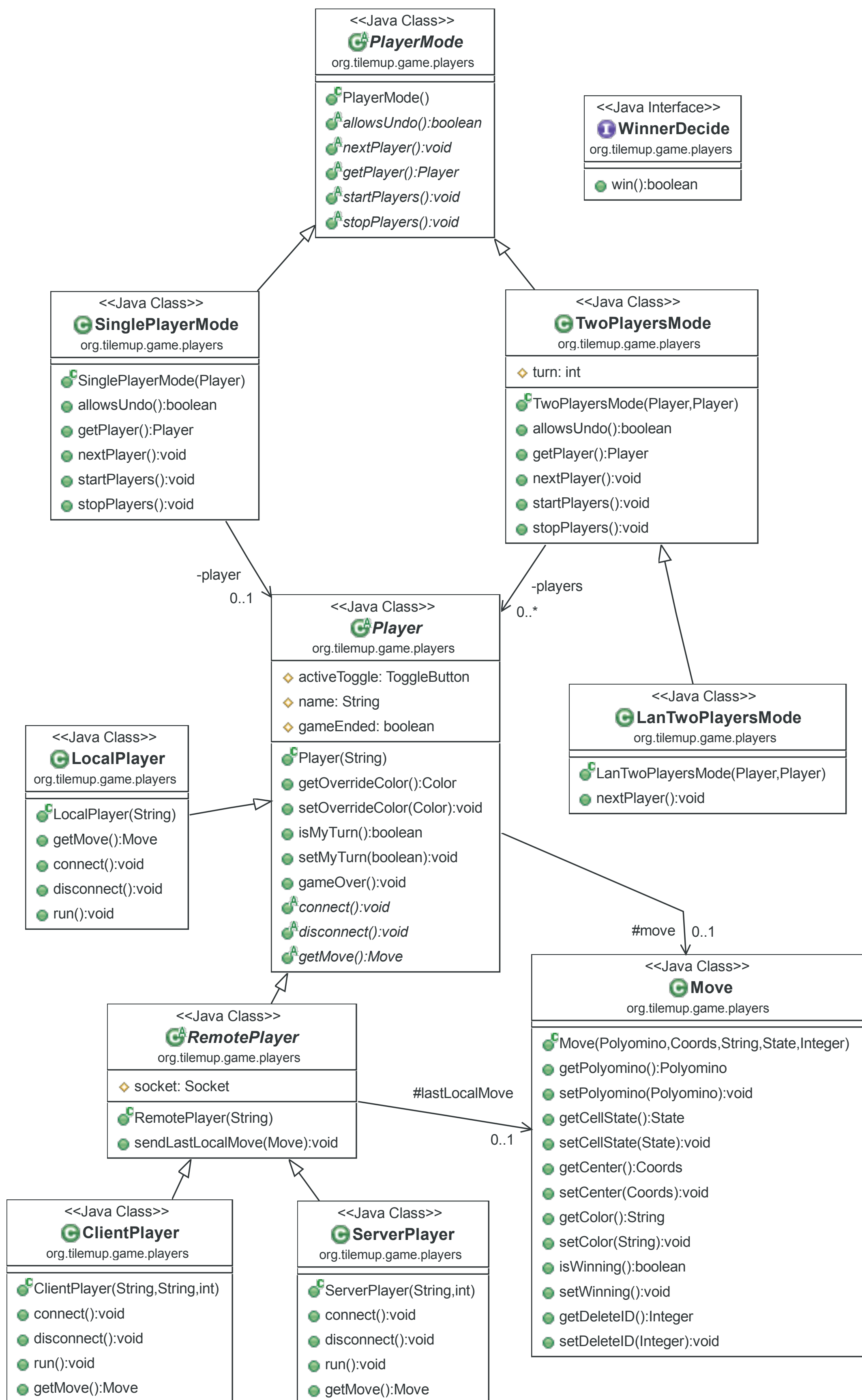
WinnerDecide

When we check if the user has won, we call the *win* method on an instance of this interface.

PlayerMode

This abstract class defines a player mode, which decides whether the player is allowed to use "unset the last polyomino placed" as a move. Also, it shall respond to *getPlayer* and *nextPlayer*, in order for the player loop to work. Three specialized versions of this class are provided by default:

- **SinglePlayerMode** — This class implements a single player mode, "undo" is allowed and *nextPlayer* does nothing.
- **TwoPlayersMode** — This class implements a two players mode, "undo" is disallowed. It is further specialized to:
  - **LanTwoPlayersMode** — This class, when *nextPlayer* is called, will handle the updating (and the dispatch to the other player) of *lastLocalMove*.



<<Java Class>>
PlayerMode
org.tilemup.game.players
PlayerMode()
allowsUndo():boolean
nextPlayer():void
getPlayer():Player
startPlayers():void
stopPlayers():void
<<Java Interface>>
WinnerDecide
org.tilemup.game.players
win():boolean
<<Java Class>>
SinglePlayerMode
org.tilemup.game.players
SinglePlayerMode(Player)
allowsUndo():boolean
getPlayer():Player
nextPlayer():void
startPlayers():void
stopPlayers():void
<<Java Class>>
TwoPlayersMode
org.tilemup.game.players
turn: int
TwoPlayersMode(Player,Player)
allowsUndo():boolean
getPlayer():Player
nextPlayer():void
startPlayers():void
stopPlayers():void
-player
0..1
-players
0..*
<<Java Class>>
Player
org.tilemup.game.players
activeToggle: ToggleButton
name: String
gameEnded: boolean
Player(String)
getOverrideColor():Color
setOverrideColor(Color):void
isMyTurn():boolean
setMyTurn(boolean):void
gameOver():void
connect():void
disconnect():void
getMove():Move
<<Java Class>>
LanTwoPlayersMode
org.tilemup.game.players
LanTwoPlayersMode(Player,Player)
nextPlayer():void
<<Java Class>>
LocalPlayer
org.tilemup.game.players
LocalPlayer(String)
getMove():Move
connect():void
disconnect():void
run():void
#move
0..1
<<Java Class>>
Move
org.tilemup.game.players
Move(Polyomino,Coords,String,State,Integer)
getPolyomino():Polyomino
setPolyomino(Polyomino):void
getCellState():State
setCellState(State):void
getCenter():Coords
setCenter(Coords):void
getColor():String
setColor(String):void
isWinning():boolean
setWinning():void
getDeleteID():Integer
setDeleteID(Integer):void
<<Java Class>>
RemotePlayer
org.tilemup.game.players
socket: Socket
RemotePlayer(String)
sendLastLocalMove(Move):void
#lastLocalMove
0..1
<<Java Class>>
ClientPlayer
org.tilemup.game.players
ClientPlayer(String,String,int)
connect():void
disconnect():void
run():void
getMove():Move
<<Java Class>>
ServerPlayer
org.tilemup.game.players
ServerPlayer(String,int)
connect():void
disconnect():void
run():void
getMove():Move

Player

This abstract class defines a player, which is basically a thread which, when it's not his turn to move, waits. Being a thread, it extends *java.lang.Thread*. It shall respond to *connect* and *disconnect*. These implementations are provided by default:

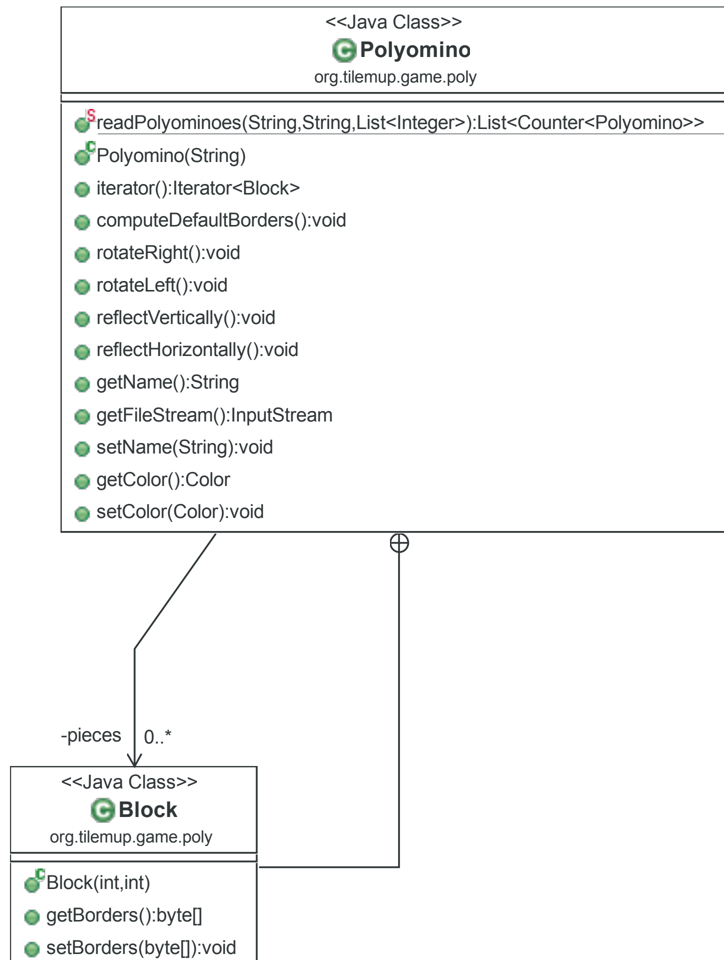
- **LocalPlayer** — This player "connects" by just binding mouse and key events the local game field (and "disconnects" by unbinding the same events).
- **RemotePlayer** — This abstract player is able to send a given move over the network. It's specialized to:
  - ◇ **ServerPlayer** — This player "connects" by creating a *ServerSocket* and waiting a client to connect. It "disconnects" by destroying the open socket.
  - ◇ **ClientPlayer** — This player "connects" to a server by creating a *Socket*; it "disconnects" by destroying it.

Move

This class "serializes" a player's move, in order to send it over the network (if needed). Every *Grid* shall respond to an *applyMove* method which takes a *Move* as parameter.

4.3.6 org.tilemup.game.poly

This package, in the *Polyomino* class, defines what a polyomino actually is. Also, a static method is provided for reading polyominoes from ASCII text files.



Polyomino

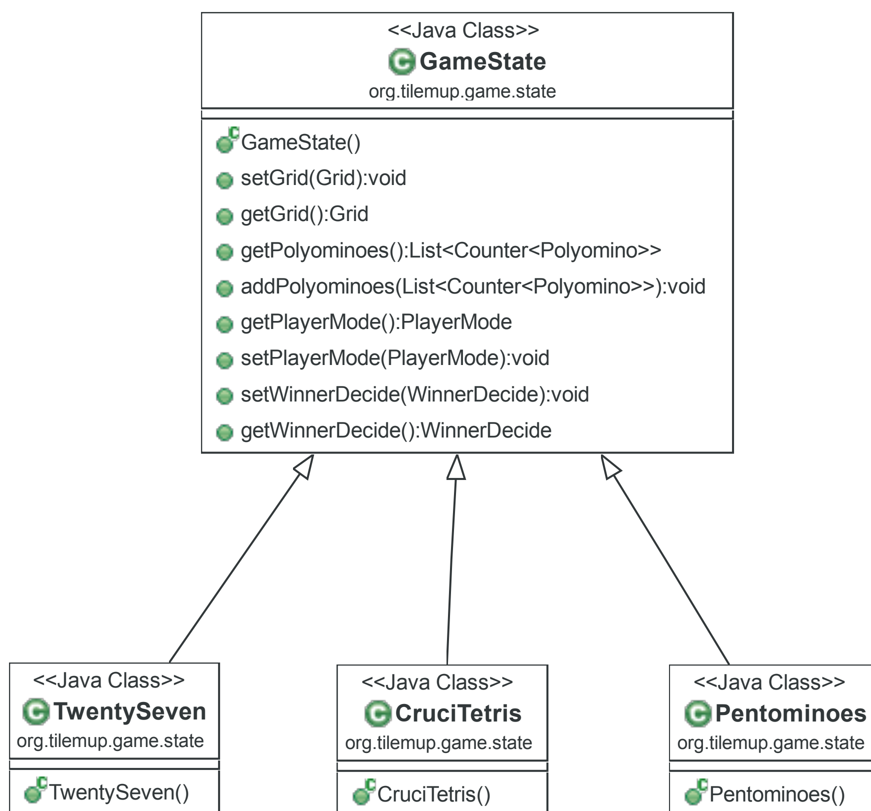
This class lays out the structure and behavior of a polyomino. It also defines its entire logic (rotation, reflection). It implements the *Iterable* interface, in order to provide an easy way to visit every block of the polyomino (as hinted in the introductory chapter).

Polyomino.Block

This nested class represents a single block of the polyomino. It extends *Coords*.

4.3.7 org.tilemup.game.state

Defines the concept of *GameState*, which is just a class that encapsulates the structure of a certain game, in terms of: grid type, polyominoes available and player mode.



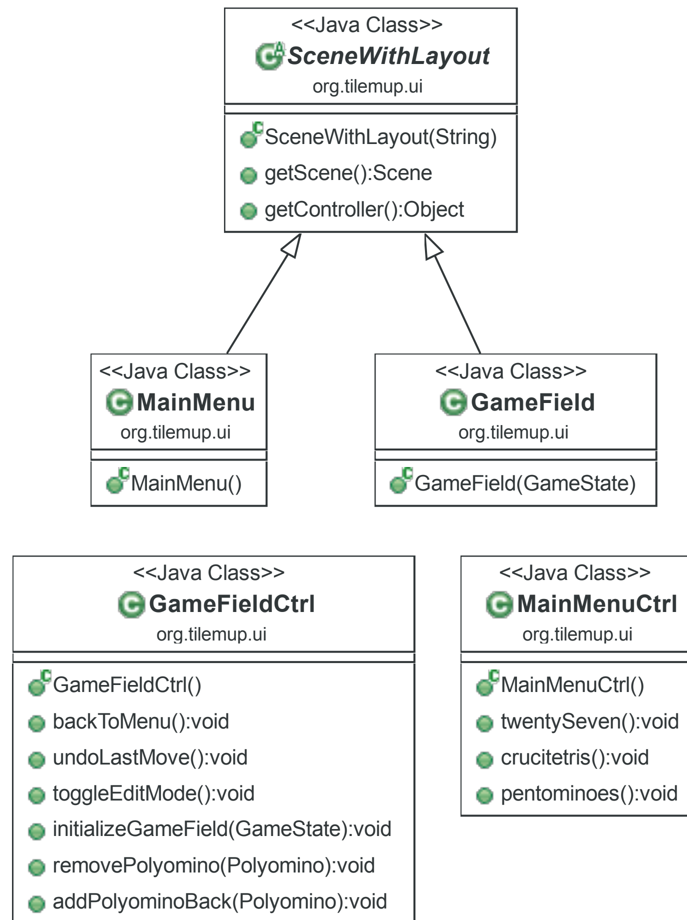
GameState

This class follows the **strategy** pattern, since it encapsulates all the properties of the game type that will be played, such as: grid type, polyominoes available, player mode, winner decision technique. The three specialized “strategies” provided, which have already been described in the “Overall analysis” chapter, use the following winner decision techniques:

- **TwentySeven** — When there are exactly $12^2 - 27 \cdot 5 = 9$ free cells left, the player won.
- **CruciTetris** — When there are 0 free cells left, the player won.
- **Pentominoes** — When the biggest fillable area on the grid drops below 5 squares, the current player has made the winning move.

4.3.8 org.tilemup.ui

Mostly UI-related code. The view for each "window" is loaded from resources which can be found in this package.



SceneWithLayout
This class extends *javafx.scene.Scene* and provides an easy way to load FXML layout files. When the *Node* gets loaded, the specified controller gets instantiated and injected. Then, both the scene and the scene's controller are made available via getter methods. Ideally, each window in the game should extend *SceneWithLayout*, passing its FXML layout file to the constructor.

MainMenu
Loads `MainMenu.fxml` (styled via `MainMenu.css`).

MainMenuCtrl
Defines the events bound to *MainMenu*'s buttons.

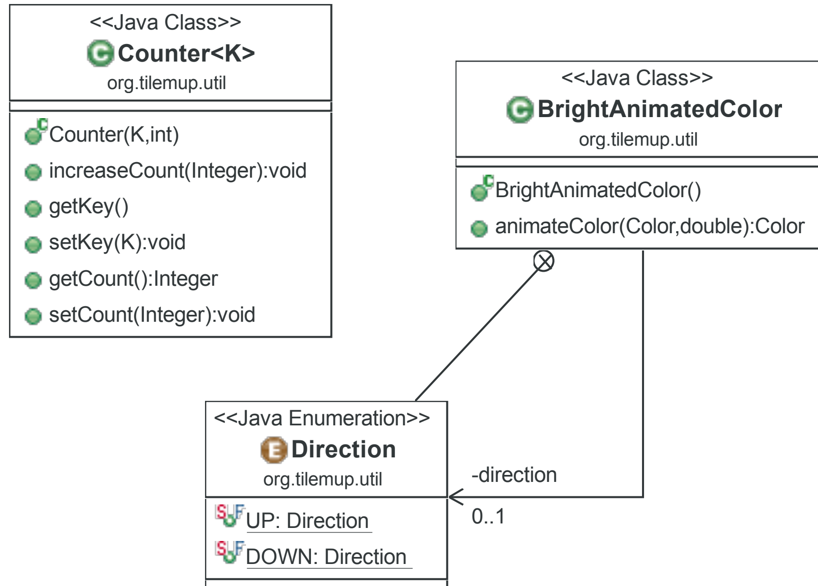
GameField
Loads `GameField.fxml` (styled via `GameField.css`).

GameFieldCtrl
Defines the events bound to *GameField*'s buttons, handles the right bar (where polyominos available are listed) resizing it appropriately. It also handles cursor changes, and starts the *MainLoop* and *PlayerLoop*. It follows the **observer** pattern, since it implements the *GridObserver* interface.

In that way, when the grid receives or loses a polyomino, *GameFieldCtrl* will be notified and will act accordingly (that is, it will change the value of that polyomino's counter).

4.3.9 org.tilemup.util

Some useful "general-purpose" classes.



BrightAnimatedColor
This class decorates a *javafx.paint.Color*, making it aware of the time elapsed. The *animateColor* method takes a color and the time elapsed since the last "decoration", and yields a brighter (or darker) version of the given color in such a way that, if called often enough, the color will appear to "vibrate".

Counter<K>
This is a generic class which associates an integer to some key of type *K*.

5 Conclusions

The work flow was quite straightforward, except some moving of classes between packages (just refactoring). In the future, I plan to improve *Tile 'em up!* and to clean up the code even more.

5.1 Implementation issues & External libraries

I used the *ControlsFX* open source library which provides high quality UI controls and other tools to complement the core JavaFX distribution. It can be seen in action when a match ends (the dialog control which announces the end of the game) and when a "Pentominoes" match starts (the LAN play selection dialog).

ControlsFX is one of the reasons I was forced to use the 1.8 version of the JDK (and the reason I decided to use Maven). In fact, it is only available for JavaFX 8 which, in turn, is only available for the 1.8 version of the JDK.

Other reasons that forced me to use that version include some JavaFX 2.2 bugs:

- The *MenuBar* (which I then removed anyway) flickered.
- *Stage.setResizable(false)* didn't work.
- JavaFX's startup was very slow on slow connections (if not connected, the startup time was negligible). This was caused by JavaFX 2.2 proxy auto-detecting.

5.2 What I've learned

Developing *Tile 'em up!* gave me a better grip of the Java language and of the OOP paradigm. Also, since I only knew git, it allowed me to learn Mercurial, thus deepening my knowledge of DVCSs.

I think I used a little more of the assigned 100 hours, because of the numerous bugs that brought me to switch from JavaFX 2.2 (JDK 1.7) to JavaFX 8 (JDK 1.8). Also, I wasn't used to Mercurial (and I *really* like git's staging area) so I found difficult to make "atomic" commits (so I did, sometimes, commit unrelated changes). Also, I've spent quite some of that time learning JavaFX from scratch.

5.3 Next release of *Tile 'em up!*

The next version of *Tile 'em up!* will likely feature the following:

- Keyboard shortcuts to select polyominos. Ambiguous keycodes (like "L" which can be both a tetromino and a pentomino) will be resolved by cycling through the matching polyominoes, thus, typing "L" one time will trigger the L-tetromino, typing it again will trigger the L-pentomino.
- Game levels will be supported. For example, there will be more than a single *CruciTetris* puzzle.
- Add the option to play *Pentominoes* against the computer. AI algorithms will be required.