Task 06: flexible 2048

Fabian Hoffmann
Simon Hintersonnleitner
David Kranewitter

Top-Level architecture

Data structures and responsibilities of classes

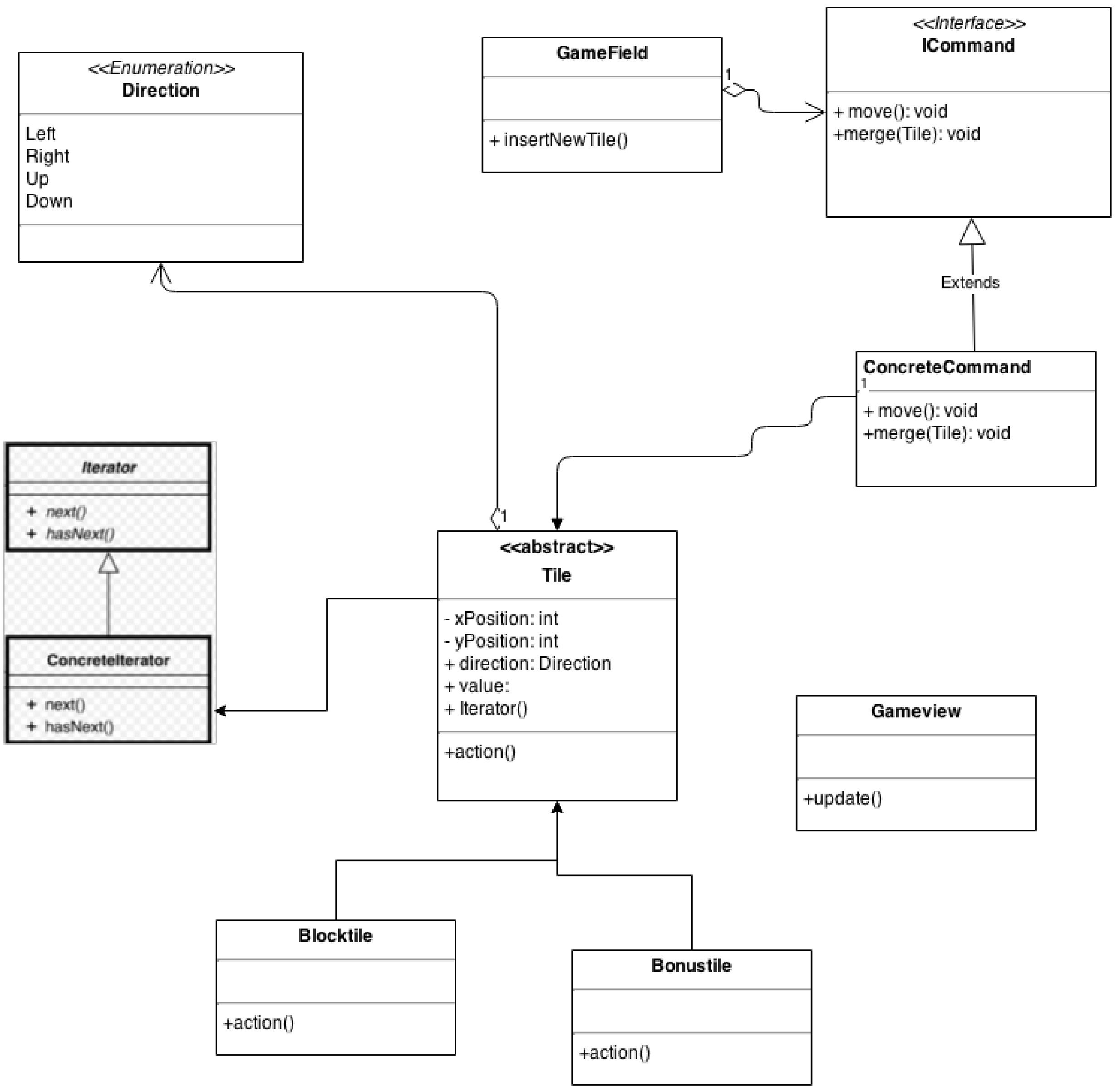
Playing Field: Two-Dimensional Array of Tiles

Tiles: Abstract Classes with two fields for the x and y position

special Tiles: inherit from “normal” Tiles and override move and merge method that they conform their Behavior

Game: contains a Playing Field, and a Score, and a Boolean flag whether game Over.

Directions: possible moving directions are stored in a enumeration



<<Enumeration>>
Direction
Left
Right
Up
Down
GameField
+ insertNewTile()
1
<<Interface>>
ICommand
+ move(): void
+merge(Tile): void
Extends
ConcreteCommand
1
+ move(): void
+merge(Tile): void
Iterator
+ next()
+ hasNext()
ConcreteIterator
+ next()
+ hasNext()
1
<<abstract>>
Tile
- xPosition: int
- yPosition: int
+ direction: Direction
+ value:
+ Iterator()
+action()
Gameview
+update()
Blocktile
+action()
Bonustile
+action()

Moving & Merging

Example
When move right: take all rows, delete all 0s, (from right to left) check each remaining element if left neighbor has same value, then multiply with itself and remove left neighbor. After all elements fill empty array spaces on left side with 0s.
Each move operation must be used on all tiles of the Gamefield. For this operation we use a for-loop to iterate over all tiles and call the move function which is provided ins each tile.

Diagonal direction is not possible, as it is not implemented in our “direction” Enumeration.

Unit Testing
move() and merge() our most important functions. Possible tests could be :

- move on a empty tile
- move on another tile -->merging or not
- move on blocked tile, or the wall
- move on bonus tile
- move in a direction where no move is allowed
- try to merge tiles which are mergeable
- try to merge tiles which are not mergeable

Board Size
The Board size is given through an two Dimensional array, at initialising this, we can build different board sizes. Custom Boards Layouts are be possible by putting “Block” Cells on the field at initialisation.

Undo
The “Undo” function is done with the command Pattern. Every move is implemented as a command. Every Time a command is executed, another Command with the exact opposite action is pushed on a stack. Every undo-step is an executed function, popped from the stack. It’s necessary that each command has his own undo. For example if there had been a mering you must recreate the two tiles with the half value and set them on the correct position on the gamfield. For the normal move function it’s enough to move the cube in exactly opposing direction.

Inserting new Tiles

1. Check all Tiles if they are empty (Iterator Pattern checks if boolean “empty” is true)
2. Remember the amount of empty tiles
3. Create Random Number ‘n’ from zero to amount.
4. Iterate again, at the ‘n’th Iteration set the new Tile.

Overall game flow
The Game starts with two random placed tiles with the lowest possible value (eg. 2). Then the user can make his moves. After every move it must be checked whether the game is over or the game is winning. The game is over when all game fields are used and no move is in each direction possible. After each move the Gamefield and the Score must be updated for this task we use the Observer Pattern. If the Game is winning oder lost, give the user a restart button.

MVC Pattern

We strictly seperate the View from our game logic. Finally, the view takes arrays with objects (the tiles) from the controller, and builds a graphic user interface from it.

Used Patterns in Project

- Iterator
  iterates above tiles
- Command
  for “undo” function