Software engineering methods assignment 3

by

Irene van der Blij, Mayke Kloppenburg, Samuel Sital,
Kiran Kaur, Takang Kajikaw Etta Tabe

TI2206 Software Engineering Methods

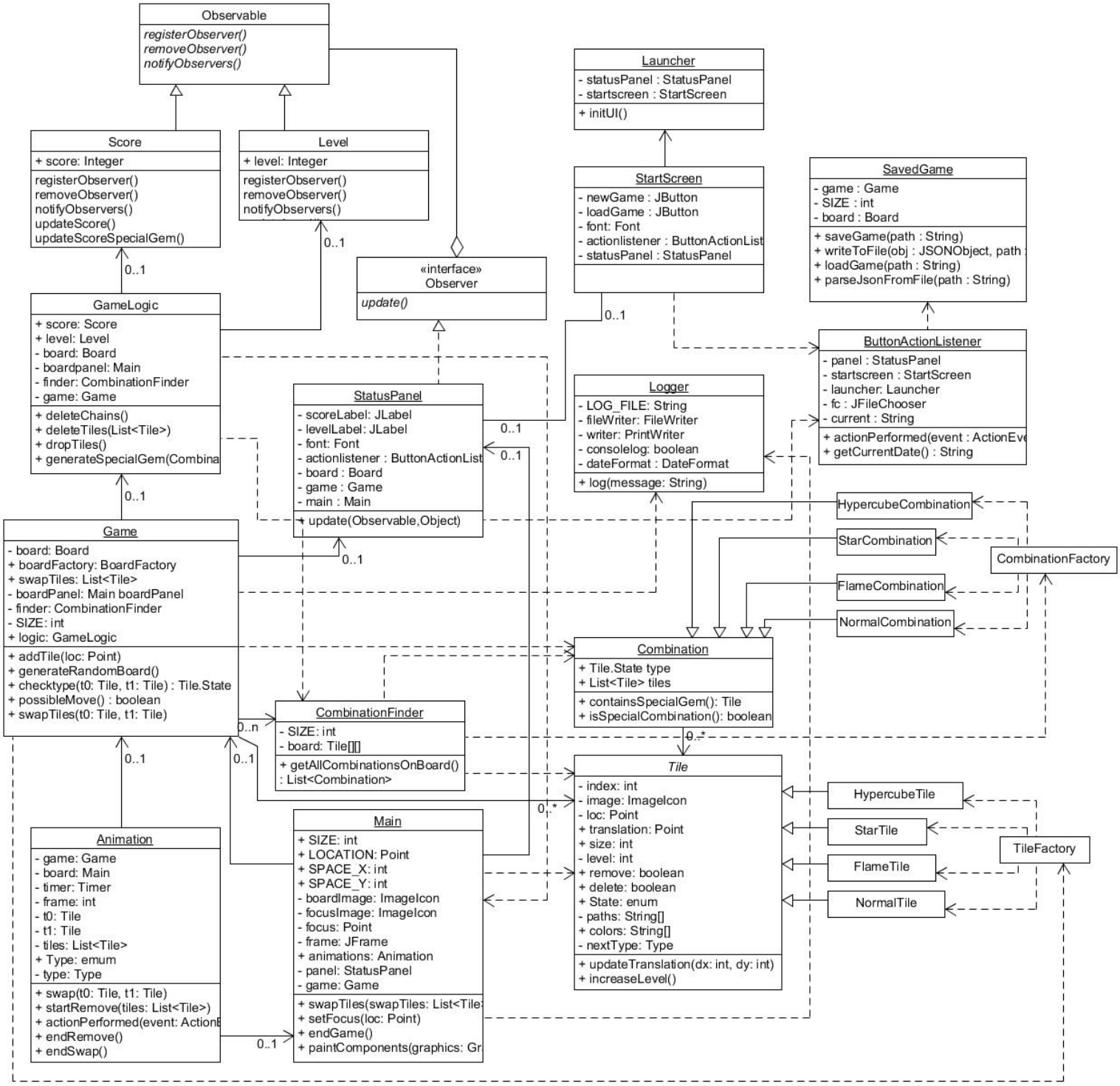
Group 37

Exercise 1 – 20-time, reloaded

To illustrate what we have done to further implement multiple saved games, the classes involved will be listed. Changes and additions made will be explained.

- **Launcher class** *(change)*
  Since the game is now save by clicking a button "Save Game" we do not require the game to be saved when the JFrame is closed. So the code in this class to save a game was deleted. Also, instead of immediately making a statuspanel (this is the screen with the bejewelled board), a StartScreen object is made. The responsibility of this class is to initialise a frame and stick the first screen on it. This essentially stayed the same; just a different screen.
- **StartScreen class** *(new)*
  This class initialises a start screen, which has two buttons: a load game button and a new game button. The responsibility of this class is only to make a screen with two buttons. The buttons have an action listener which handles the button actions.
- **ButtonActionListener class** *(change)*
  Since we have new buttons, the require new actions to be taken. It was decided all button have the same listener class, as the actions require StartScreen, Launcher or StatusPanel objects.
  - The save game button (from the statuspanel) now contains the code that was previously in the launcher class. Additionally, it calls an new method that generates a unique name (date + time) for the saved game.
  - The quit button (from the statuspanel) repaint the screen on the Jframe, so it wil return to the startscreen.
  - The load game button (from the startscreen) opens a dialog which allows a player to search the save game. When cancel is clicked, the dialog box goes away and when a file is selected and 'open' is clicked, the saved game will be loaded and the statuspanel with the board is painted.
  - The new game button (from the startscreen) paints a new random board and makes a new game with a score of 0 and a level of 1.
- **Main class** *(change)*
  Previously, a load game was always loaded in the constructor of Main. This has now been removed. The responsibility remains the same; although the Main no longer takes care of saved games. It only sets the game if there is a saved game loaded.
- **StatusPanel class** *(change)*
  The new buttons were added. The responsibility stayed the same.
- **SavedGame class** *(change)*
  The responsibilities stayed the same. It reads and writes to a file. There was only a small change made to accommodate multiple saves with different names. The methods now have an argument that gives the path of a file on a string. Previously this class had a final path attribute; this has been removed.

Here is an overview of our class diagram, including this extension:



Exercise 2 – Design patterns

The first design pattern we chose to apply to our code is the factory method.

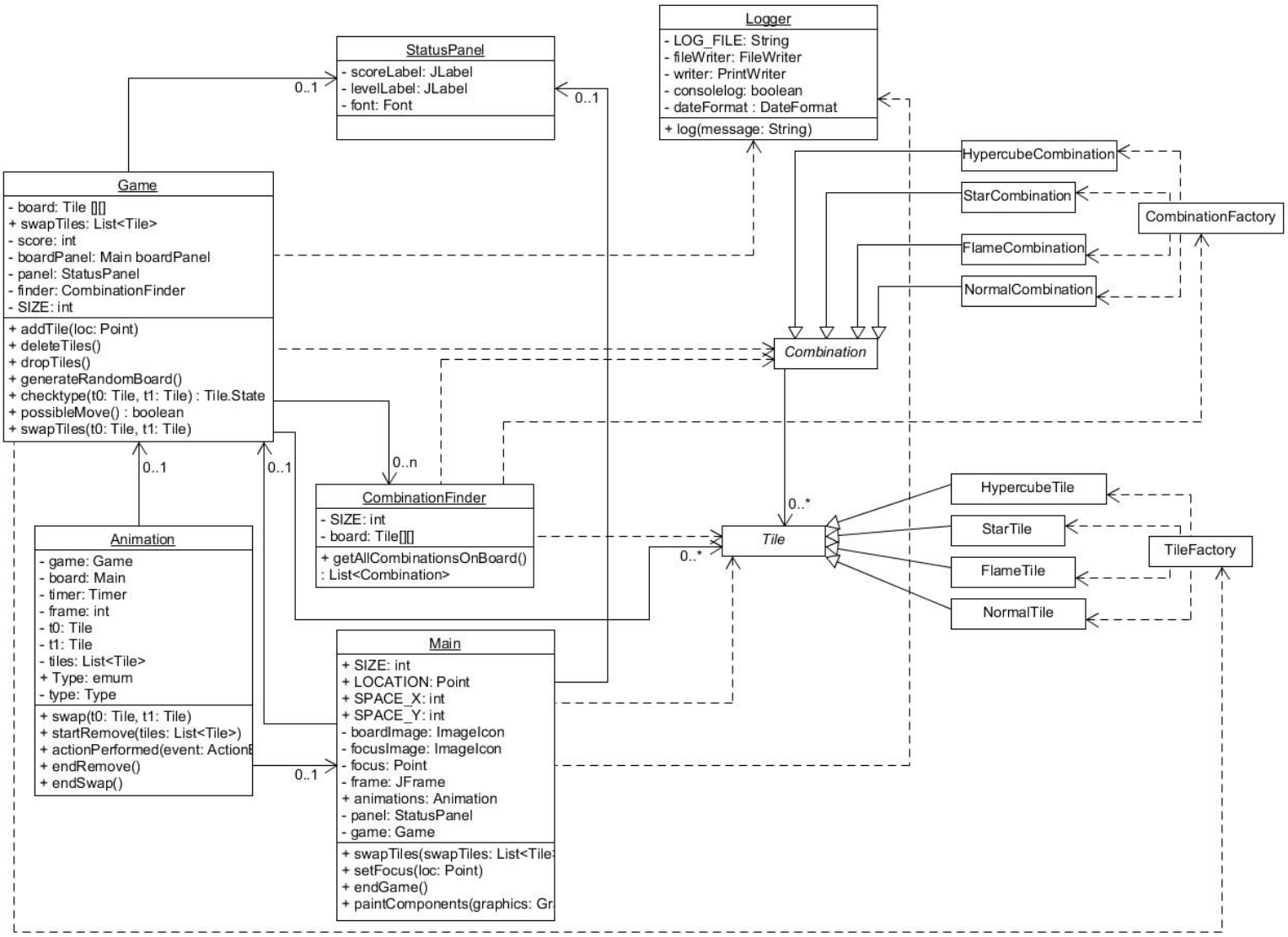
1. We chose to apply this pattern for the following reason. In our code we had the classes Tile, Flame Tile, StarTile and HypercubeTile. Whenever a new Tile had to be set via the setSpecialTile method, we had to check with if statements what the type of the new tile should be, and then create a tile of that class. To do this in a neater way we created a TileFactory, an abstract class Tile, and the classes NormalTile, FlameTile, StarTile and HypercubeTile which extend Tile. By creating a factory for Tile, the if statements only have to be once in that class, and so the code in the other classes can be simplified. An example of how this changed our code:

```
    public Tile setSpecialTile(int xi, int yi, Type type) {
      Tile tile = null;
-     if (type == Type.NORMAL) {
-       tile = new NormalTile(xi,yi);
-     } else if (type == Type.FLAME) {
-       tile = new FlameTile(xi,yi);
-     } else if (type == Type.STAR) {
-       tile = new StarTile(xi,yi);
-     } else if (type == Type.HYPERCUBE) {
-       tile = new HypercubeTile(xi,yi);
-     }
+     tile = TileFactory.generateTile(type, xi, yi);
+
      tile.setIndex(board.getTileAt(xi, yi).getIndex());
      Logger.log(type.toString() + " " + tile.getLoc());
      tile.setImage(new ImageIcon(tile.paths[tile.getIndex()]));
```

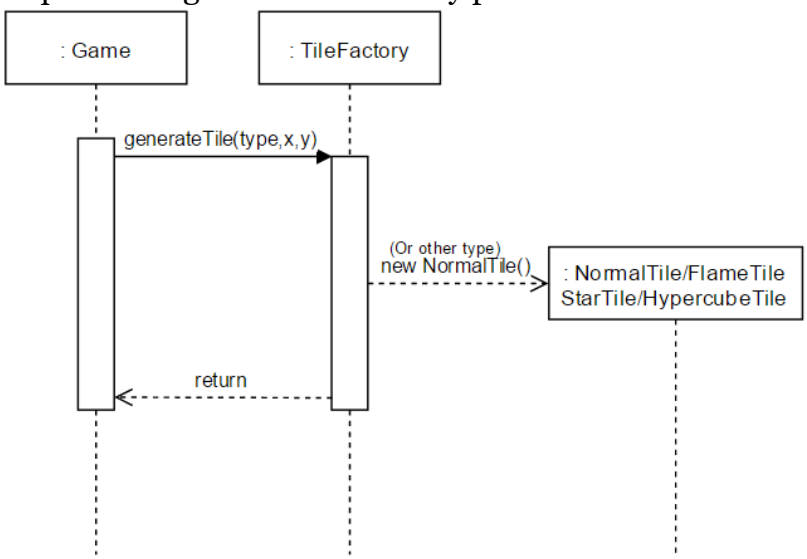
We did the same thing for the class Combination; so we created a CombinationFactory, an abstract class Combination, and the classes NormalCombination, StarCombination, FlameCombination and HypercubeCombination that extend Combination. This also for the sake of reducing the amount of if statements in our code. Here another example of how this pattern reduced our average method length (we added a method score in every subclass of Combination, so we could get rid of the switch statements):

```
-   public void updateScore(Type type) {
-     int score = 0;
-     switch (type) {
-       case NORMAL:
-         score = 50;
-         break;
-       case FLAME:
-         score = 150;
-         break;
-       case HYPERCUBE:
-         score = 500;
-         break;
-       case STAR:
-         score = 150;
-         break;
-       default:
-         break;
-     }
-     this.score += score;
+   public void updateScore(Combination combi) {
+     this.score += combi.score();
      Logger.log("Add score: " + score);
      Logger.log("Total Score: " + this.score);
      panel.setScore(this.score);
```

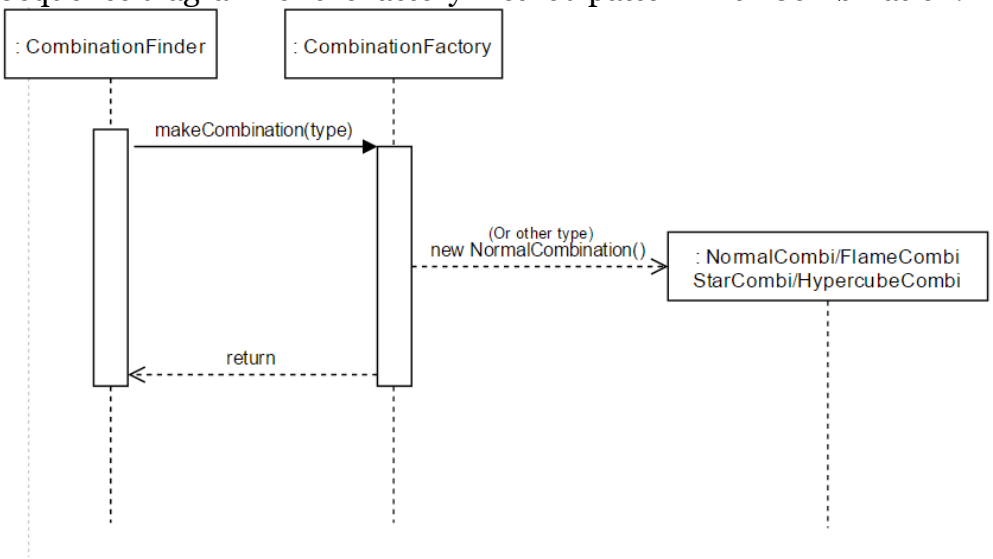
2. The class diagram of how the pattern is structured statically.



3. Sequence diagram for the factory pattern with Tile.



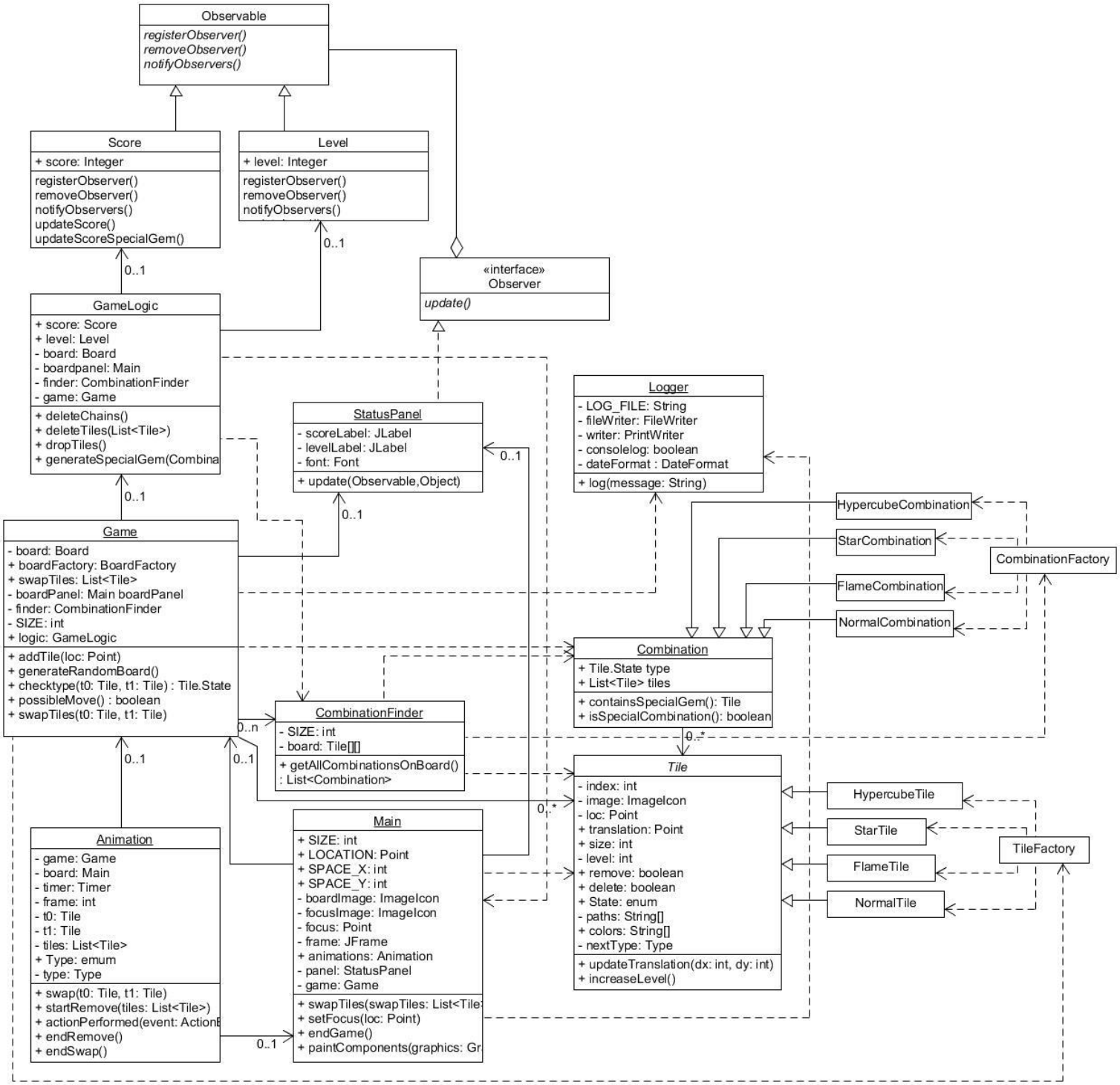
Sequence diagram for the factory method pattern with Combination.



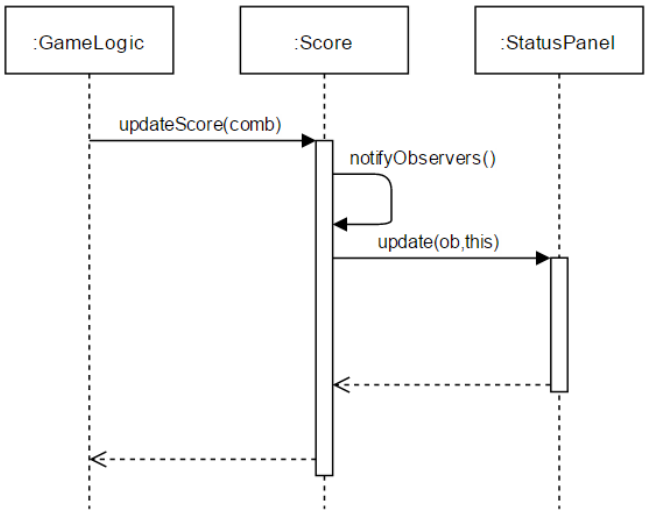
The second design pattern we chose to apply to our code is the observer pattern.

1. In our code, the intelligence calculates the current score and the current level, and this has to be displayed on the GUI. Of course we want to keep the coupling in our system as low as possible, and therefore this design pattern is a great way to do this, since this keeps the intelligence and the GUI separated. Also, if we ever want another object to also get the updated score and level, this object can easily be added to the list of observers. For updating the score we implemented this as follows. We made a class Score which extends Observable, since we want the score to be observed. Whenever a method is called which changes the score, the method notifyObservers() is called. In this way all the registered observers are given a notice of the updated score. The class StatusPanel now implements Observer, since this class wants to observe the score to be able to set the right score on the GUI. This class implements the method update() (which is called by notifyObservers()), which actually updates the score on the JLabel. In exactly the same manner Level is updated.

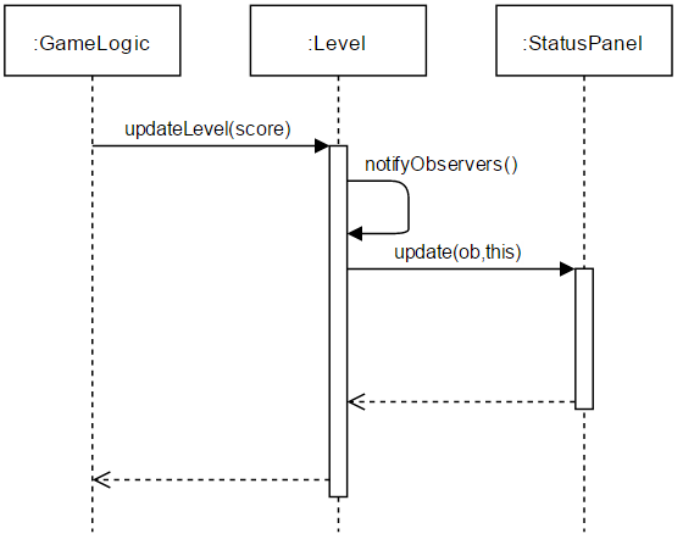
2.


Observable
registerObserver()
removeObserver()
notifyObservers()
Score
+ score: Integer
registerObserver()
removeObserver()
notifyObservers()
updateScore()
updateScoreSpecialGem()
Level
+ level: Integer
registerObserver()
removeObserver()
notifyObservers()
«interface»
Observer
update()
GameLogic
+ score: Score
+ level: Level
- board: Board
- boardpanel: Main
- finder: CombinationFinder
- game: Game
+ deleteChains()
+ deleteTiles(List<Tile>)
+ dropTiles()
+ generateSpecialGem(Combina
StatusPanel
- scoreLabel: JLabel
- levelLabel: JLabel
- font: Font
+ update(Observable,Object)
Logger
- LOG_FILE: String
- fileWriter: FileWriter
- writer: PrintWriter
- consolelog: boolean
- dateFormat : DateFormat
+ log(message: String)
Game
- board: Board
+ boardFactory: BoardFactory
+ swapTiles: List<Tile>
- boardPanel: Main boardPanel
- finder: CombinationFinder
- SIZE: int
+ logic: GameLogic
+ addTile(loc: Point)
+ generateRandomBoard()
+ checktype(t0: Tile, t1: Tile) : Tile.State
+ possibleMove() : boolean
+ swapTiles(t0: Tile, t1: Tile)
HypercubeCombination
StarCombination
FlameCombination
NormalCombination
CombinationFactory
Combination
+ Tile.State type
+ List<Tile> tiles
+ containsSpecialGem(): Tile
+ isSpecialCombination(): boolean
CombinationFinder
- SIZE: int
- board: Tile[][]
+ getAllCombinationsOnBoard()
: List<Combination>
Tile
- index: int
- image: ImageIcon
- loc: Point
+ translation: Point
+ size: int
- level: int
+ remove: boolean
+ delete: boolean
+ State: enum
- paths: String[]
+ colors: String[]
- nextType: Type
+ updateTranslation(dx: int, dy: int)
+ increaseLevel()
HypercubeTile
StarTile
FlameTile
NormalTile
TileFactory
Animation
- game: Game
- board: Main
- timer: Timer
- frame: int
- t0: Tile
- t1: Tile
- tiles: List<Tile>
+ Type: emum
- type: Type
+ swap(t0: Tile, t1: Tile)
+ startRemove(tiles: List<Tile>)
+ actionPerformed(event: ActionE
+ endRemove()
+ endSwap()
Main
+ SIZE: int
+ LOCATION: Point
+ SPACE_X: int
+ SPACE_Y: int
- boardImage: ImageIcon
- focusImage: ImageIcon
- focus: Point
- frame: JFrame
+ animations: Animation
- panel: StatusPanel
- game: Game
+ swapTiles(swapTiles: List<Tile
+ setFocus(loc: Point)
+ endGame()
+ paintComponents(graphics: Gr
0..1
0..n
0..*

3. Sequence diagram for updating the score:



Sequence diagram for updating the level:



Exercise 3 – Software Engineering Economics

1. How good and bad practices are recognized

To recognize bad and good practices you first have to analyse the overall performance of all the projects in a specific repository.

A project is assessed as good practice when the performance on both cost and duration is better than the average cost and duration corrected for the applicable project size.

A project is assessed as bad practice when the performance on both cost and duration is worse than the average cost and duration corrected for the applicable project size.

For each project you calculate the deviation from the average trend line and express it in a percentage (negative when below the average trend line and positive when above the trend line).

On the basis of these percentages you should plot the projects in a matrix, which results in four negatively or/either positively deviated quadrants. Such matrices give us important performance indicators.

Taking into account the aforementioned performance indicators we can recognize bad and good practices, since good practices on average show the best Productivity, Time-to-Market and Process quality of all the four quadrants. And for Bad Practices, Productivity, Time-to-Market and Process Quality are the lowest of all the four quadrants.

2. Why Visual Basic being in the good practice group is a not so interesting finding of the study?

Visual Basic was in the good practice group mainly because the people dealing with it had better Visual Basic programming skills and due to the fact that Visual Basic project environments are relatively less complex and easy to use.

However, this clearly means we are valuing less complex projects and programming skills over all the other qualities. *And a company cannot simply change its programming language overnight*. Both of these reasons make it a not so interesting finding of the study.

3. Other factors which could've been studied in this paper

Listed below are a few points which are listed but not studied in this paper. The chosen practice(good or bad) is not sure but is chosen based on our own analysis of what makes a practice good or bad.

1. Minor enhancements:

In any project, once started, problems always arise which need to be fixed. This is done in the course of the project. Minor enhancements typically contain code optimizations, bug fixes, high-value/low-risk requests for enhancements, and minor user

interface changes. There is limited architecture development in minor releases. The changes must be accomplished using the architecture deployed in a former major release. This therefore leads to an enhancement in the project, less defects per function points, decrease cost per function points if this is properly carried out because it means less money at the end of the project for bug fixes etc. Due to this reason, this will be considered as a good practice.

2. Oracle:

This is a tool or mechanism which is used to determine whether a test has passed or failed. Outputs of a given system are checked for a given test-case input and compared to what the oracle thinks it should be. Due to the fact that 26 of the ORACLE projects fall under good practice and just 3 under bad practice, we will consider this to be a good practice problem because it will lead to reduced costs per function point especially when it comes to the testing part of the software. This will therefore be put under good practice and also because it will lead to a project with less defects per function point and as such a project in the good practice quadrant.

3. Phased Project:

The management of a project is solely based on the idea that a project goes through a number of phases characterized by a distinct set of activities or tasks that take the project from conception to conclusion. Projects may be big or small, constrained by cost and time, often complex and therefore it is important to take a structured and defined approach to managing them through their entire lifecycle. This dealing of the project in phases is likely to lead to less defects per function point and also less cost per function point and as such a project in the good practice quadrant.

4. Bad practices and why they are in the bad practice group.

Bad practice factors are those factors that cause projects to be in the bad practice quadrant, which means that the performance of these projects on both cost and duration is worse than the average cost and duration for all the projects in the repository.

A list of some of these factors are :

- Rules and regulations.
- Dependencies with other system
- Once-only projects
- Technology based projects

1. Rules and Regulations:

This has to do with the fact that things have to be done in projects just because the government or authority says they have to be done and not because they really have to be done. This means the team is being forced to build this. This mostly leads to failures in projects and has been noticed to be one of the main reasons why projects fail and because of this, it is considered as a bad practice and it leads projects to the bad practice quadrant.

2. Dependencies with other systems:

This has to do with the fact that the system we are trying to build depends in one way or the other on other systems. This means building complexity on complexity, considering the fact that the system our system depends on is also a complex one. If

the system on which ours depends on fails, then it is obvious that our system is going to fail irrespective of the fact that it has been properly implemented. This mostly leads to project Failures and as such projects are led to the bad practice quadrant.

3. Once only Projects:..

We assume that starting up a once-only project, including having to do things for the first time, and for one project only, leads to a high probability of ending in between Bad Practice. This has to do with the fact that there is less time available for learning the required skills needed for that project and also that this project has to be performed once, meaning less time to acquire an experienced team which will most of the times lead to a failure in the project and causing this projects to fall under the bad practice quadrant.

The aforementioned factors all lead projects to the bad practice quadrant.
This quadrant holds projects that scored worse than the average of the total repository for both cost and duration. These factors are strongly related to a high percentage of Bad Practice. Due to these factors, the duration of a project per Function point in days becomes higher, the cost per Function of the project in Euros becomes higher as well and the total defects also increase, leading projects to the bad practice quadrant.