Assignment 4 **Pang!**:

Group: -1

Jurriaan den Toonder: 4431324

Erik Wiegel: 4476328

Govert de Gans: 4491955

Jaap-Jan van der Steeg: 4456130

Tim Rietveld: 4472926

Exercise 1 - Code improvements (30 pts)

1. Where do you use if or case statements in your source code? Refactor 10 of these statements used in different scenarios, so that they are not necessary anymore and describe your refactoring. Instead of if or case statements you can use, for example, design patterns, polymorphism, or a technique called double dispatch.

If you have less than 10 of these if or case statements, you have less work to do! If you find some rare cases of if or case statements that cannot be refactored you can convincingly explain why (we recommend to consult with your TA before going this way); these cases count toward the 10 you have to refactor (30 pts).

Case 1:

We replaced the switch case in NumberToken with:

```
   changeImage("/sprites/numbers/hud_" + number + ".png");
```

This is possible because the numbers have been properly named, hud_0.png hud_1.png etc.

Case 2:

In ScoreSystem the following if-statement was present:

```
if (Places.size() == 0) {
   Application.setScoreKeeper(new ScoreSystem());
   } else {
         for (NumberToken token : Places) {
        token.setScoreToken(0);
   }
}
```

We changed this to:

```
for (NumberToken token : Places) {
        token.setScoreToken(0);
   }
```

Because this check is not needed, ScoreSystem is a singleton, and the Tokens are never removed.

Case 3:

We replaced various clampings we had done with:

```
Target = variable
If(variable <or> constant){
        Target = constant
}
```

With the appropriate:
Target = Math.max(variable, constant) or Math.min(variable, constant)

We did this in:

- ScoreSytem for **Score = max(999999999)**
- TimeSystem for **time = max(999)**
- Ball for **xpos = min(0)** and **max(board.width - object.width)**
- Ball for **ypos = max(board.height - object.width)**
- Drop for **xpos = min(0)** and **max(board.width - object.width)**
- Drop for **ypos = max(board.height - object.width)**
- Player for **xpos = min(1)**
- FreezeTimeDrop for **timeFrozen = min(1)**
- HookDrop for SetMaximumHooks(value) and **value = min(1)**
- StickyHookDrop for **stickyDelayTime = min(0)**

Which were all individual cases adding up to 12 different removed if-cases.

Exercise 2:

Requirements Pang!:

Difficulty:

Group: -1

Jurriaan den Toonder: 4431324
Erik Wiegel: 4476328
Govert de Gans: 4491955
Jaap-Jan van der Steeg: 4456130
Tim Rietveld: 4472926

Functional Requirements:

Must have:

- There must be a new screen where you can choose difficulty
- There must be a button for each difficulty
- There must be different levels for each difficulty
- There must be at least an easy, medium and hard difficulty

Should have:

- The difficulty in a difficulty category should increase over the levels
- Each difficulty should be available to both singleplayer and multiplayer mode

Could have:

- The game could have an insane difficulty

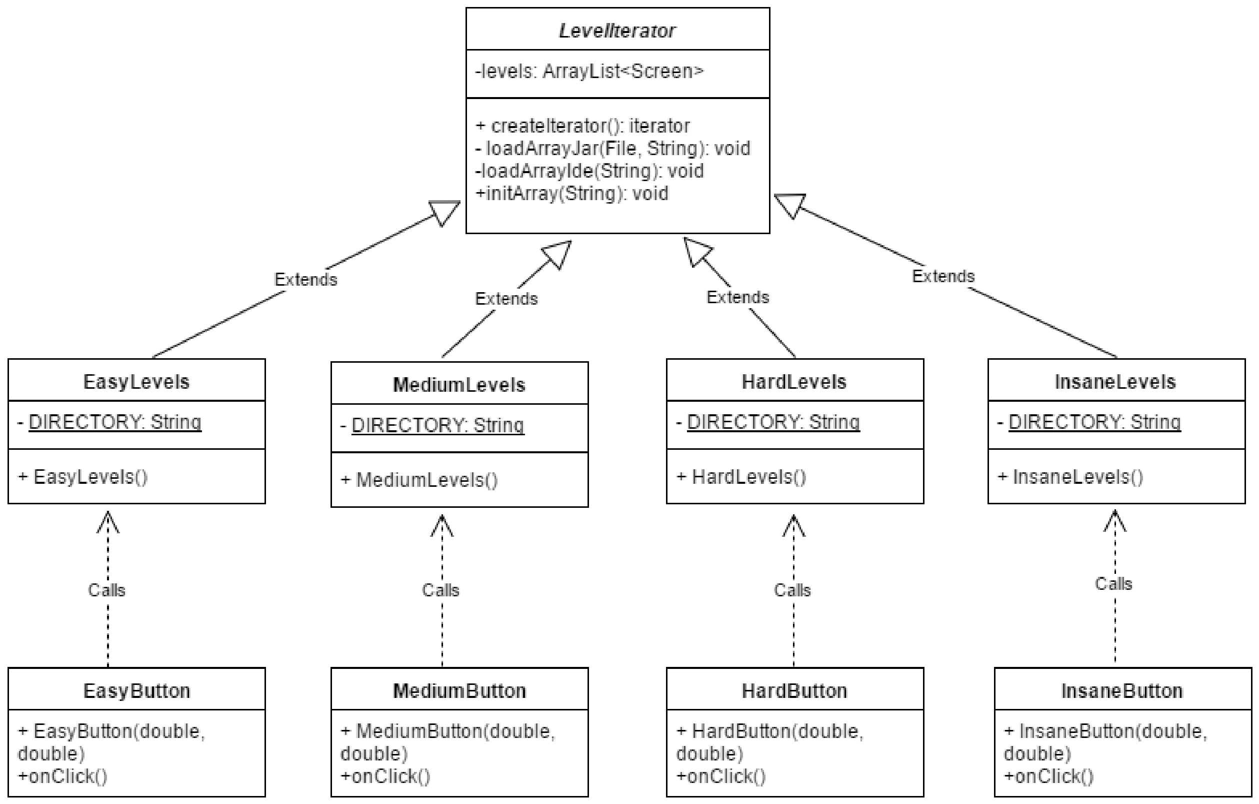
Won't have:

- There won't be a difference between levels for singleplayer and multiplayer

Non-functional Requirements

- The buttons must be in the same style as the rest of the game

2. During the analysis and design phases of this extension use responsibility driven design and UML (push to the repository the single PDF file including all the documents produced) (8 pts).



Exercise 3 - Walking in your TA's shoes (30 pts)

Reading, understanding, and evaluating code written by other software engineers is the bread and butter of professional software engineering. Ask your TA for the codebase of another group (which will be provided as an anonymous .zip archive) and perform the following tasks:

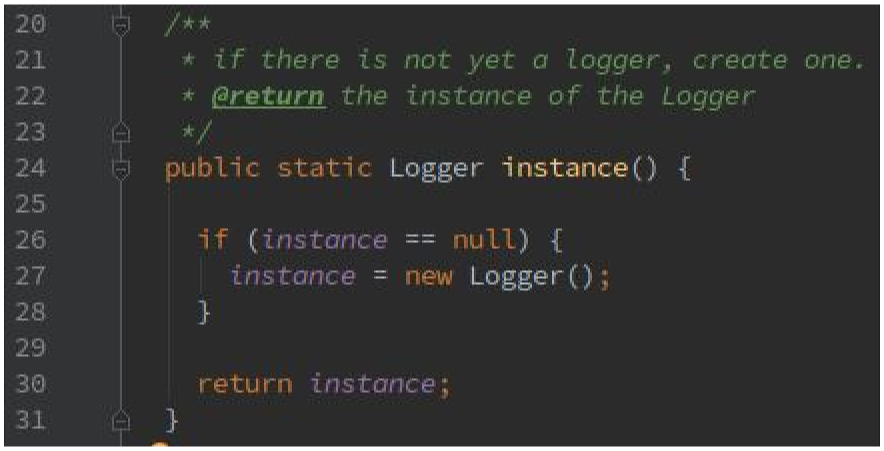
1. Grade the code quality according to the rubrics. You have to deeply detail the reasons for your choices. (15 pts).

		Exemplary			Competent			Developing			
		10	9	8	7	6	5	4	3	2	1
Source code quality						X					
		Design patterns are well placed and implemented. Methods are of a proper size. No design flaws are present. Other software engineering principles are also used appropriately and correctly.			Design patterns are well placed, though not implemented completely. Only few and limited design flaws are present. Other software engineering principles are also used appropriately yet sometimes incorrectly.			Inappropriate use is made of design patterns or anti-patterns are used. Design flaws (e.g., God classes and feature envy methods) are readily present. Other software engineering principles are also used inappropriately or not all.			
Testing					X						
		The system has a >75% of meaningful test coverage (maven cobertura, line coverage).			The system has a >45% of meaningful test coverage.			The system has a <20% of meaningful test coverage.			
Code readability	**Formatting**					X					
		Code is well formatted and follows the language's conventions.			Code is well formatted yet diverges from the language's conventions with little motivation.			Code is poorly formatted and does not follow the language's conventions.			
	Naming				X						
		Appropriate and clear names are used for variables, methods, classes, etc.			Most names are clear and appropriate; some names are ambiguous or vague.			Names are vague, ambiguous, or unrelated to their contexts.			
	Comments				X						
		Appropriate use is made of comments to document the code correctly.			Most comments are appropriately used yet some lack purpose or are unnecessary.			Little to no comments of value; code is poorly documented via comments.			
Tooling (i.e., GitHub, Cobertura, FindBugs, PMD, CheckStyle)						X					
		The required tools are correctly setup and used with every change.			The required tools are correctly setup but not correly used with every change.			The required tools are neither correctly setup nor correctly used with every change.			

2. Propose meaningful enhancements to the other's group codebase. Meaningful enhancements are, for example, based on design patterns and design principles. These enhancements must be worth 30 points for a weekly assignment. (15 pts). Push to the repository the single PDF file including the output of the aforementioned two tasks.

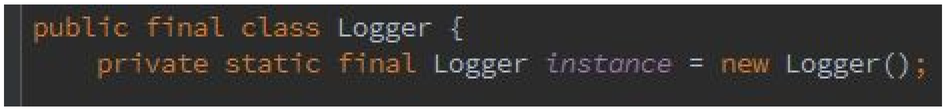
Logger.java

This singleton logger is not thread safe due to the following piece of code:

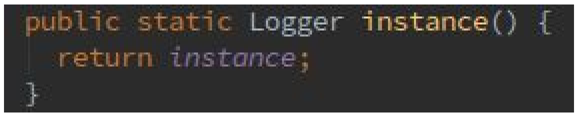


```java
/**
 * if there is not yet a logger, create one.
 * @return the instance of the Logger
 */
public static Logger instance() {

  if (instance == null) {
    instance = new Logger();
  }

  return instance;
}
```

Multiple threads are able to enter this method simultaneously. The instance should rather be made eagerly:



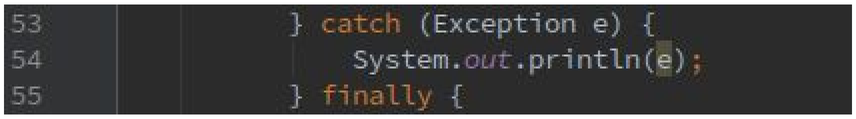
```java
public final class Logger {
    private static final Logger instance = new Logger();
```



```java
public static Logger instance() {
  return instance;
}
```

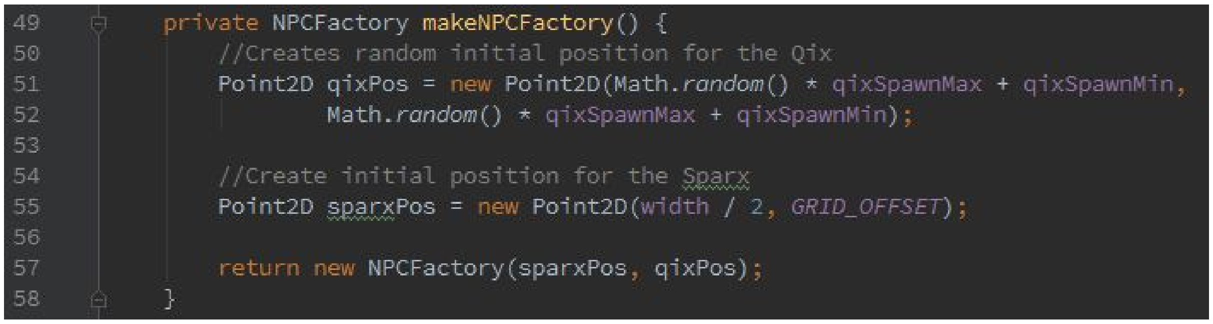
The protected void log(..) method at line 46 should be set to private, as there is currently no extension available for Logger.

Is it really good practice to catch all Exceptions here? How about, per example, OutOfMemory exceptions, should these not be handled?



```java
        } catch (Exception e) {
            System.out.println(e);
        } finally {
```

GameFactory.java

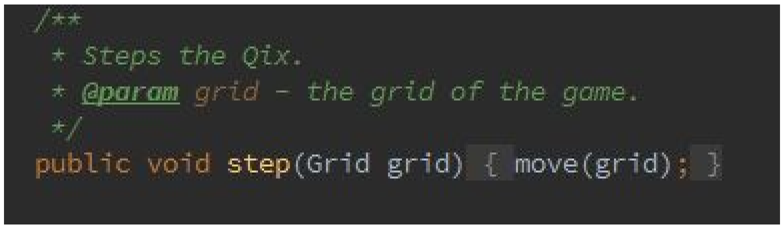


```java
    private NPCFactory makeNPCFactory() {
        //Creates random initial position for the Qix
        Point2D qixPos = new Point2D(Math.random() * qixSpawnMax + qixSpawnMin,
                Math.random() * qixSpawnMax + qixSpawnMin);

        //Create initial position for the Sparx
        Point2D sparxPos = new Point2D(width / 2, GRID_OFFSET);

        return new NPCFactory(sparxPos, qixPos);
    }
```

It doesn't make sense to give the sparx and qix positions to the factory. The factory should return the NPC objects based on a parameter given or method called, but movement and position should be handled in the NPC itself. Currently the random positions of the NPC's are determined in the game factory, which means that 3 classes are now intertwined.

Grid.java

Would it not make sense to transform Grid to a singleton? There can only be one per game. This also removes the need to send the instantiated grid as a parameter to – per example – *Qix.step()*:



```java
/**
 * Steps the Qix.
 * @param grid - the grid of the game.
 */
public void step(Grid grid) { move(grid); }
```

NPCFactory.java

This is a wrong implementation of the factory design pattern. A factory should be created, and then told 'what type of' object to make. Thus: the NPC factory should receive in its parameters what type of NPC to make, or there should be different methods implemented for the different types of NPC's. In that case an Abstract Factory Pattern would be even better. It should not handle the positions of these NPC's. Currently NPCFactory is just used to spawn in the different types of NPC's at the beginning of the game. But one could not, per example, make another level using this factory, as the amount of NPC's and positions of these NPC's are 'hardcoded' into the NPCFactory. One should be able to instantiate this factory to create and retrieve NPC's (at runtime). This is currently not possible.

JQixUI.java

Methods like moveNPCs(), updateStix(), updatePlayerLine(), updateTerr() and moveCircle() should be moved to NPCanimation. The logic of moving the NPC should be moved to the NPC itself or another overseeing object.

General remarks:

There is a lot of logic intertwined between classes. Extending code in one class **guarantees** required changes in other classes. Take, per example, the NPC's. They get accessed using known indexes in a list. If one would remove an NPC from the game, he or she would be required to look through all the classes and find usages of this list and alter these pieces of code too. The application is therefor not open to extension as it is currently.

Some basic remarks about standard Java code style are:

- You should not use final static variables as parameters for methods in other classes, as these should be defined in that class itself (GameFactory makeGame() for example uses DEFAULT_SCORE).
- We doubt the added value of your checkstyle.xml file, as running checkstyle with the default configuration results in more than 600 errors (like missing javadoc, unused JavaDoc tag etc).
- We noticed that your WINDOW_X and GRID_X are independent in your application, whereas they should not be independent.
- You should not include parameters in javadoc that are not included in the method (Grid.java addPointsToTerr for example).
- Unused methods should be removed (ShapeUtils.java cloneTerr for example)
- Methods invoking returning an property of an object should be defined in that object class itself (for example getPlayerPos() in BackEnd.java should be defined in Player)
- Try to seperate code as much as possible (JQixUI is a very big class right now)