1. Code improvement

Refactoring if- else statements

1. **Refactored if-statements in the tetris/Keybindings class**
   Commit #83
2. **Refactored the Logger using Polymorphism in the logging/Logger clause**
   Commit #69
3. **Removed if-statements in the tetris/Controller class**
   Commit #26
4. **Refactored case-statements in the tetris/TetrominoMovementHandles**
   Commit #66
5. **Refactored even more in tetris/Keybindings class**
   Commit #43
6. **Pauze Button refactored using double dispatch in the tetris/Controller class**
   Commit #82
7. **Refactored Cases in the tetris/View class**
   Commit #36
8. **Refactored an if-statement in the tetromino/TetrominoQueue class**
   Commit #36
9. **Refactored an if-statement in the logging/Logger class**
   Commit #31
10.

Code coverage

One of our tasks this week was getting our code coverage up to 60%. To do this we tested the settings class and the action class, which got our coverage up to 58%. We got the coverage up to 73,6% after testing the GameScreen. The GameScreenTest only tests a view parameters, but covers almost all the code. Maybe Something for next week to write some more tests.

2. New Features

Wall kick tetromino

One of the features that was supposed to be implemented this week was the ability for the tetromino to push itself away from the wall, instead of just not turning. The AbstractTetromino class checks, after the rotation, if the grid is free, is not, it rotates back. This method, to check if the grid is free, asks for a Coordinate object. To make this function work I created 2 methods to get the coordinate (instead of an integer) of the most left and most right coordinate of the tetromino. Now I could rebuild the method that instead of turning rotating back, it could push itself to the right or to the left (depending on which side of the tetromino didn't have space in the grid).

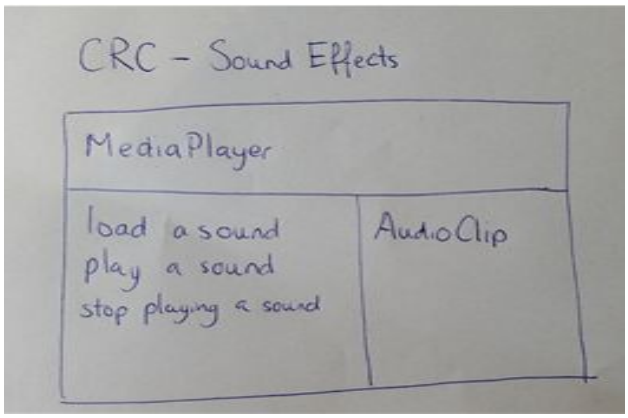
Pauze Button

The second feature we wanted to implement, was a pause button. If the button is pushed, the game will pause, obviously without resetting the score, screen, etc..
Initially two buttons were created, a pause and an unpause button with obvious functions. The pause button pauses the timer and tells the boolean that the player is gameover(gameover= true). This has to happen, because otherwise playerinput would still be registered. The unpause button does the exact opposite of the pause button.
Eventually both buttons were merged with if-statements using the boolean gameover as a check whether to pause/unpause. These if-statements were further refactored.

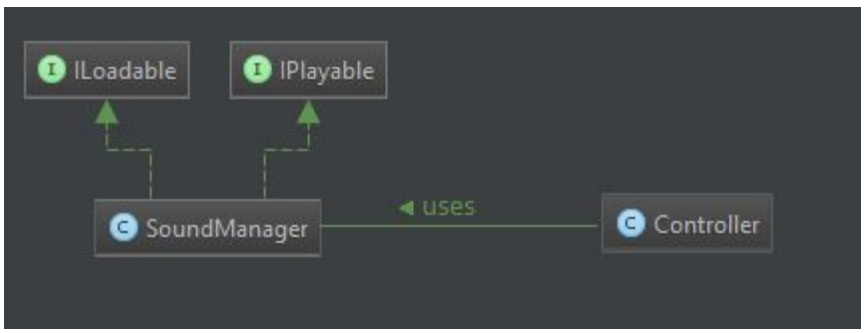
Music Player

JavaFX makes it quite easy to implement audio playback. You simply load up an audio file and create an AudioClip object that is in the javafx.scene.media package. The game should have several different audio clips that should be initiated when for example a movement or rotation is performed by the player. Therefore we needed some kind of object that store a reference to an audio file along with an identifier to be able to play the sounds at any desired moment more easily.

Using Responsible Driven Design, the only class needed for this feature is MediaPlayer (eventually called SoundManager in the real implementation)



The following UML represents the actual implementation of MediaPlayer in the game:



The controller holds a reference to an instance of SoundManager. The controller then loads some audio files like the Tetris theme song and some sfx for movement and rotation into the SoundManager object. When a player start the game the controller calls the play() method on the SoundManager instance and the theme song is started.

Every in-game action a player can perform has an attempt() method that tries to perform the action. This method had a void signature. Therefore it could not be known from the outside of this method if the attempt was successful or not. We had to change the signature of these attempt() method into the return of a boolean. Luckily this was very easy to change.

3. Pang Review

This review covers the Pang game of a group anonymous programmers. The review involves evaluation of test coverage, code quality and tooling.

1. Overall observation:

The source files in the project are split into packages by functionality (display, gameplay, graphics, input, menu, miscellaneous, sound, utility). The resources folder contains packages with game assets (maps, sounds, sprites) and a few images not in a package.

The test folder contains packages with names corresponding to the packages in the source folder. One test class (GameStateManagerTest.java) is not in a package.

The other project root folders are ‘target’ used by Maven and .idea containing Intellij-settings.

The root folder also contains some setting file for i.a Maven, CodeCov, Travis and CheckStyle.

There is a README.md file with a brief explaination for how to create a CodeCov report, a list with known bugs and acknowledgment for sounds used in the game.

When starting the game, without any further knowledge of the details of the code, the game seems to be working well. A soundtrack is heard and all the buttons work. When starting the game, the UI looks fine and some nice graphics are used. The game is missing some key-functions(dying for instance), but supposably these are going to be added in the future. The biggest concern is that the game crashes after shooting a few times/ being in game for a bit.

2. Test coverage

Out of the 78 test all of the tests pass and with these 78 tests the cover 62,7% of the code, which is acceptable. This percentage obviously needs to go up for the game to be properly tested. Looking at the code that needs to be tested, getting the coverage up shouldn’t be too much of a problem. For instance looking at the class Bubble. A Dummy setup could be made in the test code, and after certain actions (methods) have happened like splitLeft() or collide(), and after it has updated, the location of the Bubble can be checked. I quess the tests aren’t done due to a lack of time, but seeing this does give a good perspective for the future.

The tests that they have done do seem good to me, the test the expected outcome using assert tests and verify(). Verify is something we haven’t used yet in our own test, maybe this is something we can look into the upcoming week.

3. Code quality

This evaluation includes the following considerations:

- Readability Was the code readable in terms of naming, formatting and comments?
- Modularity Was the code ordered in logical units?
- Consistency Was the implementation quality the same way throughout the code?
- Maintainability Is the code easily extendable, fixable or modifiable?
- Usability How well does the code perform in terms of user interaction?

3.1 Readability

Was the code readable in terms of naming, formatting and comments?

Throughout the code there were some inconsistencies in terms of formatting. Some files use 2 spaces for indentation (e.g FpsCounter) while other files use 4 spaces for indentation (e.g Sound). We would suggest using either 2 spaces of 4 spaces of indentation (but not mixed) for consistency.

In addition, there are some methods that are not used anywhere in the code. This clutters up the overall readability. The lines 140:150 in Map.java are wrapped in a if-statement that never returns true. We would suggest to either remove this code or fix it, so that it doesn't degrade the readability of the rest of the code.

As far as concerned, naming convention was clear and consistent.

Most methods have a clear description of what they should be doing. Some methods, like update() in Bubble.java and Player() in Player.java, are a bit too long. Nevertheless, most of the code is accompanied by clear comments.
Commenting is really well done. Everything that might be hard to understand has a comment. This makes the overall readability easier to understand.

Besides this, to improve readability we suggest to set some stricter rules in CheckStyle to adhere to.

3.2 Modularity

Was the code ordered in logical units?

The code is very well organized by functionality. One minor point of improvement (only when open sourcing this code) could be to separate packages so that it is clear what packages belong to the core-logic of the game and what part belongs to the UX/UI part. This will make it easier to reuse only the core-logic and build a different UX/UI layer on top.

3.3 Consistency

Was the implementation quality the same way throughout the code?

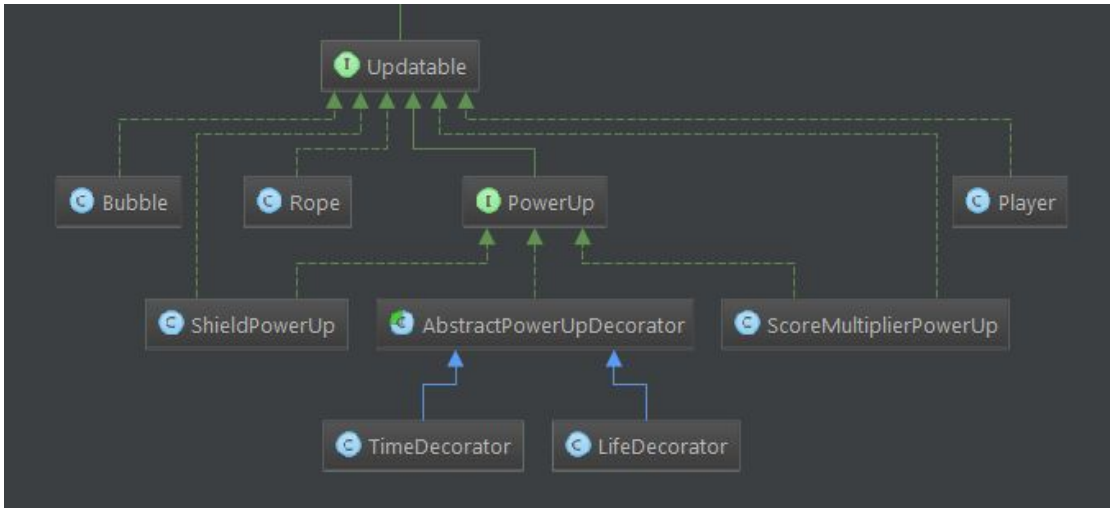
The implementation of logic is very consistent throughout the code. Delegation is properly used, so that every class is minimized to having just a single reason to change i.e Single Responsibility Principle. Pieces of logic are split into logical methods and no excessive nesting is present.

3.4 Maintainability

Is the code easily extendable, fixable or modifiable?

This comes down mostly how well the code adheres to the SOLID principles and if design patterns are used in a correct manner. As mentioned in 3.3, most (if not all) code is nicely split up in class that carry just a single responsibility. This makes it easier to modify existing code, because it is not intertwined with other responsibilities.

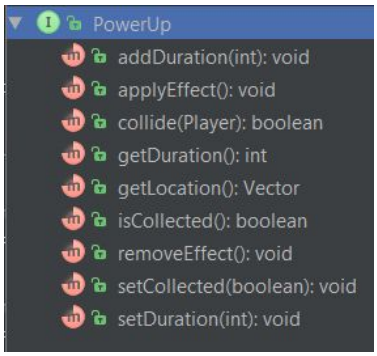
The following illustration show the hierarchy of power ups.



For instance, TimeDecorator and LifeDecorator are both AbstractPowerUpDecorators. This enables to use of polymorphism, because it doesn't matter what concrete class of AbstractPowerUpDecorator is initiated to be able to call a method in the public API of AbstractPowerUpDecorator. Because of this polymorphistic structure it is easy to add a new class that extends AbstractPowerUpDecorator without modifying existing code. On other words, the code is open for extension, but closed for modification, i.e Open Closed Principle.

We didn't spot a derived type in a hierarchy that was not properly replaceable by its base class. Therefore we no violation of the Liskov Substitution Principle was found.

As can be seen in the illustration above there is a PowerUp interface. This interface has the following methods:



Implementing this interface forces a client to implement all of these interfaces. When for example a powerup is created that is permanent for the whole game the duration methods are redundant and meaningless. Therefore it would be better if these methods are moved to another interface, like an expirable interface. This way the client is not forced to implement interfaces it don't need, i.e the Interface Segregation Principle is not voilated.

When we were looking at the constructor of Player, we noticed a Rope object was created on line 55. The class Player depends on the lower level class Rope which voilates the Dependency Inversion principle. Both should instead depend on an abstration layer. This

makes it easier to inject other types of Ropes and helps to make unit testing easier since the tested unit does not depend on another unit anymore.

3.5 Usability

How well does the code perform in terms of user interaction?

The overall impression was that the game runs very well. It feels smooth and robust. Besides, one show-stopping exception, we didn't came across any other issues.

4. Tooling

This code is supported by a task runner called Maven. The pom-file containing Maven's configuration is found in the project's root. Several external libraries were used, like JUnit for testing, Cobertura for code coverage reporting, Powermock for mocking objects in unit tests.

For code quality CheckStyle is integrated. As recommended in 3.1 we would suggest to set some CheckStyle rules for indentation stricter to have a more consistent indentation throughout the code.

PMD en FindBugs are included for code robustness which can be considering as a good thing.

Singleton Design pattern implemented in the logger class

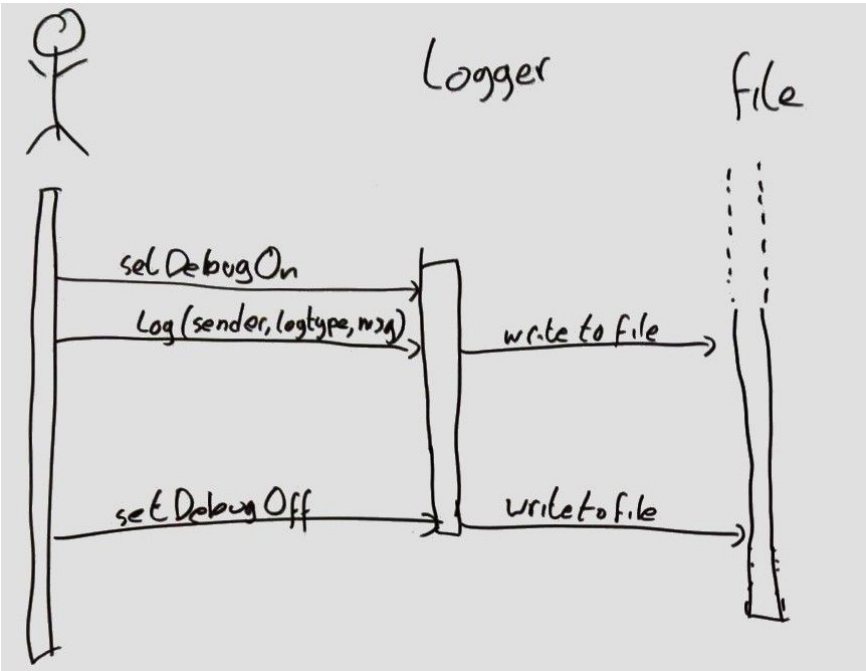
The application logger has been implemented with the singleton design pattern. This offers two adavantages: 1) The logger does not need to be either passed around to each object or newly created in each object, and 2) The logger can buffer write actions, which significantly decreases the number of I/O operations. Hence, the logger is accessed by simply importing the logger class and calling the static methods that append your message to the queue. In addition the logger can be activated and deactivated via static methods to optimize efficiency for release builds.

Implementation specifics

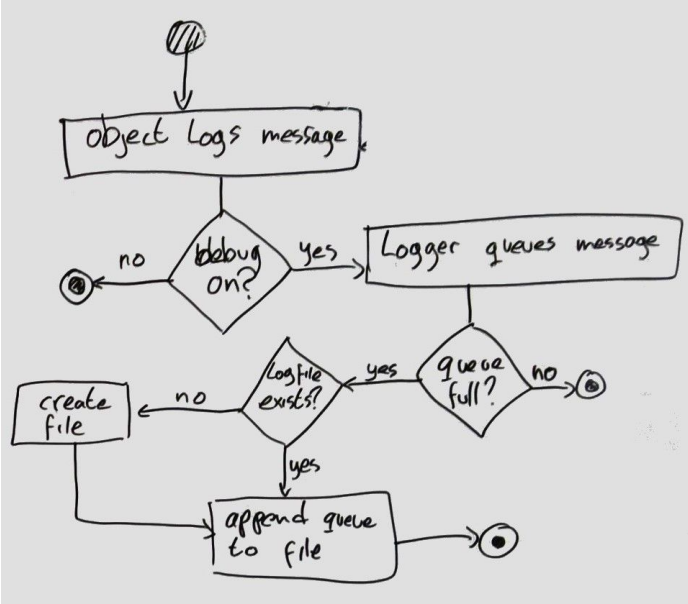
Unlike plain-vanilla singleton implementations, there is no need for a getInstance() method. Instead the state-switch methods setDebugOn() and setDebugOff() are in charge of instance management.
Since the logger does not alter the state of the application beyond himself and the log file, The logger is thread-safe. Also, the logger should cause as little interference with the program as possible, hence the logger instance inherits from Thread such that any expensive I/O operations are executed parallel to the main application.

How the logger should work

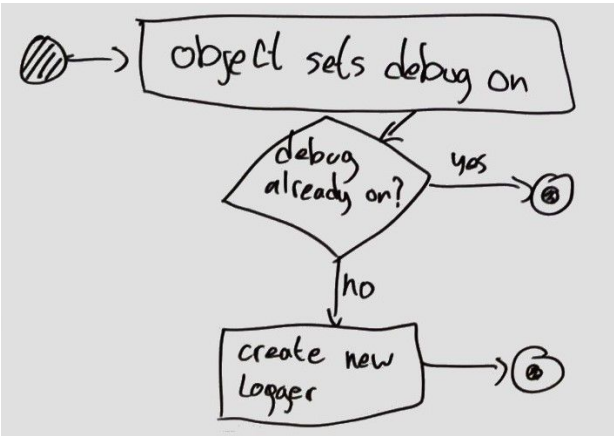
The following sequence diagram shows a typical lifeline for the logger. Whether the file exists when the logger wants to write to the file doesn't matter as the logger will create a new file if absent and will always append to the old file.



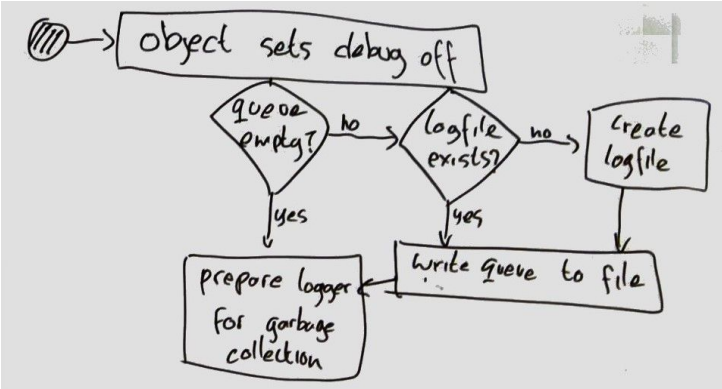
The following Activity diagram shows how the log action is performed.



The functionality of the setDebugOn method is shown in the following activity diagram:



And lastly the setDebugOff functionality is given by this activity diagram:



Dependencies

By design the logger has no dependencies to other classes of the application. This is so because a logger can be seen as output of the application, thus any dependency of the logger to the application would be unwanted; this dependency should be application-internal.