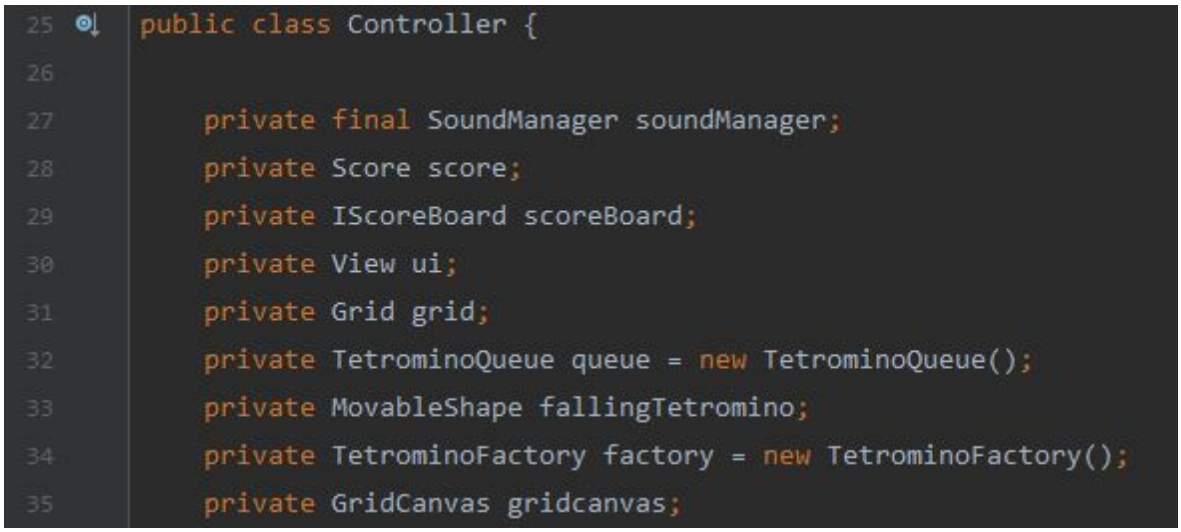
Anonymous peer suggestions

Last week our game was reviewed by some anonymous colleagues. They noticed a violation of the Open/Closed principles. This week we tried to refactor as much code as we could in order to eliminate this violation.

We had a look at our code and spotted a violation of the Open/Closed principle. For example, adding a shape to the game would force modifying existing code. We decided to refactor this in a way that we could easily add shape to the game without modifying any code (i.e it must be open for extension but closed for modification).

At line 34 of Controller.java we see that an instance of TetrominoFactory is created. This is the factory used for the creation of shapes to be dropped in the grid.



Let's have a look at TetrominoFactory:

```java
public class TetrominoFactory {

    /**
     * This creates one of the shapes defined in TetrominoType.
     *
     * @param   type            Type of Tetromino to create
     * @return  AbstractShape   Tetromino
     */
    public AbstractShape create(TetrominoType type) {
        switch (type) {
            case I:
                return new TetrominoI();
            case J:
                return new TetrominoJ();
            case L:
                return new TetrominoL();
            case O:
                return new TetrominoO();
            case S:
                return new TetrominoS();
            case T:
                return new TetrominoT();
            case Z:
                return new TetrominoZ();
            default:
                throw new IllegalArgumentException(String.format("Unkno
        }
    }

}

```

One thing to notice is that the create() method accepts a TetrominoType enum. Which looks like this:

```java
public enum TetrominoType {

    I, J, L, O, S, T, Z;

    private static final TetrominoType[] VALUES = values();
    private static final int SIZE = VALUES.length;
    private static final Random RANDOM = new Random();

    public static TetrominoType random() {
        return VALUES[RANDOM.nextInt(SIZE)];
    }

}
```

If we take a closer look at line 34 in Controller.java we can see that the field factory only accepts instances of TetrominoFactory.

```java
    private TetrominoFactory factory = new TetrominoFactory();
```

This is generally a bad idea, because we can not swap this instance for another factory without inheritance. And because inheritance doesn't make any sense in this situation (i.e this would not be a 'is-a' relation) we would be better off to program this to an interface instead. This interface needs two methods: one for creating a random shape and one for create a shape of type X.

```java
public interface IFactory {
    AbstractShape create(TetrominoType type);
    AbstractShape createRandom();
}
```

A new problem has arisen. Every factory implementing this interface needs to accept an TetrominoType enum in its create method. This means we have to modify TetrominoType and and new values to it. Since enums should be considered as a final set, it would be better to create a new enum. But there is one problem: enums can not be extended like classes. The solution we came up with involves creating another interface for shape types.

```java
public interface ShapeType {}
```

Every enumeration of a shape set should implements this interface. This way we can accept a more general shape type in our factory interface:

```java
public interface IFactory {
    AbstractShape create(ShapeType type);
    AbstractShape createRandom();
}
```

Still there is one problem: ShapeType is not an enum. This means we have to modify the create() method in TetrominoFactory because we can't use the switch statement anymore.

```java
public AbstractShape create(ShapeType type) {
        if(type == TetrominoType.I) {
            return new TetrominoI();
        }
        if(type == TetrominoType.J) {
            return new TetrominoJ();
        }
        if(type == TetrominoType.L) {
            return new TetrominoL();
        }
        if(type == TetrominoType.O) {
            return new TetrominoO();
        }
        if(type == TetrominoType.S) {
            return new TetrominoS();
        }
        if(type == TetrominoType.T) {
            return new TetrominoT();
        }
        if(type == TetrominoType.Z) {
            return new TetrominoZ();
        }

        throw new IllegalArgumentException(String.format("Unknown how to cre
    }
```

We have to admit the switch statement looks better, but this was the final obstacle we had to overcome to make to game as a whole more extensible.

We decided to add a new factory to the game that creates shapes consisting of three minos (blocks). Notice how this factory implements IFactory (the factory interface).

```java
public class TrominoFactory implements IFactory {

    /**
     * This creates one of the shapes defined in TrominoType.
     *
     * @param   type            TetrominoType of Tetromino to create
     * @return  AbstractShape   Tetromino
     */
    public AbstractShape create(ShapeType type) {
        if(type == TrominoType.I) {
            return new TrominoI();
        }
        if(type == TrominoType.J) {
            return new TrominoJ();
        }
        if(type == TrominoType.L) {
            return new TrominoL();
        }

        throw new IllegalArgumentException(String.format("Unknown how to cr
    }

    @Override
    public AbstractShape createRandom() {
        return create(TrominoType.random());
    }
}

```

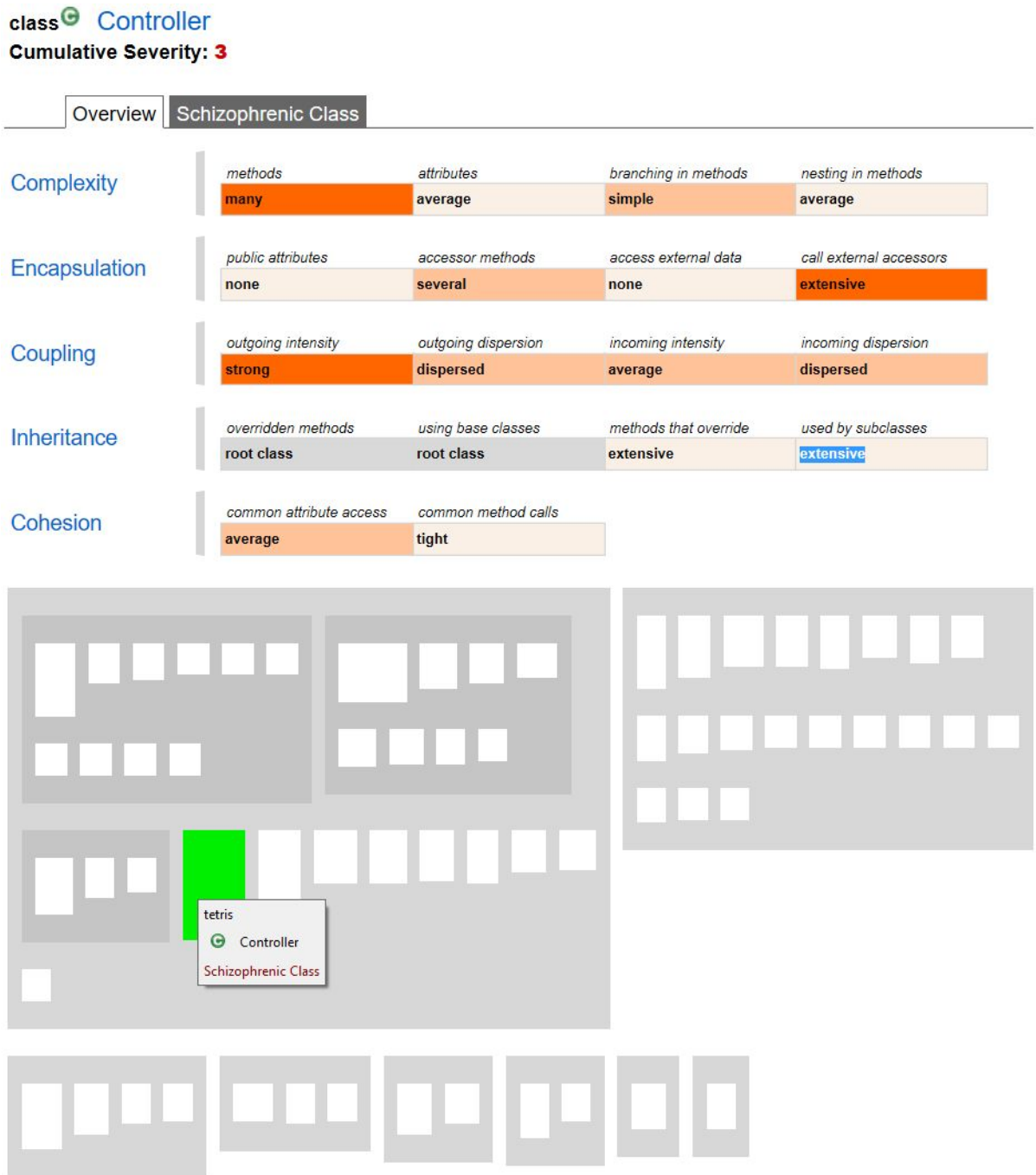
This factory uses the following enum:

```java
public enum TrominoType implements ShapeType {

    I, L, J;

    private static final TrominoType[] VALUES = values();
    private static final int SIZE = VALUES.length;
    private static final Random RANDOM = new Random();

    public static TrominoType random() {
        return VALUES[RANDOM.nextInt(SIZE)];
    }

}
```

The client should now be able to uses the factory of choice.

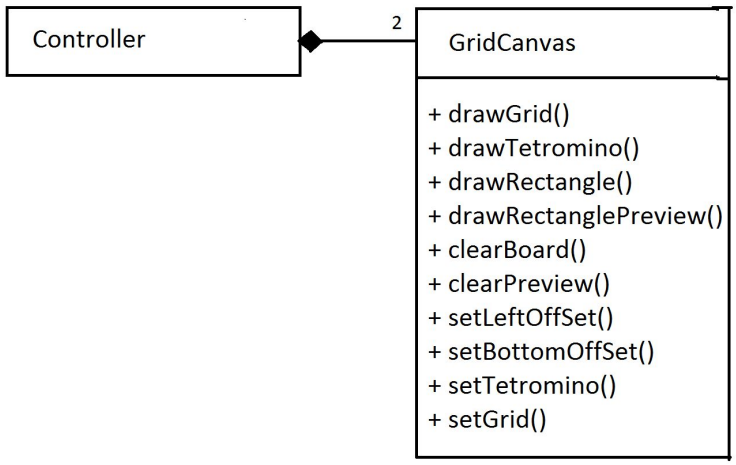
Software metrics

At the start of this week we ran our tetris game through the inCode program to compute the software metrics of our code. We didn't really know what to expect, but the result showed only our controller class was the problem, this was a schizofrenic class. We did already expect the controller to be our biggest problem, because a lot of gets executed in this class with a lot of other classes playing a part in that process. The following results was shown:



The class had a lot of methods, strong outgoing intensity and called a lot of external accessors. Our first initiation was to split the class up, get some methods out of there. This should lower the amount of methods and outgoing intensity. As it would probably still need to call these methods, maybe this wasn't the best plan to lower the call of external accessors.

Our plan was to get the drawing methods out of the controller into their own class: the GridCanvas.



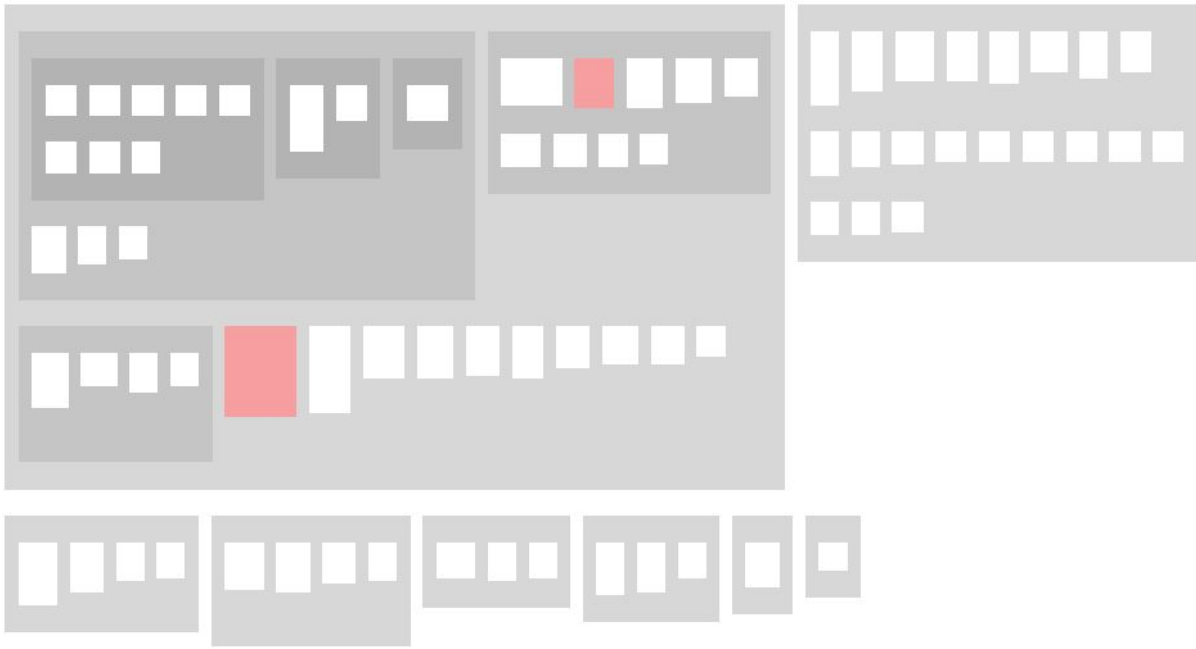
Making the controller smaller is also good for the team. Especially this last week we really noticed that one large class that controls everything is hard to work with alongside each other. It is very easy to get merge conflicts etc.

That being said, our initial idea wasn't really going to work. We had to split up the GridCanvas in the GridCanvas and the GridCanvasPreview. We had our hopes up, until we got the results of the new metrics back:

class Controller
Cumulative Severity: 1

Overview | Schizophrenic Class

Complexity	*methods*	*attributes*	*branching in methods*	*nesting in methods*
	many	**many**	**simple**	**average**
Encapsulation	*public attributes*	*accessor methods*	*access external data*	*call external accessors*
	several	**several**	**none**	**extensive**
Coupling	*outgoing intensity*	*outgoing dispersion*	*incoming intensity*	*incoming dispersion*
	strong	**very dispersed**	**weak**	**dispersed**
Inheritance	*overridden methods*	*using base classes*	*methods that override*	*used by subclasses*
	single class	**single class**	**single class**	**single class**
Cohesion	*common attribute access*	*common method calls*		
	average	**tight**		



This is of course not what we intended to achieve. Because we added these two new couples, this also gave us two new attributes in the controller. This week we also changed things in the music player, adding new attributes to the controller, which then resulted in many attributes given by inCode.

We expected a better result in the amount of methods, we went from 28 to 22, including some new methods also added this week. Apparently this is not enough, but due to lack of time we couldn't remove any more methods.

The coupling also seems to be a big problem in controller. By splitting out the GridCanvas we expected the outgoing coupling to go down and the incoming coupling to go up, to balance it out a little bit. The results actually only show the outgoing coupling going up. This is probably, because the controller calls the GridCanvas and GridCanvasPrev. Our initial thought was that these classes would carry out the methods, called for in the controller, therefore making it incoming coupling.

Unluckily one of the new classes also has cumulative severity 1, but this time it is a data class. InCode tells us this class is showing a significant amount of data in its public interface, either via public attributes or public accessor methods. This, to us, doesn't seem legit, as this class has four private attributes, 3 private methods and 4 public methods. Within these 4 public methods there are three set() methods, which get information from the controller. There are no get() methods to distribute its data. Hereby we can conclude that this design flaw can be ignored for now.

Although we weren't able to get rid of the schizophrenic controller class, we have degraded it from severity 3 to severity 1. Because the controller is so mixed up in all of code, it was hard to refactor it more within the given time. We have now seen what problems can appear by giving this much power to one class. It is very hard to refactor the code and it makes it a lot harder for co-workers to work alongside each other, without getting merge conflicts due to the controller.

class GridCanvasPrev

Cumulative Severity: 1

Overview | Data Class

Complexity

methods	*attributes*	*branching in methods*	*nesting in methods*
average	**average**	**simple**	**average**

Encapsulation

public attributes	*accessor methods*	*access external data*	*call external accessors*
none	**several**	**none**	**extensive**

Coupling

outgoing intensity	*outgoing dispersion*	*incoming intensity*	*incoming dispersion*
weak	**focused**	**weak**	**focused**

Inheritance

overridden methods	*using base classes*	*methods that override*	*used by subclasses*
single class	**single class**	**single class**	**single class**

Cohesion

common attribute access	*common method calls*
tight	**tight**

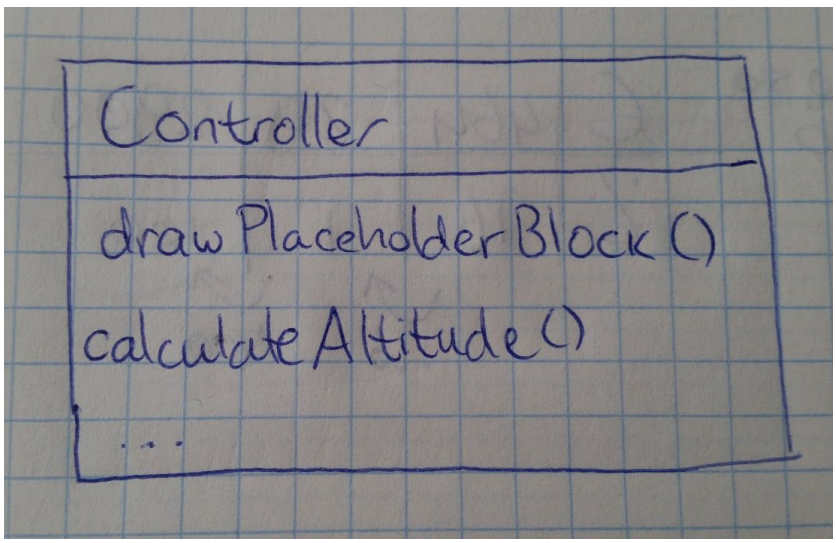
Shape drop shadow

This week we had to implement a new game mode. This game mode would initiate with some blocks randomly placed in the grid. The player wins the game when all preplaced blocks are cleared.

Due to the fact that we spent a lot of time getting the bot working and had to do a lot of refactoring to solve some issues from the InCode report, we were not able to implement this game mode.

Instead we came up with the idea to implement a drop shadow. The time we had left was just enough to implement this and we thought this would greatly improve the user experience.

For this feature we had to add two methods to Controller.java. One drawing a rectangle on the board that looks like a placeholder for the falling shape blocks. And one method that calculates the number of rows a shape can drop (altitude).



To calculate to altitude of the falling shape we had to make a copy of the falling shape. We would just shift this copy until it could not drop any lower. We reused the check mechanism for the falling shape. Then the only thing left to do was implement the method to draw the drop shadow on the board, which was fairly simple because we could reuse most of the code for drawing the falling shape.