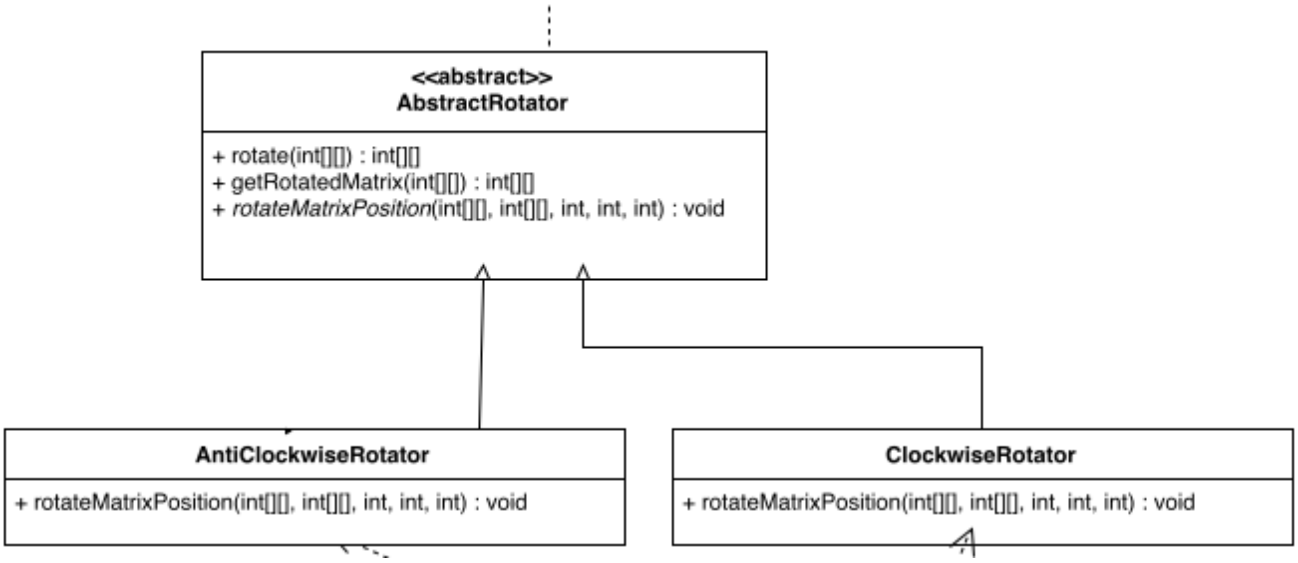
Analysis of Tetris code base

This analysis is divided in two parts, first a rundown of three different code patterns that appear in the code base. After that some potential problems that break OOP principals are identified and possible fixes are suggested.

Design Patterns

Template Method Pattern

The class **AbstractRotator** defines the algorithm for transforming the rows and columns of a 4x4 matrix to what it will be post-rotation. The actual new value of each field in the matrix is calculated by invoking the abstract function *rotateMatrixPosition*. This function is defined in subclasses to provide the actual details of the rotation. The codebase has two such concrete details: one for rotating clockwise (**ClockwiseRotator**) and one for rotating anti-clockwise (**AntiClockwiseRotator**). These are utility classes for applying rotation to 4x4 matrices. This setup is a usage of the template method pattern. See the diagram below.



As for usage. The program has a class **ShapeLayoutToBoardCellMapper** that, as the name suggest, maps a shape to the cells it occupies. It is also responsible for applying rotations, which is where the aforementioned rotators come into play.

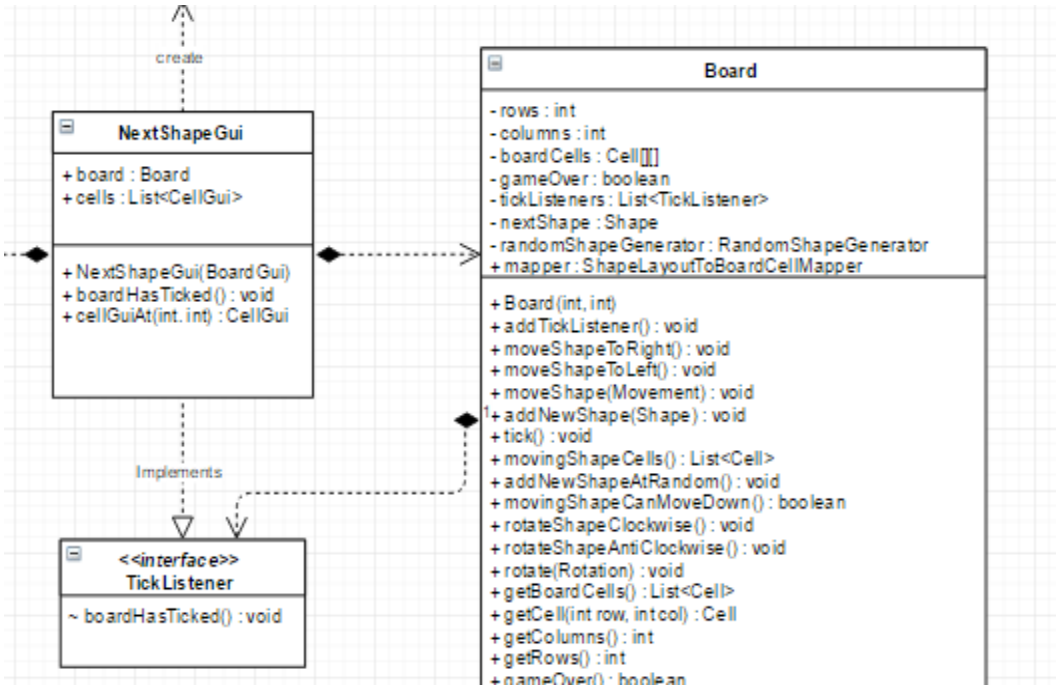
Observer Pattern

Why is it used?

The creator of the project uses the observer pattern to let other parts of the program be notified when the **Board** is updated. This lets the board communicate with the stuff around it without having to know about said stuff.

Where is it used?

The **Board** has a list of **TickListeners** that are invoked whenever the board is updated. **NextShapeGui** is a **TickListener** so when board is updated it will notify the **NextShapeGui**.



Factory Pattern

Where is it used?

In the codebase you have a hierarchy of rotations, in concrete terms you can pick between clockwise and anticlockwise rotations. To do the actual selecting, rather than make objects of the rotational classes, the creator of the project decided to make a **RotatorFactory**, where he applies the factory pattern for concrete object creation.

Why is it used?

Factory pattern makes it easier for any client to create a rotation. If a client was to rotate a matrix directly it would have to create a specific **Rotator**. It then needs to put the matrix into the object.*rotate()*. Therefore, it is very obvious that creating a rotation through a factory is quicker and

easier. In addition, the creator can avoid exposing his implementation. Moreover, factory pattern allows the creator to follow the SOLID principles. In particular, factories and interfaces give more flexibility for decoupling for the project.

Suggested Code Improvements

Possible breakage of OCP

Let's imagine a scenario where we receive an order to expand the Tetris game. The publisher now wants us to make Tetris XL, with a bigger playing field for double the fun. The **Board** class constructor is parameterized on number of rows and columns, so this seems like an easy task. However, along with our bigger playing field it follows naturally to want bigger shapes. To do so we would simply inherit from the **Shape** class and add an entry in the **RandomShapeGenerator**, the factory responsible for making shapes. So far so good, the code seems quite adherent to the Open Closed Principle.

However, the function that returns the structure of the shape, **Shape**::*getLayoutArray*() is notably not abstract. In fact it returns a 4x4 matrix. And any child of **Shape** needs to fill in this 4x4 matrix to describe its shape. So, to make bigger shapes we would have to modify **Shape** to make *getLayoutArray*() an abstract function that lets us have any sized shapes. This also forces us to modify all existing children of the **Shape** class, making this a problematic design from the OCP point of view.

So, having applied OCP to the **Shape** class we expect all to be good and well. Unfortunately, the code while it compiles stops working properly. The reason being that all around the code it does not deal abstractly with the **Shape** instead it assumes that the **Shape's** underlying data is a 4x4 matrix. So rotation code works not on shapes, but 4x4 matrices, placement code (mapping the shape to the board) also works directly with the underlying data.

Clearly this is a pretty big breakage of OCP as well as failing at Data Hiding. The classes that do movement & rotation should possibly work directly with shapes, and return a decorated shape that represents the changed shape. Like this:

```
// We get back a RotatedShapeDecorator where the cells have been rotated
Shape newShape = rotatorFactory.rotate(Rotation.Clockwise, shape);
```

Right now, a **Shape** cannot (correctly) be rotated or changed in any way. So once a shape has been rotated, we need to pass around the resulting raw data (the 2d array of the shape) to all the user code. By applying the decorator pattern here we can keep **Shape** pure while still having the ability to treat a rotated shape as a shape on its own, making it possible to have our methods take shapes and not raw data.

Refactor code using MVC pattern

The code we analyzed doesn't use the Model-View-Controller pattern, how we found out about it is that the project has no separate "controller" class nor "view" class. The GameRunner class contains

both the ”view” and ”controller”. Since the MVC pattern is used for separating applications concerns it would be useful for a Tetris game.

We analyzed the classes and the “view” was quite spread around the classes so GameRunner had both “view” and “controller” in it. Inside GameRunner the **-ShapeMover implements KeyListener** and the methods inside it handles all the user inputs. Since the UI code is spread over GameRunner, NextShapeGui, BoardGui and CellGui we would have to remake quite a lot of the project structure to make it ideal for the MVC pattern.

Our steps to actually make this project to follow the MVC pattern is that we would have to make a “view” class and since the classes for handling the UI is already created, the “view” class could create them and encapsulate them in the “view” class. To break up the relation of the “controller” from GameRunner we create a “controller” class to handle all the user input. The “controller” and “view” classes will communicate through two interfaces one representing each class. Thus the GameRunner will be left with creating the “controller” and “view” class and initiate the game.

Possible problems with SRP

The code breaks the SRP (Single responsibility principal) by having so many responsibilities inside the **ShapeLayoutToBoardCellMapper**. Apparently, this class is responsible for everything that is related to the dropping shape at gameplay (the shape that gets dropped from the top of the board). Its responsibilities include rotating, moving the dropping shape as well as making sure that all movements and rotations are valid.

Also the class is highly coupled with Board. Besides, breaking the SRP the code works directly with data rather than abstractions, making the code unnecessarily complex.

Here is our solution for these issues. We add a rotating functionality to the Shape class with help of the decorator pattern. We add an object of class Shape into the Board class to represent the “*droppingShape*” as well as keeping track of its position. We remove the **ShapeLayoutToBoardCellMapper** meaning the **Board** is now responsible for anything regarding the *droppingShape*. We reduce the complexity of the rotating methods and the moving methods by manipulating cell data in a single place in the **Board**.