Overview

Tetris (Links to an external site.) was arguably the most popular single-player computer game of all time by the early 1990's. It's incredibly easy to learn and fun to play, and even to this day it's widely considered to be among the best video games of all time.

There are some animations in the link above, but basically the game consists of a narrow well, into which randomly-selected geometric 4-unit pieces begin to fall one at a time. The player manipulates each piece as it falls, by moving it horizontally within the well, and rotating it in 90° increments. Once the piece can fall no further, it becomes locked in place, taking up space in the well. However, if the player forms one or more full horizontal rows, those rows are removed, with all blocks above shifting downward by 1 unit. The goal is to continue playing as long as possible, so the player must attempt to pack the pieces strategically. Eventually, though, the well fills upward to a limit line, and if any portion of a piece is locked in place above this line, the game is over.

You'll see all of this in action soon enough, and it's actually quite easy to understand once you've played it for just a few minutes.

Requirements

Building a full-functional Tetris application requires a little more Java than what we've covered yet this term, so you'll be provided a working application that you can download and play. It's a somewhat simplified version, with a very rudimentary scoring mechanism, but it's still almost as enjoyable and challenging as the original.

So, if the application is already provided, what is this assignment all about? The goal here is to develop a basic Artificial Intelligence ("AI") system, capable of controlling the game as if it were the human player.

Implementing an Interface

In previous assignments, you were given a class file with method declarations, Javadoc comments, and placeholder implementations. In this assignment, you'll be given an *interface* instead. An interface contains the method declarations and the Javadoc comments, but no code at all (not even placeholders). Your task is to *implement* this interface, which means you'll develop a class that provides code for all the methods declared in the interface. In addition to the methods specified by the interface, you will likely need to develop several private helper methods in order to fully implement it without redundancies.

System Description

The interface you will implement is called `edu.vt.cs5044.tetris.AI` and your implementation class must be called `edu.vt.cs5044.TetrisAI`. You will need to develop the code needed to implement the methods defined by the interface. One of the methods will be the overall AI system. This method receives some parameters, defining the current state of the game board and the shape to be placed, and will need to find the "best" placement for that shape. The other methods are responsible for intermediate calculations. These methods would normally be private, and not specified by the interface, but for academic purposes we're exposing these methods to ease the testing requirements. Speaking of tests, your JUnit test file must be called `edu.vt.cs5044.TetrisAITest`.

Downloads

tetris5044.jar library file containing all the compiled game engine classes
tetris5044-api.jar library file containing the Javadocs for the game engine

Setting up Eclipse

Download all the files from the links above to your computer, placing them in any convenient folder. Your Eclipse workspace is fine, as long as it's not within any project folder. Open Eclipse and create a new Java Project for this assignment. Do NOT create the **module-info** file, if prompted. If you accidentally created it, or if you're not prompted, just please be sure to delete it now.

Right-click the project and select Build Path | Configure Build Path, then select the Libraries tab. Select "Classpath" then click "Add External JARs" and navigate to where you placed the downloaded JAR files, select `tetris5044.jar`, and click OK. You should now see that file listed in the Classpath section (along with the JRE System Library in the Modulepath section).

Now click the little triangle to expand the new library, select the node called `Javadoc location: (None)`, and click the "Edit..." button. Here be sure to select "`Javadoc in archive`" first, then click Browse. Navigate again, this time selecting the `tetris5044-api.jar` file, and click the "Validate" button. It should tell you the location is "likely valid" so click OK, then OK, to get back to the Libraries tab.

Now select "Classpath" again, and click the "Add Library..." button, select JUnit, then select JUnit version 4 (this version is not the default!) and Finish.

Ensure you see all these libraries now:

- Modulepath
  - JRE System Library [JavaSE-11]
- Classpath
  - tetris5044.jar

        - Javadoc location: *[path to tetris5044-api.jar]*
    - JUnit 4

Note that it's perfectly fine if these appear in a different order within each category, as long as they're all listed in the correct categories. Click "Apply and Close" to accept this build path and return to your project.

Right-click your project name, select New | Source Folder, and name it "test" and then add a New | Java Package called "edu.vt.cs5044" to both source folders ("src" and "test").

Shall We Play a Game?

At this point, you can right-click the project name, and select the Run As | Java Application. Several classes will appear, and you should see `Tetris5044 - edu.vt.cs5044.tetris` listed near the top. However, you may need to type the first few letters to search for this. Select this class, and click Run to play Tetris!

A game window should appear with an empty game board and a prompt to type 'P' to play. Go ahead and try this now, to see the game in action.

(If the application appears to launch, but the game window doesn't appear, please ask in Piazza and make sure to note which OS you're using.)

Once you type P, you will see a new random piece near the top, slowly falling toward the bottom of the well. There are several alternative keyboard controls you can use, but for now just type 'J' and 'L' to move the piece left and right, and type 'I' to rotate the piece. Once the piece is situated as you wish, you can use 'K' (or the space bar) to drop the piece immediately, or you can just enjoy watching the piece fall gently into place. The piece will change color once it's locked into the board, and can no longer be manipulated.

Notice in the console there is a Welcome message. You can type '?' into the game (click the game screen before doing so, to ensure the game receives the keystroke) at any time to produce a listing in the Eclipse console of all of the available options and keyboard controls. Try this now. Some of these features will be extremely useful during the assignment. You'll probably notice some interesting options, including a way to enable the Player AI by typing Ctrl-P. When you enable this mode, just before a new shape appears in the game, the `AI` is asked how to best rotate then position the shape.

Try actually typing Ctrl-P into the game window. If you're in a game, it will end, and you should see a text warning in the console that your AI implementation couldn't be found. That's perfectly understandable at this point.

Back To Work

Close the game for a moment now, so we can get to some coding tasks for a while. First, right-click the `edu.vt.cs5044` package of the "src" source folder, and select New | Class.

Before you enter the class name, click the "Add..." button to the right of the Interfaces section. In the search box at the top, type `AI` and you should soon see the `AI` interface (in the `edu.vt.cs5044.tetris` package) selected. Click Ok to add that interface. For now, un-check the option to create "Inherited abstract methods" (if it was checked by default) then enter the class name as `TetrisAI` and click Finish.

Notice that the new file declaration says `public class TetrisAI implements AI`. This is what tells Java that we intend to be compatible with any system that can work with the `AI` interface. You'll also notice Eclipse is already showing an error on the class declaration line. Click the tiny red X in the margin and double-click "Add unimplemented methods" from the pop-up. This asks Eclipse to generate all the method placeholders for you! (The option you un-checked earlier would have done the same thing; this way you're able to see exactly what it's doing.) Save the file, and check out your new code. It's already compatible with the game system, even though all it does is return placeholder values. That obviously won't meet our requirements, but it's actually plenty to get started. Let's see if it's recognized by the game engine.

Launch the application again, and type Ctrl-P to activate the Player AI. Confirm that the console says the Player Mode is now AI. If so, go ahead and type P to start the game.

Well, that's not very exciting. The console just keeps printing warnings that the AI has selected a null move (which is invalid) for each shape, and we still have to manually place the pieces. That's fair enough, since all we have are placeholders that Eclipse generated for us, but at least we know the game successfully found our AI class, and seems to be communicating with it. Close the game again, so we can code a little more.

NOTE: You must close the game and restart it, each time you make any source code changes.

Not Very Deep Thoughts

Let's at least provide a valid suggestion, even if it's always exactly the same suggestion. In the `findBestPlacement()` method, let's replace the placeholder:

```
  return null;
```

with this:

```
  return new Placement(Rotation.NONE, 0);
```

Eclipse may complain that it doesn't know about the Rotation class yet; if so, just double-click to import these classes from the "edu.vt.cs5044.tetris" package. That should resolve the error, so save your file again. This really doesn't seem like much of an improvement, but we'll try it anyway.

Launch the game and use Ctrl-P then P to check it out. Notice that now it's placing everything against the left edge (column 0) without any rotation. Also notice that the falling shapes are now gray, meaning the placement was done by the AI. They can't be manipulated by the keyboard in this state.

We've taken an important step here. Our AI is actually placing the pieces automatically for us! Right now, the decisions aren't very good, but things can only get better from here. Close the game once you're ready to code again.

Read All About It

Obviously we need to examine the shape and the state of the board in order to make some reasonable decisions. Let's take a look at all those Javadocs that make up the API. In your source file click within the word AI of "implements AI" and type Shift-F2. This should open the Javadocs for the `AI` interface using an internal browser within Eclipse. You can set it to use an external browser, if you prefer, but this is fine for now.

Browse through the Javadocs, to get familiar with the interface, and the classes you'll be using to implement it. For example, click the findBestPlacement() method of the interface, and you'll see it actually provides a brief outline of a strategy we can use. Click the "Package" link at the very top of any Javadoc page to see the package overview, then navigate throughout the various pages to see what each class and enum can do, along with the requirements for the interface. A fundamental part of this assignment is to learn how to use Javadocs to explore new libraries, as well as to see how a somewhat larger scale system is divided into multiple classes, each with its own purpose.

Note that class Tetris5044 is for internal use only; you won't need it at all. Also, classes RandomMode and ShapeStream are only needed for the optional "challenge" section below.

Strategy Games

There's no such thing as a perfect Tetris strategy, so we won't even try. We are necessarily taking the heuristic (Links to an external site.) approach to this problem, so there will necessarily be trade-offs involved. In every placement decision, there are advantages and disadvantages. How can we hope to find anything approaching a "best" placement? Now is probably a good time to look at those other methods you'll need to implement in the `AI` interface.

Development Phase 1: Costs Methods

From the API, the remaining methods of the interface are related to making some measurements to evaluate a board position. The idea here is that we'll compute four distinct "cost" scores, or factors, which tell our system something about how bad (or good) a particular placement might be. We'll need to develop those methods first, before we can hope to have a working AI system.

We're still focusing on TDD, so start with some fairly straightforward test cases to ensure the individual scoring methods work. Just construct a test board object, then assert the cost score you expect your implementation to compute. You'll need to study the API for the Board class in order to construct a Board with an arbitrary arrangement blocks for testing purposes.

Note that you must complete Phase 1 before attempting Phase 2. Phase 2 would be meaningless without a working Phase 1.

Development Phase 2: Searching for the Best

Once all of your individual cost scoring methods are working properly as expected, only then will it be time to put it all together. The findBestPlacement() method will need to iterate through all the possible placements, and for each it will combine these individual cost scores in a weighted sum, then choose the lowest cost to return to the game engine. The weightings are important so that certain factors can have more influence on the overall decision than others. At first you can just set all the weights to 1, by simply summing the individual scores together, even though eventually we'll need to adjust the weights to achieve better results.

Strategic Limitations

Luck -- in the form of randomness -- plays a starring role in Tetris. Even with the best of strategies, it's critical to recognize that some games simply provide a more fortunate sequence of shapes than others. Thus it can be difficult to objectively judge exactly how good a particular strategy might be. The same strategies that do very well in some games may do very poorly in others. To help with this, our implementation provides a set of 4 repeatable "TEST" sequences we can use as a benchmark. The average number of pieces placed from these TEST sequences will act as a very reasonable metric of how well our strategy is working overall.

Teaching to the Test

Eventually we'll be making a linear combination of our four scores, which just means we multiply each score by some weight, then add the weighted scores. The equation is very straightforward:

overallCost = weight1*score1 + weight2*score2 + weight3*score3 + weight4*score4

Computing the individual scores really isn't too bad, but what about the weights? As noted above, we can start by simply setting all the weights to 1, and indeed that strategy should place exactly **61.5** pieces, averaged over the four provided TEST sequences. Individually, the scores are 67, 73, 58, and 48.

Standards of Learning

Can we do better? Sure! As a start, reasonable weight ranges for the selected factors will be 0 through 12, in increments of 4, meaning you should use combinations of weight values of 0, 4, 8, and 12 for each cost. You can just use trial-and-error if you wish; watch your AI play the test games and try to figure out which factors need more or less emphasis. For example, if your AI is creating too many unnecessary gaps, try to increase the weight of the gap count score. Be sure to use Turbo mode to speed up the testing process.

To earn full credit, your AI must be able to place at least **100** pieces, averaged across the provided TEST sequences. This is an improvement factor of over 1½ times the performance of setting all weights to 1. To achieve this, manually adjust the weights (using combinations of 0, 4, 8, and 12 for each cost). Of all these possible combinations, about 20% will result in an acceptable level of performance, so even just guessing the weights will eventually yield a suitable combination.

Please note that you can't (unless you've completed the optional challenge below) develop a JUnit test to verify the level of performance of your weight combinations. Instead, you must confirm this by manually, by actually running the game in all four TEST modes (using Turbo mode to greatly speed up the process) and making note of the displayed scores for each mode. Please do NOT rely upon Web-CAT for this; if you need help, please ask in Piazza!

Fuller than Full Coverage

Unlike in the previous assignment, creating JUnit tests to cover all of your code will be very straightforward, since there aren't very many branches involved in the solution. You can likely achieve full coverage with just one well-designed test case that asserts something about each method. As a result, complete coverage is not nearly enough to ensure you have correctly implemented your code.

Therefore, you must **generate at least 5 test cases, with reasonably complex boards**, for each interface method. You can satisfy this by creating at least 5 distinct test boards which you will simply share among assertions involving all the cost methods. You may find it useful to create a separate set of 5 test boards to test your findBestPlacement() method.

The test boards are expected cover a reasonably wide range of board layouts. You're also encouraged to generate several simple test boards, to exercise cases like an empty board or an almost empty board. These simple boards don't count toward your required 5 test cases mentioned above. Note that Web-CAT won't (and can't) enforce this requirement, but the human grader will definitely be looking for this.

Functional Decomposition

Judicious use of helper methods can significantly reduce the overall amount of code you need to develop. Your code will be inspected during the human review for redundancies, so be sure to **develop helper methods** as appropriate.

OPTIONAL: Automating the Automation

This section is presented as a challenge that is purely optional, and will NOT affect your score in any way.

You might have noticed that it got bit tedious to try lots of combination of 0, 4, 8, and 12, for each of the weights, even in Turbo mode. You might have also noticed that you've been given sufficient tools to run entire simulated games on your own, without any user interface at all!

As a challenge, see if you can identify the best combination of 0, 4, 8, and 12 weights, by writing a separate class file called `edu.vt.cs5044.WeightFinder` with its own main() method. This file will be completely ignored by Web-CAT, and you don't need to generate any tests for it at all. You'll also need to add a public method, such as setWeights(int w1, int w2, int w3, int w4) to your `TetrisAI` implementation, to apply the various weight combinations. This new method will need to be exercised by at least one assertion, just to be sure it's covered within your own JUnit tests. The best weights among these combinations will place an average of exactly **871** pieces for the TEST sequences. That's quite an improvement, unless you happened to get *very* lucky in your trial and error phase!

This process shouldn't take more than several seconds or so to run, but of course the time depends on the system on which it's running. Once you've found the optimum combination, you can hard-code those values as defaults into your `TetrisAI` as the defaults in order to watch them in action. You can just call your own setWeights() method from the constructor. Confirm that this is all working as expected, and that you're definitely placing the expected average number of pieces over the TEST sequences, before the next part of the challenge.

It should then be a fairly simple change to your code to try stepping by 3 instead of stepping by 4. This means you should test combinations of 0, 3, 6, 9, and 12, for every weight. This process takes somewhat longer to run than

stepping by 4. You can expect something closer to a minute, so you might want to println() some intermediate results to ensure everything is going well. If everything is working properly, you'll only need to do this once, and you'll discover that your newly-tuned weights will average almost **2,200** pieces placed for the TEST sequences. That's over 2½ times our best score found when stepping by 4, so we'll call that a definite win!

Still not satisfied? Well, you can always try stepping by 1 instead, meaning all possible weight combinations of 0 through 12. Again it's a trivial code change at this point, but note that this process will take even longer to run, so we're likely talking about dozens of minutes, possibly up to an hour. Also, we're reaching a point of diminishing returns, so this won't be such a dramatic improvement. However, you'll essentially improve your high score by another factor of 2½ times, as your newly-optimized AI will possess the astonishing capability of averaging just over **5,500** pieces placed for the TEST sequences! (By the way, to save yourself some processing time, you can just stop the iterations when you reach this goal; there's nothing better to find, even when stepping by ones.)

Use these optimized weights in your solution, and watch this particular combination in action via the graphical interface. It's quite interesting to compare the variation in results between the worst and the best, as these scores differ by a factor of 40.