Institute of Technology, University of Washington Tacoma
Autumn 2017 TCSS305 - Programming Practicum
Project Tetris
Value: 11% of the course grade

Due:

- **Milestone A** (View – initial incomplete version) (6%):
  due Friday, 1 December 2017, 23:59:59
  Code to play a single game using a graphical user interface.
- **Milestone B** (Final) (5%):
  due Friday, 8 December 2017, 23:59:59
  Sophisticated multi-game GUI with all features completed.

This assignment is a 2-part project designed to test your overall understanding of the concepts covered in the course, including object-oriented design, graphical user interfaces, events, and animation.

Game Description:

For this project, you will write a set of classes to play a graphical game of Tetris. Tetris is a game in which the player moves and rotates dropping pieces so that they form solid lines. Completely filled lines are cleared, giving points to the player. The Wikipedia page for Tetris is a very good source of information about the various incarnations of the game, if you have questions. There are many free versions of Tetris online that you could look at such as http://www.tetrislive.com/

Game Pieces:

There are seven pieces in standard Tetris, typically referred to by letter (picture from Wikipedia):

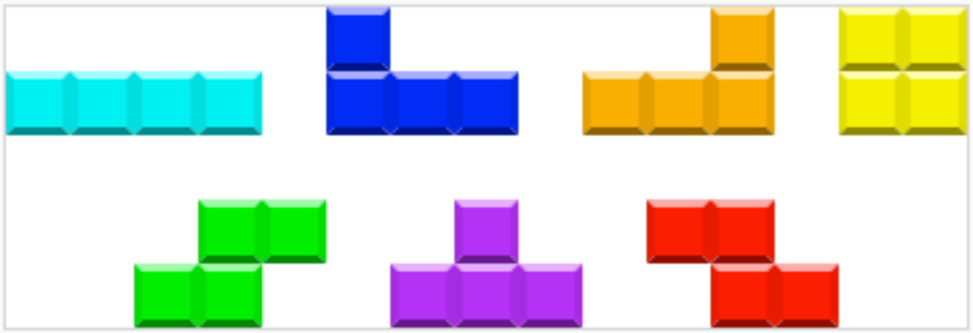

The seven possible *Tetris* pieces in their *Tetris Worlds* colors. Top row: *I, J, L, O*. Bottom row: *S, T, Z*.

Figure 1

Each standard piece is composed of four blocks. The "L" and "J", and "S" and "Z", pieces are mirror images of each other, but we'll just think of them as similar but distinct pieces.

Our abstraction will be that a MovableTetrisPiece object represents a single Tetris piece with position and rotational state.

A piece can be rotated 90° clockwise or counter-clockwise. Enough rotations get you back to the original orientation. Each Tetris piece has one (the O), or four (the L, J, T, I, S and Z) distinct rotational states as shown on the next page. At the top of the diagram notice the increasing series of numbers 0 – 3 which represent the various rotational states. New pieces are initialized to the 0 state. Clockwise rotations increment the rotational state; counter-clockwise rotations decrement the rotational state.

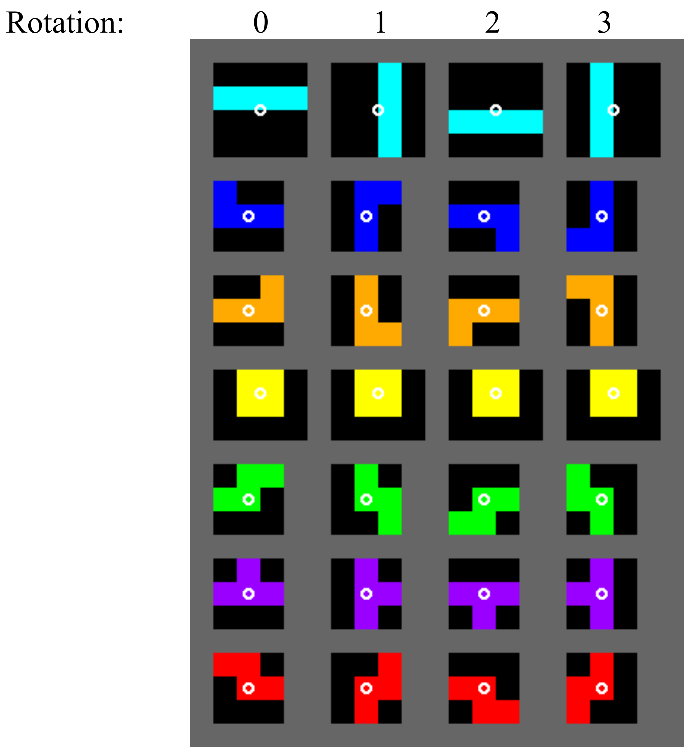
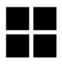


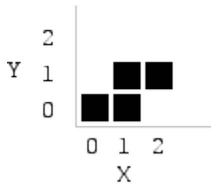
Figure 2

A piece is represented by the coordinates of its blocks, which we can refer to as the "body" of the piece. Each piece has its own coordinate system with its (0, 0) origin in the lower left of the rectangle that encloses the body as shown in figure 2. The coordinates of blocks in the body are relative to the origin of the piece. So, the four blocks of the O piece are:



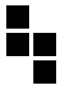
```
(1,1) - the lower left-hand block
(1,2) - the upper left-hand block
(2,1) - the lower right-hand block
(2,2) - the upper right-hand block
```

Here is another example: Rotation 2 of the S piece, with the axes drawn explicitly:



```
[(0,0), (1,0), (1,1), (2,1)]
```

Note that not all pieces will actually have a block at (0, 0). For example, the next rotation of the S has the following body:



```
[(0,1), (0,2), (1,0), (1,1)]
```

Provided Code (Tetris Model):

The provided Tetris model is a set of classes to represent all game pieces and the game board. These classes are fully functional.

If you choose to make ***any*** changes and/or additions to these classes, then the following requirements apply:

1) Add a non-javadoc (implementation) comment to the code wherever you make changes. These comments must explain the purpose of ***every*** change that you make to the provided code.

2) Provide complete javadoc comments for ***every*** new interface, class, field, constant, or method which you add to the provided code.

3) Additionally, LIST and briefly describe ***every*** change and/or addition you have made to the provided code in your executive summary. (Provide a **numbered list** of changes in your summary.)

NOTE: You will **NOT** need to make any changes to the model code to implement a fully functional Tetris game; however, for some types of part B extra credit it is possible that you may need to make minor changes and/or additions to the `Board` class.

NOTE: The piece classes and the `Board` class serve a specific function. The abstraction they define is the *geometry* of the game. You should **NOT** add code to these classes which represents some other abstraction, such as the information tracked by the current game for scoring purposes (the score, the level, the total number of lines cleared in a game, etc.). Information needed by the game which is not related to the geometry of the game should be stored and manipulated elsewhere. The model code (pieces and `Board`) should not contain ANY code to implement GUI, colors, graphics, or sound.

NOTE: No new classes should be added to the model package or to the 'default' package. All of your GUI classes should be in the view package, or in other named packages.

Tetris Pieces:

A piece keeps track of its position on the `Board` and rotational state. A piece has methods to move in various directions (left, right, down, rotate clockwise, rotate counter-clockwise).

A piece has a local coordinate system, as described above; rotating the piece means changing the locations of its 4 blocks in the *local* coordinate system, and moving a piece means changing its position on the board in the *board's* coordinate system. A piece does not make any assumptions about the board it will be placed on. In particular, it does not assume any particular board position is valid or invalid, because it is the board's job to determine whether a piece is in a legal position. A piece is printable via a `toString()` method, which generates a 2D text drawing which clearly depicts the rotational state of the piece. This is useful for testing.

Tetris Board:

The project includes a class called `Board` which is a representation of a Tetris game board. The board is essentially a grid of squares that may be filled with previously dropped pieces (and remnants thereof). The standard Tetris board is 10 squares wide and 20 squares tall, but can be initialized with arbitrary dimensions.

The `Board` class contains the following data and functionality:

- A representation of all filled squares.
- A current piece-in-progress, which conceptually starts *above* the middle of the board; this means that the piece does *not* occupy any of the squares on the board when in its starting position. (The piece will not be visible in the GUI when it is first instantiated.)
- Methods to move the current piece left, right, and down and to rotate the current piece while doing collision detection – that is, if the method to move the current piece left is called, and the current piece cannot move left (because that would cause it to collide with a previously frozen piece, or move off the board), the current piece does not move. Rotations near walls will cause the piece to offset enough to allow the rotation when necessary (wall kicks).
- The ability to provide a specific sequence of pieces to use – by default the `Board` uses randomly generated pieces (see below).
- The `step()` method updates the game state by one step. Calling `step()` causes the current piece-in-progress to move down by one row, and may cause additional changes to the board as follows:
    - when the piece-in-progress is resting on top of existing filled squares or reaches the bottom row of the `Board` and attempts to move down one more time, it freezes and its blocks become part of the board's grid of filled squares
    - when the piece-in-progress freezes, if any lines are completely filled, they are cleared from the board; all squares above a filled line drop downward by one line (note that up to 4 lines may be cleared at a time)
    - when the piece-in-progress freezes, a new piece-in-progress is chosen either randomly or from a predetermined sequence of pieces.

There are multiple reasons for the ability to specify whether the board uses random pieces or a specific piece sequence. The ability to specify a sequence of pieces is helpful for testing; it allows you to drop pieces in a specific order, onto the board to see that they give correct board layouts as they drop. The ability to specify a sequence of pieces can also be useful for gameplay features because it gives you the ability to "replay" games, or to always use the same sequence of pieces (as in tournament play). Clearly, the ability to randomly generate pieces is important in standard gameplay, as it makes the game unpredictable for players.

The game is advanced externally, by using a timer to call the `step()` method.

The `Board` class extends the `Observable` class, so that the GUI you write in Milestone A will be notified of changes that occur on the game board. The `Board` notifies observers of changes whenever a piece moves, rotates, or lines are cleared (and does *not* notify observers when a piece attempts to move left or right, or rotate, but does not change position because of bounds checking).

The `Board` is printable using a `toString()` method. The `toString()` produces a 2D text drawing of the entire board, including all fixed filled squares and the current piece in progress. Do **NOT** use the `toString()` to determine the geometry of the `Board` for your GUI. `toString()` is for testing/debugging purposes only.

Milestone A (Initial Graphical User Interface):

The first version of your code is an initial graphical user interface (GUI) for the Tetris game. The GUI must contain the following:

- a graphical representation of a 10 x 20 Tetris game board, showing its filled squares and its piece-in-progress
- a graphical display of the "next" piece that will come after the current piece-in-progress. This piece preview must show the "next" piece in the same rotational state in which it will appear on the board. The piece preview must show the "next" piece horizontally and vertically centered in a panel.
- the ability for the user to move the piece in progress left, right, down 1 line, rotate it clockwise, and ability to instantly drop a piece downward to the bottom using keyboard keys. You must encode the following default key controls:

Move Left	left arrow and 'a' and 'A'	(All 3 must work)
Move Right	right arrow and 'd' and 'D'	(All 3 must work)
Rotate	up arrow and 'w' and 'W'	(All 3 must work)
Move Down	down arrow and 's' and 'S'	(All 3 must work)
Drop	space	

- some visual indication (help window, display in the main window, *etc.*) of the keys that control the game. The user must be able to access this information during the game.
- an animation timer that makes the game board update at least once per second, causing the current piece to fall.
- some message or indication that the game is over, displayed when the game ends (that is, when frozen pieces extend above the top of the board)
- the GUI MUST implement Observer to complete the observer design pattern for communication between the GUI and the board. Neither your Timer nor your KeyListener should call repaint(). Instead, they should call methods of the `Board` class, which will, in turn, notify the GUI after updating the model.

Your game should not lock up (due to an infinite loop or memory overflow), produce console output, or throw exceptions (check the console window in Eclipse).

5% extra credit can be earned in this phase **for giving the user the ability to customize the keys** that are used for moving, rotating, and dropping pieces. Regardless of whether or not you provide this customization ability, you *must* have some visual indication in your GUI of the keys that control your game. If you do implement this extra credit, then whenever a user customizes the assigned keys, the visual indication of which keys are in use should also update. It may also be useful for you to read online about key bindings.

5% extra credit can be earned in this phase **for giving the GUI the ability to resize gracefully**, within reasonable limits (you can, if you choose, use `WindowEvents` to enforce these limits; however, that is not necessary for the extra credit). Providing a selection of at least 3 fixed sizes for the user to choose from can earn up to 4% extra credit. To earn the full 5% extra credit the GUI must be resizable by dragging a corner. (Resizing gracefully, in part, means that squares in the graphical representation of the Tetris game board remain square as the GUI is resized. The squares should not become elongated, either horizontally or vertically, into rectangles. Resizing gracefully also means that the squares in the graphical representation of the Tetris game board should resize. The squares should not remain at a fixed size if the GUI is resized.)

NOTE: If you choose not to pursue this extra credit then select a fixed size for your GUI that cannot be changed.

NOTE: There is a section in the executive summary template for you to discuss any extra credit that you did for this project. If you did not implement any extra credit then just enter 'none' in that section of the summary.

Milestone B (Final GUI and Extra Features):

The second version is the final, polished version of your Tetris game. You will have an opportunity to demonstrate your code in class, if you choose, before it is due, which is a good way to receive feedback.

Your final submission should have the following new features:

- an option to **pause** the game, which suspends the timer and prevents the user from moving/rotating the current piece. The ability to pause and to un-pause the game must be accessible through key command(s) which do not require opening a menu.
- the ability to play multiple games; that is:
  - The user can **end a game** (The GUI continues to display the state of the game at the point it was ended. The GUI displays some indication that the game is over. The timer does not continue to animate the current piece and keyboard commands do not continue to control the current piece.)
  - The user can **start a new game** when no game is in progress (The GUI does not allow the user to start a new game while another game is in progress. The user must end the current game before starting a new game.) The user must use this feature to start all games including the first game.
- a difficulty level that causes the game to animate faster at higher game levels. Your game should start at level 1 and increment the level for each cumulative five lines cleared. (0-4 lines cleared = level 1; 5-9 lines cleared = level 2; 10-14 lines cleared = level 3; etc.)
  - implement this so that the number of lines needed to reach the next level is located in a single statement. *In your executive summary, state what class and line number where this single statement can be found.*
- a score that is updated as the game is played. The scoring algorithm that you are ***required*** to implement is one similar to the NES version of Tetris, that is:

  Add 4 points to the score when a piece freezes in place.
  Also add points to the score when lines are cleared as show below:

	1 line	2 lines	3 lines	4 lines cleared
Level 1:	40	100	300	1200
Level 2:	80	200	600	2400
Level 3:	120	300	900	3600
...				
Level 10:	400	1000	3000	12000
In General				
Level *n:*	40 * (*n*)	100 * (*n*)	300 * (*n*)	1200 * (*n*)

  - Your GUI must in some way communicate your scoring algorithm to the user. This could be in a dialog at the start of the game, in a 'help' menu, or in some other fashion. The method of presenting this information to the user is up to you.
  - Implement your scoring algorithm in a single unit (class/method). *Include the location of this unit in your executive summary.*
- a display of:
  - the current score
  - the current total number of lines cleared during the game
  - the current level
  - an indication to the user of when the next level will be reached

In addition, your Part B game **MUST have *at least one* special feature**. Implementation of **more than one special** feature can earn up to 10% extra credit, depending on the impressiveness of the extra feature(s). NOTE: Your first special feature is NOT extra credit, it is REQUIRED. The special feature(s) can be anything you want. Your executive summary **MUST** include a **numbered list** of the special feature(s) you have implemented. Below are some examples of special features. I am most interested in UNIQUE features which you think up yourself. I will award extra credit based on the difficulty, quality, and creativeness of your extra features. Extra credit will NOT be awarded in milestone B for milestone A extra credit features.

Some examples of typical special features	Max Extra Credit
sound effects and/or background music	5
multiplayer option (competitive or collaborative) (two sets of keys to control pieces on two boards simultaneously, for example)	4
ability to save/load a game	4
ability to show an instant replay of a game	3
ability to save high scores	2
ability for the user to specify or select a game board size other than 10 x 20	2
for also submitting an executable jar version of your game in your project (place the jar file in your project and commit it to SVN, then mention it in your summary)	1
ability for the user to select a specific difficulty level	1
'ghost' piece or 'projection' piece	1
a 'hold' piece	1
gameplay variants of various kinds	credit varies
ability for the user to customize aspects of the GUI	credit varies
visual enhancements to the GUI	credit varies

Submission and Grading:

Create your Eclipse project by downloading the `tetris-project.zip` file from the Tetris Part A page on Canvas, importing it into your workspace, and using "Refactor" to change "username" to your UWNetID. Remember to make this change *before* you first commit the project to Subversion.

You must check your Eclipse project into Subversion (following the instructions from Lecture 1 and Assignment 0), including all configuration files that were supplied with it (even if you have not changed them from the distributed ones). If you use any supplementary files (images, music, sound effects), those must be committed as part of your Eclipse project. If you have any questions about this, ask the instructor as soon as possible. When you have committed the revision of your code you wish to submit, make a note of its Subversion revision number. To get the revision number, *after* you commit your code, *perform an update* on the top level of your project; the revision number will then be displayed next to the project name. For each milestone, you should keep working on *the same Eclipse project* – do *not* create multiple Tetris projects.

After checking your project into Subversion, for each part of the project, you must submit (on Canvas) an *executive summary* containing the Subversion revision number of your submission, an "assignment overview" (1 paragraph, up to about 250 words) explaining what you understand to be the purpose and scope of the assignment, and a "technical impression" section (1-2 paragraphs, about 200-500 words) describing your experiences while carrying out the assignment (especially any difficulties you had using the tools). The assignment overview shows how well you understand the assignment; the technical impression section helps to determine what parts of the assignment need clarification, improvement, *etc.* for the future, and also identifies areas that potentially need more coverage in class. **Your summary must also include a list of all changes / additions you made to the provided starter code and a list of any extra credit features implemented in the**

project. On the Canvas submission page for this assignment there is a new version of the executive summary template which includes these two new required sections.

The filename for your executive summary must be "**`username-tetris-a.txt`**" for part A and "**`username-tetris-b.txt`**" for part B, where `username` is your UWNetID. for example, the filename for the instructor's submission of Part A would be "`acfowler-tetris-a.txt`". An executive summary template, which you *must* use, is available on Canvas. In particular, your executive summary must have a line "Subversion Revision Number: #", with no leading spaces, where "#" is the Subversion revision number you made a note of above (with no parentheses or other symbols). Executive summaries without a line following this exact format will be penalized. Executive summaries will *only* be accepted in plain text format – other file formats (RTF, Microsoft Word, Adobe PDF, Apple Pages) are *not* acceptable.

Part of your project's score will come from its "external correctness." External correctness will be evaluated by running your GUI and playing the game to see how it behaves in various respects.

Another part of your project's score will come from its "internal correctness." Internal correctness includes meaningful and systematically assigned identifier names, proper encapsulation, avoidance of redundancy, good choices of data representation, use of comments on particularly complex code sections, and inclusion of file header comments.

NOTE: For this project I expect you to use proper packaging of the source code. No source code should be in the "default" package. Instead, all source code should be placed in meaningfully named packages. The project is supplied with a "model" package with all classes to represent the Tetris pieces and board. There is also an empty "view" package to hold the GUI code that you will write. If you choose to create additional packages, that is acceptable, but not necessary.

Internal correctness also includes whether your source code follows the stylistic guidelines discussed in class. This includes criteria such as the presence of Javadoc comments on *every* method and field (even private ones!), the use of variable names, spacing, indentation, and bracket placement specified in the class coding standard, and the absence of certain common coding errors that can be detected by the tools. It is therefore to your advantage to be sure the plugin tools like your code before you submit it.

The percentage breakdown for milestones A and B of the project is 10% executive summary, 55% external correctness, and 35% internal correctness.

Academic Integrity Reminder:

Tetris is a popular game, and if you search the web for Tetris Java code, you will certainly find some. You may also know other students who have taken this class previously and implemented their own Tetris code. Remember that you must turn in *your own work* for every phase of this project. The instructor is well aware of the Tetris code available on the web. The instructor also has an archive of the vast majority of Tetris code that has ever been submitted for TCSS305, which can be used for comparison to submitted code. Projects that include code from the web or other sources—except as specifically permitted by the instructor, such as provided sound code—will be treated as violations of the course academic integrity policy. If you are having trouble completing a phase of this project, please contact the instructor as soon as possible.

NOTE: If your Tetris project uses any graphics, images, sound files, or other resources which you did not create yourself, list the sources for these resources in your executive summary and display credit for these resources in the GUI (An 'About' dialog is one way to do this). If you create such resources yourself, discuss that process in your executive summary.