Assignment 5: Flipping the Script – Reversi, part 1

Due dates:

- **Implementation: Tuesday, Oct 31 at 8:59pm**
- **Self-evaluation: Wednesday, Nov 1 at 11:59pm**

This assignment is to be completed with a partner. You must be signed up with a partner on Handins from this assi[illegible] be able to submit this and subsequent assignments until you are part of a t[illegible] know (or remember) how to request teams, follow these instructions. [illegible] than the start of homework 5, which is Oct 20 — if you have not requested [illegible] randomly assign you one. You only need to request a partner for Assignmen[illegible] the remaining assignments.

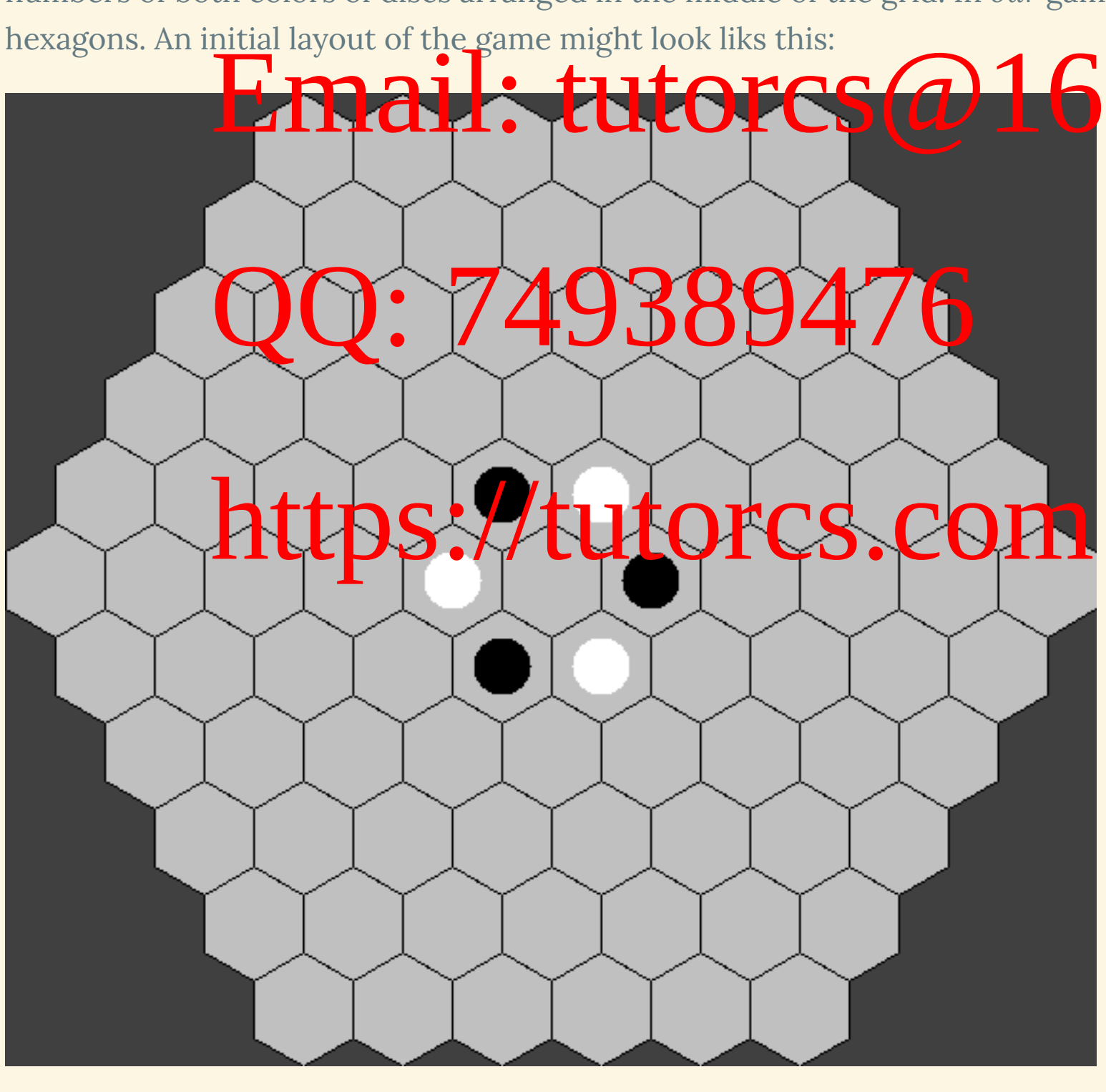
Note that you [illegible] partner to submit the self-eval. As with the code, one partner will submit a s[illegible] However, you will likely need to sit with your partner or be on the phone as y[illegible] ponses.

1 Purpose

In this project, you will be building a two-player game with a graphical interface. The design and implementation of this game is left open-ended, for you to figure out, explain, and justify. There will be several components to this project, so do not assume that *everything* in your code must neatly be described as either “model”, “view”, or “controller”.

2 Reversi gameplay

Reversi is a two-player game played on a regular grid of cells. Each player has a color—black or white—and the game pieces are discs colored black on one side and white on the other. Game play begins with equal numbers of both colors of discs arranged in the middle of the grid. In *our* game, our grid will be made up of hexagons. An initial layout of the game might look liks this:

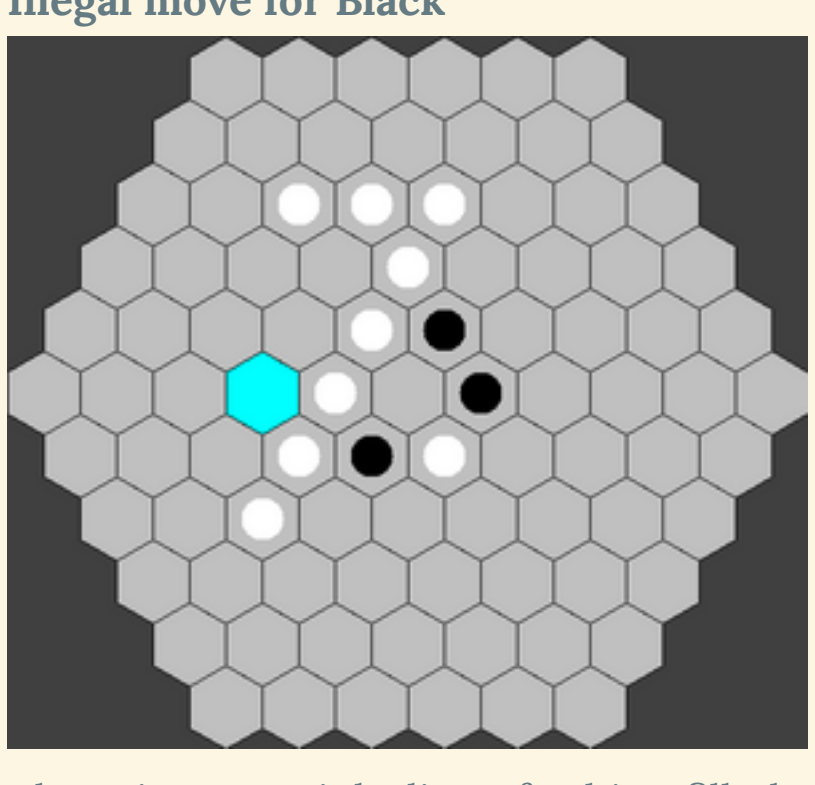


Player Black moves first. On each turn, a player may do one of two things:

- They may *pass*, and let the other player move.
- They may select a legal empty cell and *play* a disc in that cell.

A play is legal for player A if the disc being played is adjacent[1] (in at least one direction) to a straight line of the opponent player B’s discs, at the far end of which is another player A disc. The result of a legal move is that *all* of player B’s discs in *all* directions that are sandwiched between two discs of player A get *flipped* to player A. We say that player A has *captured* player B’s discs.

Illegal move for Black

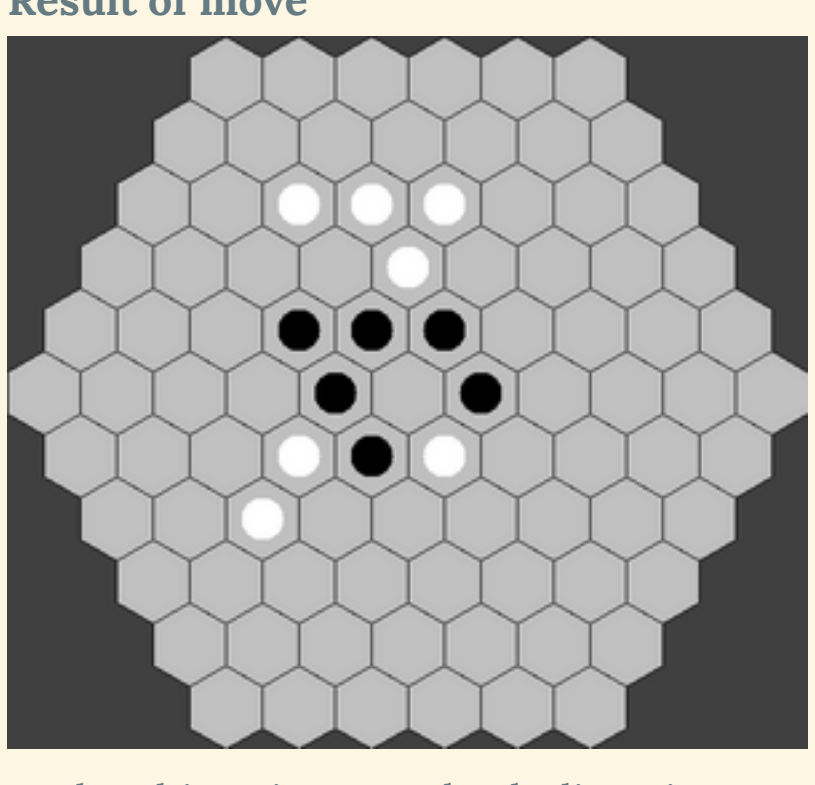


There is no straight line of White-filled cells starting from the blue highlight, that is followed by a Black-filled cell

Legal move for Black

The blue cell is legal in two directions, to the right and to the lower-right

Result of move

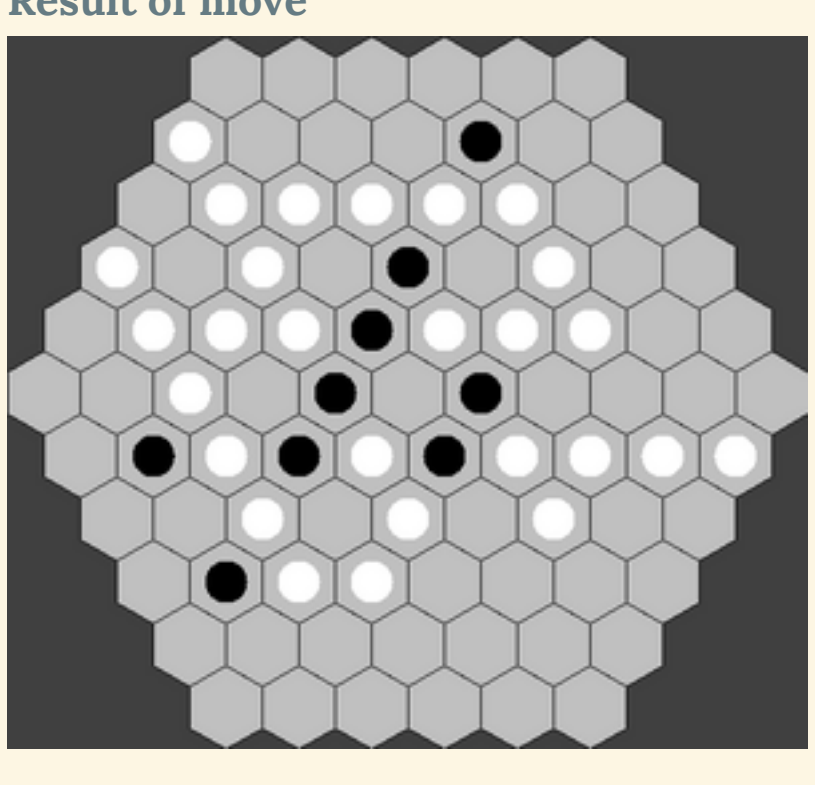


Both White pieces on both directions have been captured

Note that it’s possible to have a situation where player A has two discs, with an empty cell between them, and player B plays into that cell: in that case, player A *does not capture* player B’s disc For example, here it is White’s turn, and even though their selected cell is between two of Black’s pieces, they do not lose the piece:

Legal move for White

Result of move



The same would be true if Black’s pieces were farther apart, with a nearly-complete line of White’s pieces between them, and White filled in the last remaining gap in the line.

If a player has no legal moves, they are required to pass.

If both players pass in a row, the game ends.

3 Architectural choices

A multiplayer game involves several interacting components: you have multiple *players*, that each *interact* with a *visualization* of the *board*, along with a *rules-keeper* to ensure the game is played legally.

In our setting, the board and the rules-keeper comprise our model: together, they are what distinguish one instance of the game from another.

There are multiple ways to view the game; we will discuss two below.

Depending on how the game is viewed, we might want to design different controllers for the game.

And then there are the players, which are not part of the model, view, or controller. Players make decisions about what move they want to make, so they “have a different interface” than the other components discussed so far.

3.1 Modeling the game

You will need to represent the board and its contents. You likely will have to figure out how to represent the coordinate system of the board, so as to describe the locations of all the cells. This is an excellent description of various ways to represent hexagonal coordinate systems — and, in the next assignment, it will be an excellent resource in figuring out how to render hexagons to the screen. Looking at the screenshots of the game above, you will be using the Pointy-Top orientation of a hex grid, and you will need to choose one of the coordinate systems described in that article.

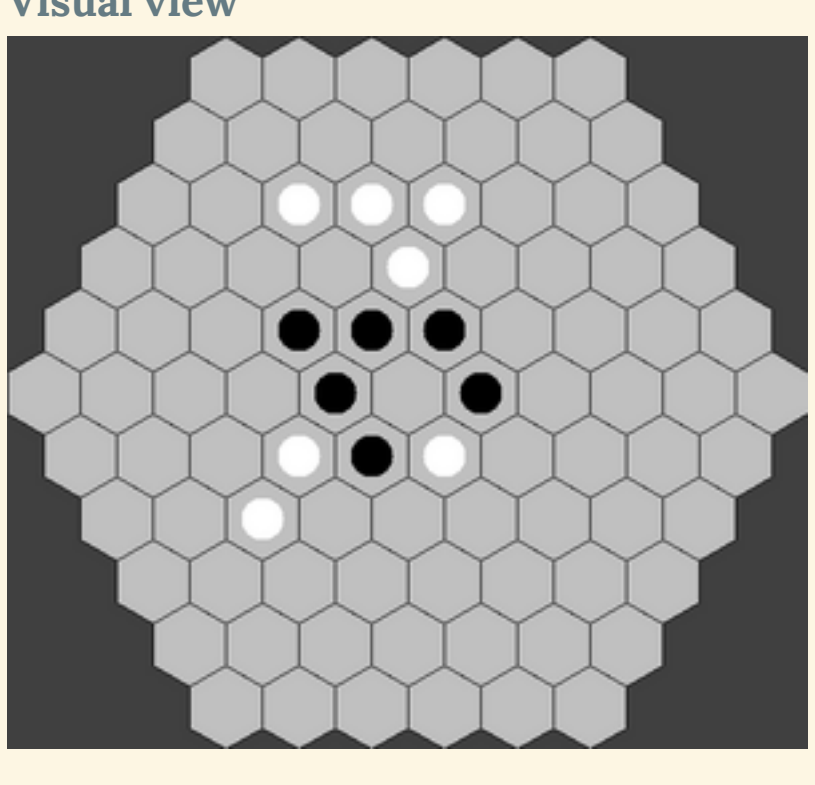
You will need to figure out how to let players make moves. Your implementation will need to (among other tasks) enforce the rules of the game, to make sure that players take turns, that moves are legal, that discs are flipped, and that the winner of the game can be determined.

Hint: your primary model interface may not be all that complicated. As with all our design tasks so far, consider all the nouns you notice in this game’s description, and all the verbs describing their interactions, observations, or mutations, and those become your interfaces and methods.

3.2 Visualizing the game

You are *not required in this assignment* to create a GUI view of your game. Instead, you will start with a simpler textual view, similar to the previous project, to make it easier to see the interim progress of your game. We recommend a straightforward rendering using `_` for empty cells, `X` for one player, and `O` for the other. For example,

Visual view



Textual view

```
     _ _ _ _ _ _
    _ _ _ _ _ _ _
   _ _ O O O _ _ _
  _ _ _ _ O _ _ _ _
 _ _ _ X X X _ _ _ _
_ _ _ _ X _ X _ _ _ _
 _ _ _ O X O _ _ _ _
  _ _ O _ _ _ _ _ _
   _ _ _ _ _ _ _ _
    _ _ _ _ _ _ _
     _ _ _ _ _ _
```

3.3 Controlling the game

You do not need to implement any sort of controller in this assignment, though you are encouraged to think about how to do so — we will build a controller in future assignments.

4 Players – human or machine?

In a future assignment, we might ask you to implement a simple AI player for this game. This will enable several possible scenarios, including solitaire play (person vs computer) and fully-automated play (which may be useful for testing). You do not have to implement a computer player right now, but you should design your model so that different player implementations could exist for human or machines to play your game. You should attempt to design a player interface that allows this to happen.

In a design document (in a separate text file), explain how you envision instantiating players and your model, so that you could play a few moves of the game. This might be one part of your README (see below), but probably should be a separate file.

We will give additional guidance in future assignments over how we suggest you implement players; the goal in this assignment is for you to think through the design possibilities and explain what you think might be an approrpiate design.

5 What to do

1. Design a model to represent the game board. This may consist of one or more interfaces, abstract classes, concrete classes, enums, etc. Consider carefully what operations it should support, what invariants it assumes, etc. Your model implementation should be general enough for multiple board sizes.

2. **You are required to identify and document at least one class invariant for the implementation of your main model class, and ensure that your implementation enforces it.**

3. Document your model well. Be sure to document clearly what each type and method does, what purpose it serves and why it belongs in the model.

4. Write a README file (see below).

5. Implement your model. If some method of your model cannot be implemented because it requires details we have not yet provided, you may leave the method body blank for now — but leave a comment inside the method body explaining why it’s empty and what details you’re waiting for. If your implementation makes additional assumptions about invariants (beyond those asserted by the interface), document them.

6. Test your model thoroughly. You are encouraged to follow the three-class split from previous assignments: an Examples class to give readers a quick understanding of your model, a ModelInterface-testing class that lives *outside* your model package, to ensure you’re testing the publicly visible signatures of your model, and an Implementation-testing class that lives *inside* your model pacakge, that can test package-visible functionality that isn’t part of your model interface.

7. Implement the textual rendering of your model described above, so we can visualize your data. Leave enough comments in your code that TAs know how to use your code to produce a visualization of a model. Test your textual rendering.

8. Design a player interface, such that human or AI players could interact with the model, and explain your design. You do not have to implement this interface for this assignment. You don’t even necessarily need to define this interface as a Java `interface`, but merely as a clearly-written English description in a textfile.

We will not be autograding this assignment, but you should emulate the textual view output above as precisely as possible, as we will likely be looking at it on future assignments.

6 How to write a good `README` file

A good README file needs to quickly explain to the reader what the codebase’s overall purpose is, what its design is, and where to find relevant functionality within the codebase. (Just because your code organization is obvious to *you* does not mean it’s obvious to a newcomer to your code!)

A README file does not have to be *long* in order to be *effective*, but it does need to be better than merely formulaic.

README files need to be in a predictable place in your codebase, or else readers won’t easily be able to find them. Typically, place them in the topmost directory of your project, or in a toplevel `docs/` directory.

Consider the following outline as a good starting point for your READMEs, and elaborate from here. This is not a verbatim requirement, but rather a launch-point for you to explain your code.

- **Overview:** What problem is this codebase trying to solve? What high-level assumptions are made in the codebase, either about the background knowledge needed, about what forms of extensibility are envisioned or are out of scope, or about prerequisites for using this code?
- **Quick start:** give a short snippet of code (rather like an Examplar example) showing how a user might get started using this codebase.
- **Key components:** Explain the highest-level components in your system, and what they do. It is trite and useless to merely say “The model represents the data in my system. The view represents the rendering of my system. ...” This is a waste of your time and the reader’s time. Describe which components “drive” the control-flow of your system, and which ones “are driven”.
  Key subcomponents: Within each component, give an overview of the main nouns in your system, and why they exist and what they are used for.
- **Source organization:** Either explain for each component where to find it in your codebase, or explain for each directory in your codebase what components it provides. Either way, supply the reader with a “map” to your codebase, so they can navigate around.

Notice that almost everything in the README file corresponds strongly with the purpose statements of *classes and interfaces* in your code, but probably does not get into the detailed purpose statements of *methods* in your code, invariants in your data definitions, etc. Those implementation details belong in Javadoc on the relevant code files. You *may* choose to mention some key methods as entry points into your code (perhaps in the Quick start section’s examples), but you should not overburden the README with such lower-level detail.

7 What to submit

Submit any files created in this assignment, along with your design document. Make certain you include your README file, which should give the graders an overview of what the purposes are for every class, interface, etc. that you include in your model, so that they can quickly get a high-level overview of your code. It does *not* replace the need for proper Javadoc!

8 Grading standards

For this assignment, you will be graded on

- the design of your model interface(s), in terms of clarity, flexibility, and how plausibly it will support needed functionality;
- the appropriateness of your representation choices for the data, and the adequacy of any documented class invariants (please comment on why you chose the representation you did in the code);
- the forward thinking in your design, in terms of its flexibility, use of abstraction, etc.
- the correctness and style of your implementation, and
- the comprehensiveness and correctness of your test coverage.

9 Submission

Please submit your homework to https://handins.ccs.neu.edu/ by the above deadline. Then be sure to complete your self evaluation by its due date.

1 Adjacency means two cells share an *edge*; it is not enough to touch merely at a corner.