Lab 8: Design

Creating any application can be broken up into three phases: designing, developing, and debugging. We've gone over aspects of all three in our previous labs, but now it's time to delve deeper into the wonderful world of design.

A good design is crucial in creating a program that not only solves your problem, but also does it in an extensible way. As your programs get larger and larger, design becomes even more important in keeping your code readable and understandable. We've provided you with a lot of information on how to design programs over the course of the semester, so let's review: A poor design could make a structurally strong implementation nearly impossible to accomplish.

**Goal:** Review techniques for designing programs by designing the game Connect Four.

**Introduction to Design**

The very first step of program design is gathering requirements. Think of this step as creating a checklist for your program. During this stage you want to answer the following questions: What is the program required to do? What is the overall goal of the program? How does it work? What should it do in various situations? How does the user interact with it? In the real world, companies often do industry research and usability testing to understand what their users expect from a program and how they expect to interact with its features. For CS15, you can usually just play with demos and read the assignment handouts to determine the requirements.

Most of the time you can break down the specifications into 4 different categories:

1. Must implement for project to function (e.g. line clearing, rotation)
2. Important features (e.g. quit button)
3. Easy to implement extra credit (e.g. lines cleared label)
4. Crazy extra credit (e.g. AI, multiplayer)

Sometimes, higher priority features cannot be added if you don't plan ahead. More complicated extra credit also requires good design. This way, if you have time to implement it, you won't have to redesign and rewrite your entire program.

Prioritizing requirements is especially helpful in keeping you on track when developing larger programs; you can regularly check your list and measure what you can accomplish given your time constraints and plan accordingly. Make sure that your top priority requirements are functional and thoroughly tested before moving on to extra credit. If adding extra credit breaks
your program ten minutes before the deadline, you should easily be able to remove that code and hand in a working project.

Connect Four Requirements

From Wikipedia:
Connect Four is a two-player game in which the players first choose a color and then take turns dropping their colored discs from the top into a seven-column, six-row vertically-suspended grid. The pieces fall straight down, occupying the next available space within the column. The object of the game is to connect four of one's own discs next to each other vertically, horizontally, or diagonally before one's opponent can do so.

Think About Your Design

Think about how to model a game of Connect Four.

The game should consist of:
- A pane that contains the board.
- Board consists of 42 clickable squares (6 rows, 7 columns).
- The initial state of the game consists of completely empty squares, which will be populated by circles on mouse click.
- When a user clicks a square, the correct piece is "dropped" down the column of the clicked square until it reaches a filled square.
  - If the column is completely full, the game should indicate the move is invalid.
- In our game, only two human players will play against each other. You'll need a pane with text to show whose turn it is.

Hint: You can design your board in a way that checking the border will be the same as checking for an occupied square on the board. This will be useful for Tetris as well.

You must also consider:
- At any point during the game a player should be able to start a new game.
- After a move is made, the game must check for a win. A player wins if he/she has four discs in a row vertically, horizontally, or diagonally.

Don't get too detailed in your design just yet! This step is designed to have you think about the game overall and to introduce you to the design issues you will face in this lab. We will take you step by step through the process of fleshing out your design in the next checkpoints.
Designing thoroughly makes coding much easier. (But you won't be coding here!) This particular assignment specification is detailed. However, even in the most detailed assignment specifications there are still going to be things that are left ambiguous. For future projects, be sure to clarify any questions with a TA and remember to document your design choices in your header comments to make your design easier to understand.

**Designing Connect Four - UML Diagrams**

**Classes**

A crucial aspect of design in an object-oriented language such as Java is deciding what classes you will need to write. A good technique is to go through the program requirements, circle the nouns, and then decide which of those nouns will have to be implemented as classes.

---

**Identifying Classes**

In a text editor, list all the classes you think you will need for Connect 4. Give a one-line summary of the classes.

Consider the following:

- What will be your top-level object for graphical components? For logical components?
- How will you keep the clickable squares organized?

  **Hint:** Think about the 2D Checkerboard from the arrays lecture and lab

- Who will keep track of whose turn it is?
- How should we model the game pieces/discs?

---

**Checkpoint 1:** Have a TA check your answers before moving on. Ask them any design questions you may have!

---

**Class Relationships**

Two ways to design a program are top-down and bottom-up. In a top-down design, we start with the class containing the mainline (in CS15 this is always the App class) and work our way down
the smallest classes. In a bottom-up design, we start with the simplest, most encapsulated objects and work our way up. Both approaches have their uses, but for this particular problem you will be designing from bottom up.

For this lab, we are going to be providing you with questions to guide your thinking. It is extremely important for you to do this when you are designing your programs. Answering these questions will lead you to a working design.

- What kind of data structure will you use to store all your squares?
- Which class should logically contain these data structures?
- Which class should handle the movement of the pieces?
- How will you model which player's turn it is?
- What instance variables will your clickable squares need to have? Which of these will need accessors and mutators?

**UMLet for Dayz**

- Create a *containment diagram* that illustrates the relationship between the classes you outlined above.
- Create an *inheritance diagram* that illustrates any inheritance relationships.
- Don't model the classes that are mostly/solely GUI components (We'll deal with that next). That means don't model Handlers, etc.

Keep your UMLet open, as you will be adding to this diagram in the next steps!

Now that you have mostly designed the building blocks, let's move on to the GUI. Again, we're going to ask you some questions to help you come up with a design.

- You have already figured out what logically contains the squares and pieces. Which class should graphically contain them? If these are two different classes then you will need a reference arrow to indicate this.
- What kinds of layout(s) should you use in order to organize all your graphical components?
- What class(es) will you use to display whose turn it is?
- What class(es) will be involved in starting a new game (i.e. when you click a reset button)?
Add the GUI!

- Add the GUI components to the containment diagram you wrote above.
- This diagram should have all the classes you will use, including layout managers and event handlers.
- Don't worry about adding all of the reference arrows in this step, we will get to that next. However, you must have all of the necessary containment arrows.

Again, keep UMLet open - in the next step you will be putting the finishing touches on your containment diagram.

The Specifics

Now that you know where all of your classes are going to be instantiated, let's cement each of their roles in the program by figuring out what important methods and instance variables each of these classes should have.

Remember: Be appropriately lazy! We love the delegation pattern. The class that initiates the action doesn't have to be the class that ultimately deals with it.

Here are the actions you are going to need to address:
- Starting a new game
- Displaying to the user whose turn it is
- Making a move and checking a valid move
- Checking for the end of the game

To add the important methods and instance variables in a class, we will make sections in our UMLet containment diagram in the bottom of the class rectangles.

To add a dividing line for methods or instance variables to a class or interface, create a new line and add -- in the Properties section and then go to the next line:

```
<table>
<thead>
<tr>
<th>ExampleClass</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object_boringInstanceVariable</td>
</tr>
<tr>
<td>Point_someLocationSomewhere</td>
</tr>
<tr>
<td>void doNothingMethod()</td>
</tr>
<tr>
<td>boolean isAwesome()</td>
</tr>
</tbody>
</table>
```

When you do this, you will likely want to resize your class or interface so that the text you added is visible.
Add the References

Finish up your containment diagram.

- Make sure every class has a reference to all the classes it needs.

- Include the major methods and instance variables each class would need to have to complete the basic functionality of Connect Four:
  - Starting a new game
  - Displaying to the user whose turn it is
  - Making a move and checking a valid move
  - Checking for the end of the game

Checkpoint 2: Call a TA over to check your containment diagram.

Designing Connect Four: Pseudocode

Pseudocode is an equally important part of program design. Before coding, you should have a clear idea of what complex methods your program needs and how these methods should work.

Pseudocode should be familiar to you from Lab 7. Consider a fictitious algorithm helping students in a queue outside of the Fishbowl. The pseudocode would look as follows:

```
// inputs: signmeup - a Queue which holds the students in line
// returns: nothing
method helpStudents(signmeup)

For each student in the queue
  Call in the student
  Wait for student to sit down
  Listen to student's question
  While question is unresolved
    Use Socratic method to reach solution
  Send away a happy student
```

You can easily translate this high level language into actual code. As methods become more and more complicated, pseudocoding becomes increasingly helpful.

For Tetris, you'll be required to write pseudocode for the line clearing algorithm. We will not be teaching formal pseudocoding in CS15; for now, we expect the same level of description in your
pseudocode as in the example above. We do ask that you follow the same indentation rules as if the pseudocode were an actual method, just like in the example above.

For more examples of pseudocode, please refer to the lectures or the Pseudocode lab.

**Make Your Move**

Write pseudocode for the `makeMove()` method in Connect Four.

**Note:** This method should also handle checking if the move is valid!

---

**Checkpoint 3:** Have a TA look at your pseudocode, and get checked off for the lab!

---

**Conclusion**

Design is one of the most important parts of programming. Without a solid understanding of what you hope to accomplish with your program, and a thorough plan for implementing it, you will run into many otherwise-avoidable pitfalls while coding.

But at the same time, programming is not a linear task. It does not always happen that you design, then develop, then debug. Often these tasks are interwoven. If you find yourself in the development stage writing ugly code that is hard to follow, it’s probably a good indication that you should rethink part of your design. And if you finish coding everything before you begin to debug, you are going to have a terribly difficult time trying to find all the errors.

Your design is the most important part of any programming assignment and is the key to coding efficiently.