Design

Note: This lab may take more than the allotted 1.5 hours. However, learning how to properly design your programs will save you time in the near future (Hint: Tetris this week and Final Projects are looming).

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 put on your thinking cap and 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 going through a thorough design for a game of Connect Four.

General 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.

Let's go back to LiteBrite. The requirements for LiteBrite were pretty simple:

- There should be a grid and palette in the main application window.
- There should be at least 3 color buttons in the palette.
- When a user clicks on the grid, a light shows up in that spot with the currently selected color.
- A user can select another color by clicking on one of the other color buttons in the palette.
- If the user clicks on the currently selected color button, nothing changes.
- If the user clicks on the "Exit" button, the program closes.

- If the user clicks on an existing light, the light will be replaced with a new light colored with the currently selected color.
- Sometimes you will also run into situations where you need to prioritize your requirements.

Most of the time you can think of 3 different rankings:

1. Must implement, will fail project/won't get full credit if I don't.
2. Desirable and somewhat easy to implement, extra credit.
3. Crazy Extra Credit! Bells and Whistles!

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. For example, you should make sure your priority 1 requirements are working before you go on to priority 2, so if implementing color buttons breaks your Cartoon two hours before the deadline, you should be able to easily take that code out and hand in the working version with just the priority 1 requirements implemented.

NOTE: Sometimes higher priority features cannot be added if you don't plan ahead. So if you want to add something really cool, like a timed animation, make sure to design for it before starting to code! This way, if you have time to implement it, you won't have to redesign and rewrite your entire program. This means spending more time in the designing phase, but it will save you a lot of time later on!

Connect Four Requirements

Overview:

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.

Connect Four

Check Point 1: Think About Your Design

Think about how to model a game of Connect Four.

The game should consist of:

- A frame 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 move is invalid, and the game should indicate this.
- In our game, only two human players will play against each other. You'll need a panel with a label on the right to somehow show whose turn it is to move.

Hint: you should be able to design your board in a way so 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. If you think the ambiguous parts are critical to the program, be sure to clarify with a TA. Remember to document your design choices in your header comments and everyone will understand your design.

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.

Inheritance

The next step in designing the program should be to determine which classes we need to make from scratch, which ones we need to extend from existing classes, and which classes can simply be existing classes with no modification. The best way to illustrate the relationships is to use an inheritance diagram. It should be fairly obvious that this program needs to have GUI. Therefore, we'll be using Shape classes heavily.

Check Point 2: UMLet for Dayz

You will now be creating a UML Inheritance diagram. Think about all of the classes and interfaces that you will need to have in this program. Consider if they should extend from other classes and/or implement interfaces.

Using UMLet, create an *inheritance diagram* based on this.
Consider the following:

- What should you be extending as your top level object?
- How will you keep the clickable squares organized?
  Hint: Think about the 2D Checkerboard from the Arrays lecture/lab
- Who will keep track of whose turn it is?
- Should the game pieces/discs extend from or contain a subclass of Shape? Which makes more sense and why?

Have a TA check your inheritance diagram before moving on.

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.

The next step to tackling this design problem is to identify the parts of the program that are going to be the trickiest. 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? How about your pieces?
- 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 squares need to have? Which of these will need accessors and mutators?

Check Point 3: More UMLet, More Fun

- Create a containment diagram that illustrates the relationship between the classes you modeled in the inheritance diagram as well as any additional classes you may need.
- Don't model the classes that are mostly/solely GUI components. That means, **do** model your squares and pieces, **don't** model your frame, top level panel, mouse listener, border layout(s), etc.
    - We'll deal with GUI at the next check point.

Now that you have mostly designed the building blocks, let's go to the next level up. In this program, the next level up is the GUI. Whenever you hear the word GUI in Java, think Shape! 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?

Check Point 4: Add Shapes!

- Add to the containment diagram you wrote above, add the GUI components.
- This diagram should have all the classes you will use, including layout managers, and listeners.
- 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.*

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.

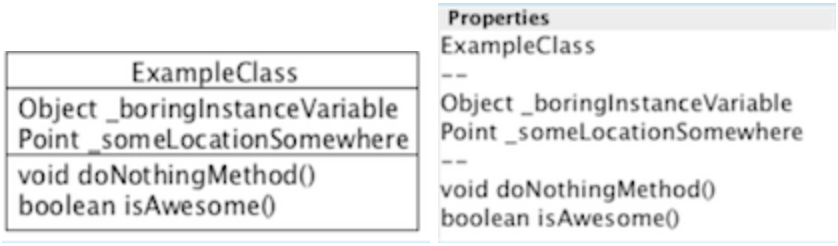
Remember: Be appropriately lazy! We love the delegation pattern. The class that initiates the action doesn't have to be the class that ultimately deals with it.

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 off 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:



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

Check Point 5: 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.

Designing Connect Four: Pseudocode

Pseudocode is an equally important part of program design. Before coding anything, you should have a clear idea of what major/complex methods your program needs (for example, adding a Lite to the Grid in LiteBrite) and how these methods should work.

Pseudocode should not read exactly as it would be programmed! This should be familiar to you from the Pseudocode lab. 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.

Check Point 6

Write pseudocode for the makeMove() method in Connect Four.
Note: This method should also handle checking if the move is valid!

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 avoidable pitfalls while coding.

But at the same time programming is not a linear task. It does not always work 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 (this is especially applicable in Tetris and final projects). 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 almost always the difference between finishing early and barely finishing before the deadline.

Make sure to have a TA check off your lab before you leave!