hw01 – Mastermind

Points: 50
Team-based? Two-member team programming — Required
Due: Monday, 2023 Oct 16

Objectives

- Research and self-learn a new API – `java.net` for network communication
- Team work with Git
- Refactoring!

Prepping for your Assignment

You will be working in a team of two members. Select your team, and then **follow the procedures in the document Teamwork with Gitlab and IntelliJ**, located on Moodle. Use the project name **`csci205_hw`** as you follow those instructions.

Mastermind

This is largely taken from https://en.wikipedia.org/wiki/Mastermind_(board_game) .

Mastermind is a code-breaking game for two players. One player serves as the **codemaker**, and the other is the **codebreaker**. The game begins with the codemaker secretly choosing a code out of six colored **code pegs**. Repetition is allowed. This code is hidden from the codebreaker. Then, the game proceeds with the codebreaker guessing what the code might be. After each guess, the codemaker scores the guess, using red and white **scoring pegs**. A red scoring peg is shown for each code peg that is in the correct position (correct color in the correct position). In contrast, a white scoring peg is placed for a code peg that is in the code but in the wrong position (correct color but in the wrong position, and exclusive of the already-scored-red pegs). The scoring pegs give no indication as to which code peg is right or wrong. That is up to the codebreaker to figure out. The object of the game is for the codebreaker to guess the code in 12 turns or less.

NOTE: There are many variations on this, so generalize your solution. The pegs could be numbers, letters, etc. Some variations even blank positions are allowed. The number of turns can be increased or decreased depending on the game setup. Regardless of the variation, the gameplay is the same. There are n positions, each with k possible objects, with repetition permitted. Therefore, there are k^n possible patterns, and the codebreaker has to guess in p turns, or they lose. The classic Mastermind game has four positions, with six different color pegs, for 6^4 = 1296 possible combinations to choose from, with 12 turns allowed.

We strongly encourage you to try the game to start wrapping your mind around how it works. It will help you think about the classes you need to represent actual objects in the real world, their interactions with each other, and the most important part of any design, the data abstractions you will use to represent all of the important parts of your game. There are many online versions available. Here is one to get you started: http://www.archimedes-lab.org/mastermind.html

Part 1 - Problem Statement

*As you are completing the functionality, **we will expect you to use good OOD principles**. Do your best to create solid, clean, easy-to-understand code! As you are thinking about the game, start writing down the different objects that are all at play in a game. These might be good candidate classes!*

Implement a console (text) based game of Mastermind. Use the classic rules for your first iteration:

- 4 code pegs, each in one of 6 possible "colors"
- 12 guesses allowed.
- For code pegs, use the numbers, `1, 2, 3, 4, 5,` and `6` to represent the six colors.
- For the scoring pegs, use two characters, * and +, to represent the red and white scoring pegs, respectively. Use the – to indicate an empty scoring peg slot. Always show the scoring pegs in order of *, then +, then -.

The codebreaker should be able to enter guesses as 4-digit numbers, and the codemaker should respond with a string representing the scoring pegs.

For this first step, the computer will represent the codemaker, and the user will play the role of codebreaker. Thus, technically, you will implement a single-user variant of Mastermind.

Here is a capture of one way you might play the game:

```
Guess my code, using numbers between 1 and 6. You have 12 guesses.
Guess  1: 1122
1122 -->  ----    Try again. 11 guesses left.
Guess  2: 3344
3344 -->  *---    Try again. 10 guesses left.
Guess  3: 3455
3455 -->  *++-    Try again. 9 guesses left.
Guess  4: 3456
3456 -->  +++-    Try again. 8 guesses left.
Guess  5: 6455
6455 -->  *+++    Try again. 7 guesses left.
Guess  6: 5465
5465 -->  **++    Try again. 6 guesses left.
Guess. 7: 5645
5645 -->. ****    YOU WON! You guessed the code in 7 moves!

Would you like to play again? [Y/N]: N
Goodbye!
```

Designing a solution

An important part of being an engineer of any type is coming up with a design or a model for your solution before you start your implementation. **You MUST submit a UML class diagram, a UML state diagram, and a UML sequence diagram as part of your solution**.

How should you proceed? Start by playing the game. Understand every object and entity that is used to manage the game. And, think about the flow of the game. And… think about future improvements you will need to make. The first part is only the beginning.

The UML Sequence Diagram

You've learned about the UML Class and State diagrams.

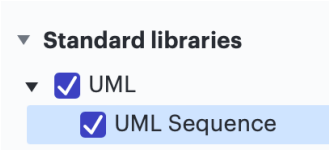
- A **UML Class Diagram** tells us about how classes in your solution will be designed (attributes and methods with their visibility) and tell us how these classes are related to each other.

- A **UML State Diagram** tells us how the state of an object changes from instantiation to its end of life.

In contrast, a **UML Sequence Diagram** shows how different objects work together to achieve different outcomes. Mastermind is a perfect opportunity to learn about these valuable diagrams.

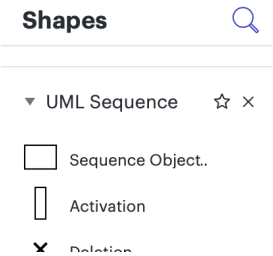
Lucidchart has a collection of very simple tutorials to help you get started with their diagrams. **Read https://www.lucidchart.com/pages/uml-sequence-diagram to learn about sequence diagrams.** Pay close attention to their examples. Then come back here and we'll help you get started.

1. Go to Lucidchart at https://lucid.app
2. Use the directions from an earlier lab to create your new diagram from a UML class template. Keep that since you'll need it for your class diagram.
3. You probably have one tab in this document called UML Class. Create a new tab and name it Sequence.
4. Click on the More shapes button on the bottom of the Shapes panel. Search for UML Sequence:

   and select **Use selected shapes**.

   You should now see template objects containing everything you need for Sequence diagrams:

You read the tutorial, so go ahead and create your sequence diagram! Consider the important objects. Use the principles we've been discussing as you think about important objects in your design. While you may have objects to represent a guess, guessing pegs, scoring pegs, etc, those objects are too detailed for a sequence diagram. Think at a higher level, especially when beginning your OOAD process. Let's suppose we have the codemaker and a codebreaker, and maybe we have a separate object called gameManager to handle game management things, such as generating a new secret code. We might have a sequence diagram that looks as follows:

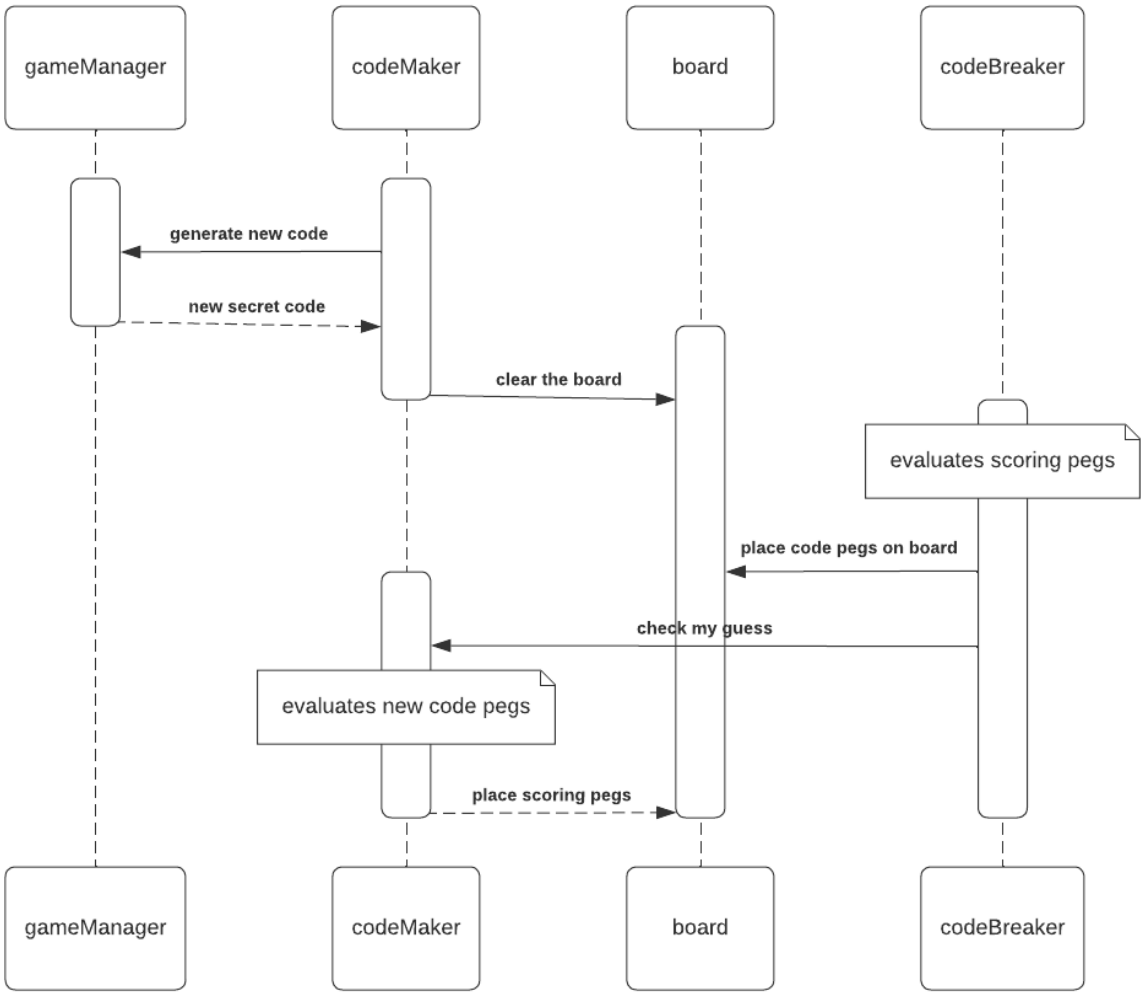


Figure 1 – Illustration showing a basic UML sequence diagram for Mastermind, with objects, lifelines, synchronous (sender must wait for response) messages, return messages, and activation boxes

Notice how this gives a nice visual representation of how you might design four classes (of which you'll likely have more) in your design of a solution. A sequence diagram also gives you a sense of the important methods you'll need to create. **A message is implemented as a method call to that object.** So, as you proceed with your implementation, you can change these messages to your actual method call. These tools help you keep your mind focused as you proceed with your implementation. (The diagram above is incomplete. For example, it's missing the loop construct.)

Like all UML diagrams, you can keep them simple when you are first brainstorming your solution and gradually refine it with more details, notes, and actual method and object names. It serves as another great tool for documenting your code, and working with a client to express your understanding of your problem.

IMPORTANT:

- Sequence diagrams usually use object names, not class names. (Remember, objects vs. classes are distinguished by the first letter, where objects are lowercase and classes are Uppercase.) However, there are times when you want to show how multiple objects of the same class interact. For example, if you

were designing a solution for an ATM, and you needed to handle the transfer of money between accounts (which are both of type Account), you could have the following in your names for the two objects:

```
savingsAccount:Account        checkingAccount:Account
```

- Do not list attributes. That's for the class diagram.
- Actor symbols are any entity that is external to the system. They may not be just humans but could be databases or other network servers, etc.
- Option loops are repetitive lines of communication. Include a note when you know how often a sequence of messages will repeat or if you know what the terminating conditions are.
- Alternative blocks are choices between different messages that may occur.
- Notice the subtle, but important differences between synchronous and asynchronous messages when reading the Lucid tutorial. Most of the time, when you send a message that requires a response, you use:

When you send a message that does not require a response before the sender can continue, you use:

All response messages will be dashed-lined arrows.

HINT: Lucidchart's **UML Sequence Markup Language** (https://help.lucid.co/hc/en-us/articles/16262874090900-Create-a-sequence-diagram-with-UML-markup-in-Lucidchart) can make it incredibly easy to create these diagrams! Look at their tutorial! For example, we created the above diagram right in LucidChart in their markup panel using the following:

```
activate gameManager
activate codeMaker
codeMaker -> gameManager: generate new code
gameManager <-- codeMaker: new secret code
deactivate gameManager
activate board
codeMaker -> board: clear the board
deactivate codeMaker

activate codeBreaker
note over codeBreaker: evaluates scoring pegs
codeBreaker -> board: place code pegs on board
activate codeMaker
codeBreaker -> codeMaker: check my guess
note over codeMaker: evaluates new code pegs
codeMaker --> board: place scoring pegs
deactivate codeMaker
```

Minimal requirements:

Your code must be cleanly formatted, clearly documented with javadoc, with proper variable names, method names, and classes that are self-descriptive.

You must have Javadoc for every class, data member of each class, and method for each class.

If you use any code that came from anywhere but your own brain, you must provide clear references to the web pages that you retrieved code from.

Each method should clearly indicate who completed the work with an `@author` tag. Failure to complete these requirements will result in lost credit.

You must develop JUnitit tests for the most important classes in your design.

- Your assignment must be contained in a package called `hw01`.
- You must write and commit a **`README.MD`** file that contains critical information, including:
  - Names on the project
  - Your primary resources used to complete the project, including URLs to all pages, github repos, etc. that you used to get help and guidance.
  - This should be readable right from your gitlab repo.
- For this assignment, **all user I/O will be console driven**. No `JOptionPane` or any GUI of any kind.
- Repeating, **your javadoc must absolutely contain references and hyperlinks to every source of information you retrieved that is NOT YOUR OWN!** There are NO EXCEPTIONS to this, and you will be reported for any violation that we find that includes work that is not your own implementation.
  - There are literally hundreds of Java Mastermind implementations available all over the Internet. The game portion of this is simple, and you should NOT need to scour the web for solutions to this.
  - All work should be the work of you and your partner! If you obtain ideas from anywhere that is not your own head, you must cite it in your Javadoc (did we say this already? Yes, yes we did. We're serious.)
  - Any code that has no references included in the javadoc will be assumed to be your own code, and if we find any indication whatsoever that it closely matches online code somewhere, you will be reported to the Board of Review.
  - You must clearly indicate in your Javadoc who worked on each method. Hyperlinks to source pages must each be prefaced with an `@see` javadoc tag, followed by embedded HTML that includes a hyperlink to the source page. An example javadoc is included below.
- You must do your very best to **follow an object-oriented approach**. MULTIPLE CLASSES MUST BE DESIGNED! ANY SOLUTION DELIVERED IN ONE CLASS WILL BE DEEMED UNACCEPTABLE!
- Create a very top-level UML diagram for your work. It must minimally focus on the relationships between classes. Not every data member or method of every class needs to be shown, but relationships must be.
- All UML diagrams must be delivered as PDF, placed in your package folder for your project.

Some helpful pointers

- Work on the UML diagrams first. This will get you thinking through a good design.
- Your methods should be short, concise, and well-defined, with good documentation. Any code we receive with enormously long methods will have major points deducted. An entire program that is received in one static main method will have *substantial* points lost!
- You should have JUnit tests for the most critical methods in your program for the game.
- Use the IDE refactoring capabilities to your advantage! There are *many* techniques built into IntelliJ that will make your life easier.

Example Javadoc with a reference to a web page…

```
/**
 * Given a path to a WAV file, play the file and return when the file has
 * completed playing
 *
 * This code was based on information found at www.codejava.net.
 *
 * @see
 * <a href=" http://www.codejava.net/coding/how-to-play-back-audio-in-java-with-examples">
 *   http://www.codejava.net/coding/how-to-play-back-audio-in-java-with-examples</a>
 *
 * @param wavFilePath is the name of the file to play as a String
 * @throws UnsupportedAudioFileException if the audio file is not supported.
 */
public static void playWAVFile(String wavFilePath) throws UnsupportedAudioFileException {
```

Saving and committing your work

Commit all of your Java files, JUnit files, and PDF class diagram(s) to your repository. Check in your work with the message "`hw01, complete`"

Push your work to the remote server

Perfection is not expected. A solid design is, with a good effort evident. Clean, structured code is mandatory. And, you must submit your UML class, state and sequence diagram. Once your work is completed, be sure to merge your final solution to the `main` branch. Push `main` to the remote. Failure to properly push your code will result in substantial points lost. We only pull the `main` branch down to grade. Get on Gitlab and make sure your code is properly pushed.