hw 01, part II – Mastermind Solver

Points: 50
Team-based? Two-member team programming — Required
Due: Friday, 2023 Oct 20

Objectives

- Refactoring
- Team work with Git
- Improving your object-oriented design
- Implementing a suite of solving algorithms

Teamwork reminder

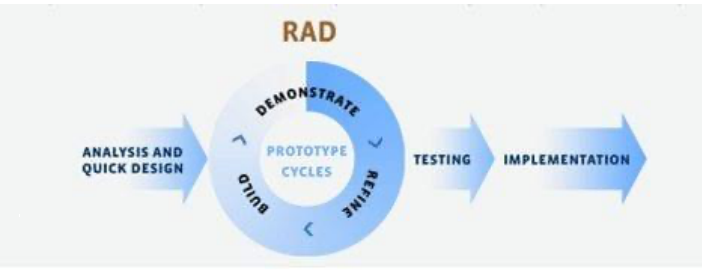
This is a strong reminder that **each team member must be an equal part of the team**. Everyone's voice is important. Every member deserves to be heard. It's common to have "tunnel vision" when you get an idea for a solution to a problem. Hearing the ideas of team members can quite often help expand and improve the central design, considering concepts that you weren't able to think of yourself. If you are part of a team and do not feel as though you are an equal part, **please confidentially contact your professor ASAP**.

Prototyping

You were thrown into a new phase of the course, where you are being given complex problems to work on. You will learn (if you haven't already) that a good design can work *wonders* for making your software development life easier. Reflect a moment on the first assignment. How did it feel? Are you proud of your code? Do you feel like you conquered a good OOAD? Did you use any of the techniques taught to identify good classes and design them well? If not, you must not neglect that information!

OK, so perhaps that first part was rough for you. Perhaps you have made it through the first phase, barely, and you realize that your codebase is a… clusterf… hmmm… let's go with…your code is a *bad* example of software that is easy to understand, maintain, improve, evolve, etc. It's more likely that you thought of dozens of improvements you could have made with your design along the way, if only you were given the time! Well, be happy! **Part of this assignment is to give you a chance to refactor and redesign your code!**

Sometimes in software development, we develop **prototypes** – rough code designed to help a team work out solutions to some of the challenging parts of a problem. Maybe it was never really meant to be a final piece of code delivered to the client. Prototypes are quite common in practice. Prototypes often yield an added benefit of helping the team better understand a good OOAD to follow. In fact, there are some earlier software engineering methodologies that rose in popularity before Agile became popular called **RAD – Rapid Application Development** (see figure). RAD embraces an iterative cycle of creating prototypes that demonstrate the feasibility of your solutions to more challenging parts of a project. They help the team gather solutions as they work toward a solid design and implementation.



Problem Statement, Part 2

Among the first decisions that you and your partner need to make is to determine what code to keep, and what code to rewrite with a better, cleaner design. That is entirely up to you, but if you decide to make major changes, then **use Git to your advantage**! Work in an entirely different branch so that you do NOT lose your working code. Use Git to help you switch between your pieces of code, and carefully merge to main only solid, tested, clean, documented code. **REMEMBER: WE ARE ONLY GOING TO GRADE WHATEVER YOU COMMIT TO YOUR MAIN BRANCH! WE DO NOT PULL ANY OTHER BRANCHES DOWN!**

There are two primary objectives with this portion of the assignment, and you must satisfy both.

Objective 1: Mastermind Solver

You must add the functionality to implement various algorithms to solve Mastermind. There are numerous algorithms online. You need to implement three solvers.

1) `RandomSolver`- A `RandomSolver` will randomly guess codes until it happens to create a match. Sometimes, that could happen right away. And other times, it could take $6^4 = 1296$ moves until it guesses the correct one, and sometimes it could take a lot more!
2) `MiniMaxSolver` - The **minimax algorithm** is a well-known AI algorithm for decision making. It works great for zero-sum games. In 1977, Donald Knuth, a world-renowned computer scientist, demonstrated how the minimax algorithm could be used to find a solution to any Mastermind game in an average of less than 5 moves (assuming the standard game of 4 pegs with 6 colors per peg).
3) Finally, you can choose a third solver algorithm of your own. You do not need to exceed `MiniMaxSolver`, but your third algorithm must do far better than random guessing.

The Wikipedia page for Mastermind documents several solver algorithms:
https://en.wikipedia.org/wiki/Mastermind_(board_game)

Github has an enormous selection of repositories that implement Mastermind and many different solutions in many different languages. One we found, written in C++, has a very nice description of Knuth's implementation on their readme page:
https://github.com/nattydredd/Mastermind-Five-Guess-Algorithm

You will need to modify your program so that it asks the user whether to run in **Solver** mode, or **Game** mode. If Game mode, then you are running the same game you had in part I. If the user chooses Solver mode, then ask the user 1) which of your Solvers to run, and 2) how many games to simulate. After the test, you should output some basic stats based on the number of turns each run took (i.e. in Solver mode, you must allow your game to go beyond the basic Mastermind game limit of 12 moves). Output:

- The number of games played
- The mean number of turns this solver took to guess the code
- The least number of turns
- The most number of turns
- The total time it took to run your test on all games played

For example, your output after running with `RandomSolver` 1000 times might look like the following:

```
RESULTS:
RandomSolver - Statistics:
Number of games: 1000
Average: 943.3 turns
Shortest: 2 turns
Longest: 3421 turns
TOTAL TIME: 1.97 sec
Goodbye!
```

Your output can be formatted however you choose. However, you must output those measurements.

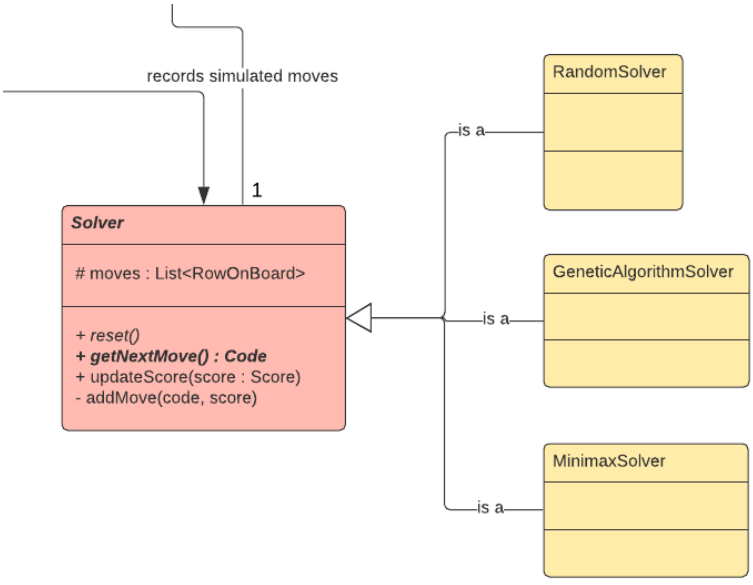
Objective 2 – Improving your design while incorporating a Solver

When you have *behavior* that multiple disparate classes are supposed to implement in a uniform way, this is a good time to use **interfaces**. However, when the common behavior is tied to a particular entity type in your system, and that entity also keeps track of data to carry out those behaviors in a uniform way, then you can either consider interfaces with default methods, or sometimes **abstract** classes are a better idea.

For our situation, we have different types of solvers that must play Mastermind. They each have a unique behavior that govern their choice on what their next move is. However, the game itself should be agnostic to whatever solving algorithm is being played. Think about it – when it comes to a game of Mastermind, from the game's perspective, the play proceeds the same whether it is from a user entering a code to guess or a solver entering a code to guess. So, a solver only really needs to provide a way to provide a code to guess, play it, and needs to receive the score feedback. That's it! The solver then has all the information it needs to figure out what code to guess next (well, unless you are a `RandomSolver`, then frankly, you really don't care what you've played before.)

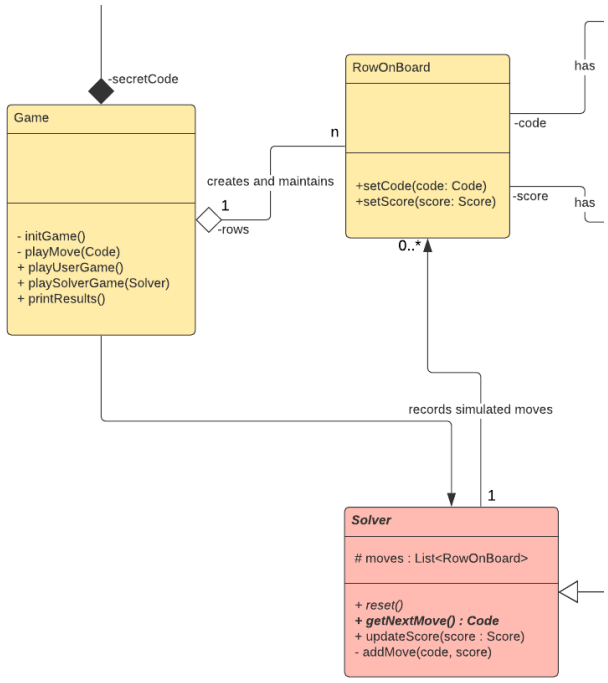
NOTE: YOU CAN IMPLEMENT YOUR OWN DESIGN! These are ideas you can consider, or not.

Here is one idea for a design that follows that line of thought. This is a design with abstract classes that you could consider. Notice the abstract class, called `Solver`. (Abstract classes have ***italicized*** class names, often in bold as well. The abstract methods in the class are also ***italicized***.) This approach has one abstract class, `Solver`, and three different concrete classes that are extended from `Solver`:



A solver simulates the behavior of a human, so ultimately, you need only to consider an interface for the abstract class that would respond with the next code to play, based on the previous moves it played and the scores on

those moves. In other words, this is the same thing the user does when playing! The purpose of `getNextMove` is the way the game can retrieve the next code to be played. Then `updateScore` is called by the game to send the solver the score obtained from the current code played by the solver. A `reset` method is a good idea too. After all, you need to keep repeating game simulations, potentially thousands of times! Remember – a solver can not see the secret code! It can only receive the score after each code is played, just like the code breaker!

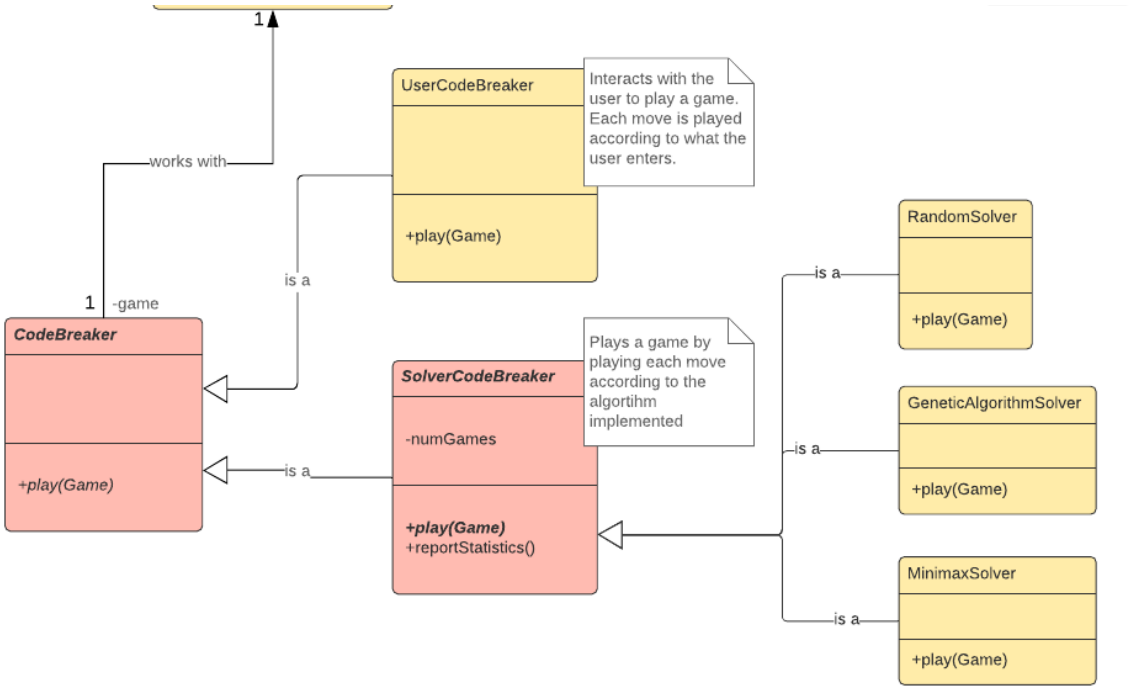


Consider the possible `Game` class on the left. With this design, the `Game` class could have two methods to play a game – one that governs the play from a human, `playUserGame`, and one `playSolverGame`.

Again, this is just one of many possible designs you could choose. Is it good? This solution puts a lot of responsibility on the `Game` class, making it a bit unwieldy. The `Game` is keeping track of the board, the secret code, and managing the play of the game between a user, playing the game with a Solver, and more! This is a low cohesion design! It works, but the `Game` class is going to be a class with bad smells and very difficult to maintain!

Here's another idea. We need to extract functionality out of the `Game` class to clean it up. What if we take the burden of playing the game and put it on a separate class representing the role of the codebreaker. We'll call it `CodeBreaker`. After all, in a real game, it is the codebreaker who is taking the burden of breaking the code and doing most of the work in the game.

Consider the following snippet of this possible design. It introduces the `CodeBreaker` abstract class. It has one important abstract method, `play(Game)`, which handles playing through an initialized `Game` instance provided to it. Only concrete classes will implement the `play(Game)` method.



Think about how much this design cleans up the `Game` class. What would a `Game` class look like now? It would only be keeping track of the board, the secret code, and evaluating every move and returning scores. Hmmm… perhaps we could simplify the `Game` class even further if we created a `CodeMaker` class, and have that be responsible for generating the secret code, and verifying an arbitrary guess to see if it matches the secret code, and scoring moves? Consider how much that improved design is easier to understand, and even easier to maintain? Then, the Game class really becomes only a Board class! Now, we are approaching a good design. Many classes? Yes. But, it'll be far easier to understand and maintain, as opposed to having fewer classes with multiple behaviors and data they need to maintain.

This is an important part of your objective for this hw01-part2. Refactor your design. Finally, be sure to organize your design as a series of packages. For example, maybe you'll consider the following?

- `hw01.game` – all of the standard game classes
- `hw01.solver` – all classes that pertain to algorithmic solutions to Mastermind

Final words…

The better job you do on creating a very good, clean design now, the easier job you will have creating a GUI for your game!

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. Additionally, 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.

- Your assignment must be contained in a package called `hw01`. But, as suggested above, you should go further than just using one package to maintain everything.
- 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.
    - 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 hundreds of Java Mastermind implementations available all over. The game portion of this is simple, and you should NOT need to scour the web for solutions.
    - There are also many different Mastermind solutions for various solvers. You are welcome and even encouraged to research to find some implementations to get some ideas on how to make this work. However, the solution you implement must be your own.
    - All work should be YOURS! 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 your 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**.
- UML – You should update and clean up your UML Class and UML Sequence diagrams, and your UML State diagram if you implemented one. Your final UML diagram must reflect your solution. Make it as clean as possible (i.e. do not include every method, only the important ones that are critical to the functionality of your program.) Prioritize making good relationships and multiplicities. And, as you see in our examples above, annotate your relationships. It can make your design more readable. Feel free to use little note boxes if it explains your design a bit cleaner.
- If you end up with a complex design with many classes, then in additional to the above, create a very top-level UML diagram for your work that indicates classes as boxes only, with relationships between classes shown only.
- You must refactor and clean your code. Make it look good. Good variable names. Good Javadoc. Good comments. Many small classes with well-defined methods. This might be your example code you could use on your Github page for internship interviews.

Some helpful pointers

- Strive for classes that are highly cohesive, with minimal coupling. We cannot emphasize enough about the importance of taking time to brainstorm your design before you code.
- 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!
- Use the IDE refactoring capabilities to your advantage! There are *many* techniques built into IntelliJ that will make your life easier.

As a reminder, here is an 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 part2, complete`"

Push your work to the remote server

Once your work is completed, be sure to merge your complete solution (code, tests, and PDFs of your diagrams) to the main branch. Push the main to the remote. Failure to properly push your code will result in substantial points lost. Get on Gitlab and make sure your code is properly pushed.