ECI Miniproject: MasterMind // Nicos Vachnadze & Bruno Flörke // 22.11.15

We will use this platform to shortly outline the different classes and point out some features of our code. Our focus lay less on the actual GUI and more on how to structure a project like this in an OO manner. We put some effort into avoiding to hardcode variables, which should give this program some flex regarding modularity and scalability.

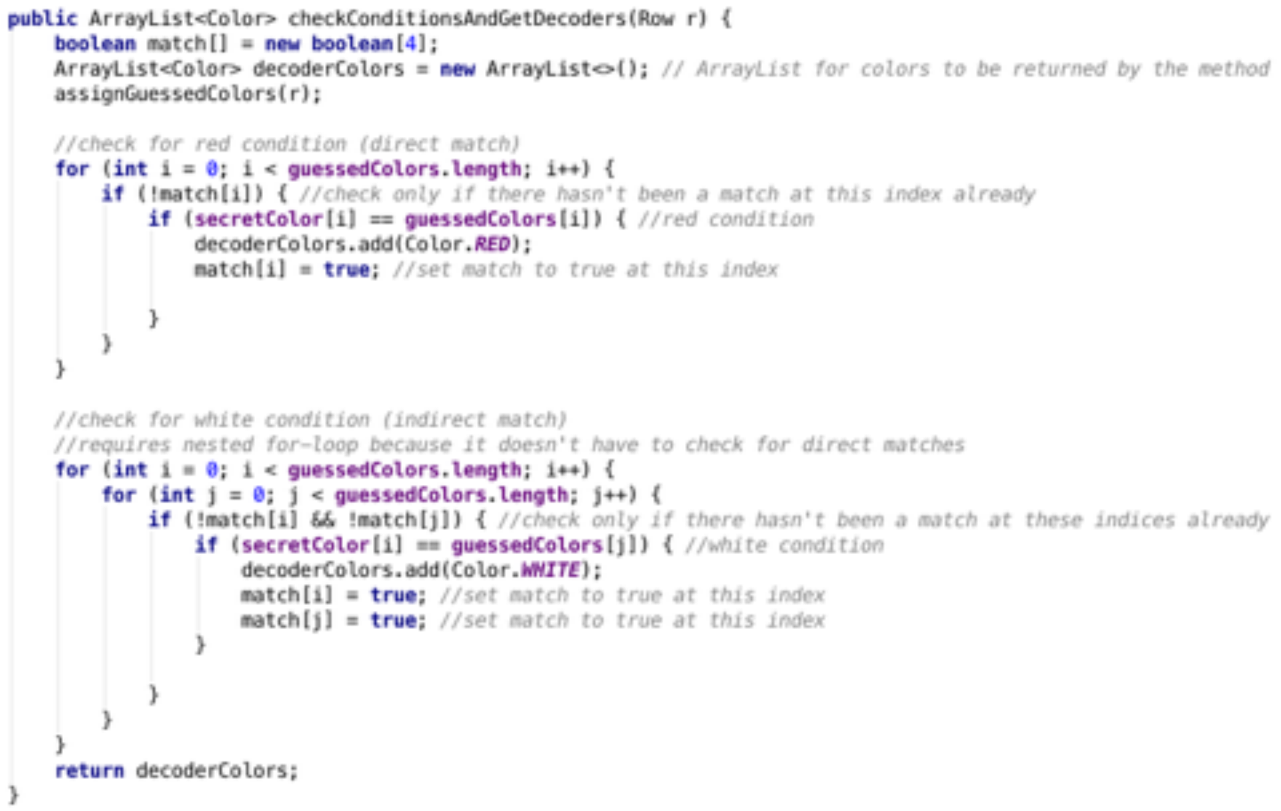
Main: is a Main. Contains the frame, instantiates Board, which acts as a launcher

Peg: holds traits of a peg, introduces some variables and extends JButton which gives us access to built-in swing-methods. toggleColor() is the only bigger method here, called when clicking a peg.

Row: is where we 'draw' the pegs and decoders, extends JPanel. They are added to respective arrays and the JPanel. It seems crucial to distinguish a Row-class, as both pegs and decoders are manipulated, well, row by row. hasBeenClicked() was a cool and esp. simple idea, and is called both 'down' in Peg and 'up' in Board.

Board: also extends JPanel. This naturally turned into the class where all paths meet, so to speak. activateNextRow() took a lot of rewrites but turned out to be quite elegant, addLockGuess() is integral to the actual game mechanics of MasterMind and, amongst other things, calls the check method from LockGuess (which we're pretty proud of :)

SecretColorList: is where the magic happens. Contains in our opinion the coolest part of the code, the algorithm in checkConditionsAndGetDecoders (which we realise now should've been called checkConditionsAndGetDecoderColors). We'll shortly outline our thought process:



```java
public ArrayList<Color> checkConditionsAndGetDecoders(Row r) {
    boolean match[] = new boolean[4];
    ArrayList<Color> decoderColors = new ArrayList<>(); // ArrayList for colors to be returned by the method
    assignGuessedColors(r);

    //check for red condition (direct match)
    for (int i = 0; i < guessedColors.length; i++) {
        if (!match[i]) { //check only if there hasn't been a match at this index already
            if (secretColor[i] == guessedColors[i]) { //red condition
                decoderColors.add(Color.RED);
                match[i] = true; //set match to true at this index

            }
        }
    }

    //check for white condition (indirect match)
    //requires nested for-loop because it doesn't have to check for direct matches
    for (int i = 0; i < guessedColors.length; i++) {
        for (int j = 0; j < guessedColors.length; j++) {
            if (!match[i] && !match[j]) { //check only if there hasn't been a match at these indices already
                if (secretColor[i] == guessedColors[j]) { //white condition
                    decoderColors.add(Color.WHITE);
                    match[i] = true; //set match to true at this index
                    match[j] = true; //set match to true at this index
                }

            }
        }
    }
    return decoderColors;
}
```

The check for the red condition (direct match) is straightforward: we iterate through the length of secretColor[] (the "key") and try to find a match in guessedColors[] for corresponding indices. If a match is found Color.RED is added to Arraylist decoderColors.

The check for the white condition (indirect match) requires a nested for-loop: we're not only checking corresponding indices but for each index in secretColor[] have to check against ALL indices in guessedColors[]. If a match is found Color.WHITE is added to Arraylist decoderColors.

Now, the problem with these two checks is that a direct match also fulfills the white condition, which results in too many added decoders. To get around this, we created a boolean array match[] of the same length as the other two arrays. By default, it contains the values {false,false,false,false,}. When we check for the red / white conditions we set the value at the corresponding index to true whenever there's a match. The important part is that the check for either condition is ONLY performed, if the value in match[] at the corresponding index is found to be false (!match[i]). since the check for red conditions is performed before the check for white conditions, this rules out the the possibility of a double positive.

LockGuess: contains some JButton-properties.

Decoder: also contains some JButton-properties.

All in all we're happy with how it turned out, it's not an especially pretty game to look at but fulfils all the bare-bones functionality one could expect from MasterMind.

About three quarters of the way through we came across the Model-View-Controller architectural pattern. Breaking down and compartmentalising the code in this way might have simplified some things and made the code easier to comprehend for others. But we had committed to our way and now it's the way it is. Next time we'll consider that methodology.

There's quite a lot of calling of methods between classes, which sometimes was hard to keep track of but for the most part we think we succeeded in commenting where necessary and naming the individual methods and variables in a self-explanatory way.

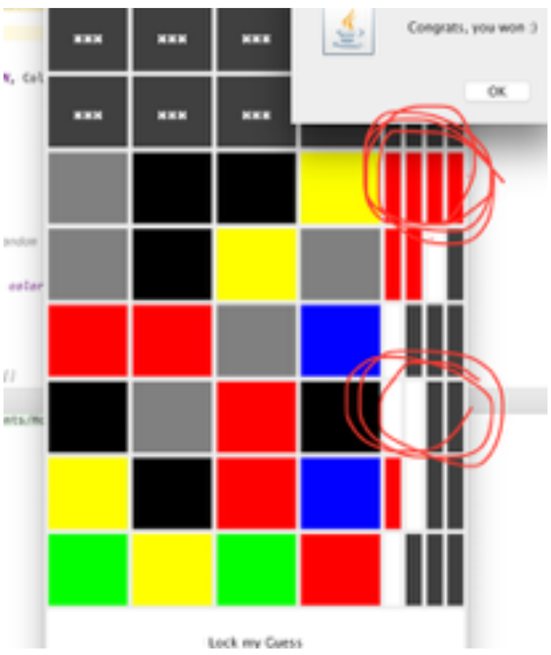
Thanks for a cool semester, looking forward to ECII.

Shoutout to our man Freddy, who provided moral support and helpful insight throughout the project.

See you in January,

Nicos and Bruno

PS: We have no idea how to write this report, please excuse if it comes off as too colloquial. We didn't want to put too much technical jargon in here as we think our comments within the code are quite on point.



PPS (8min before deadline…): found an error in the algorithm. row 3 should have printed 3 whites, not 2. We will have an explanation for that by January.