Interaction and Event Handling

CSE/IT 213

NMT Department of Computer Science and Engineering

"Perfection is finally attained not when there is no longer anything to add, but when there is no longer anything to take away."

— Antoine de Saint Exupery

"I'll create a GUI interface using Visual Basic... see if I can track an IP address!"

— Lindsay Monroe, (CSI New York)

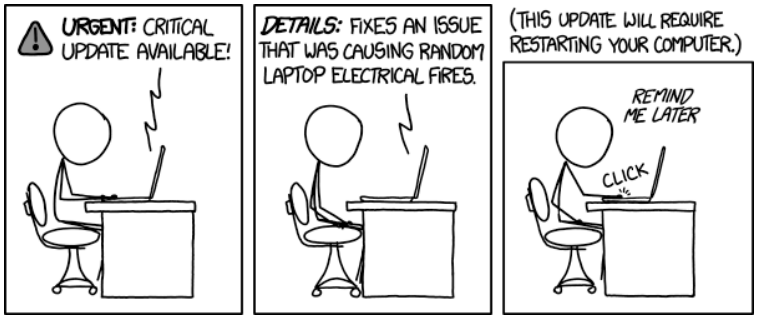
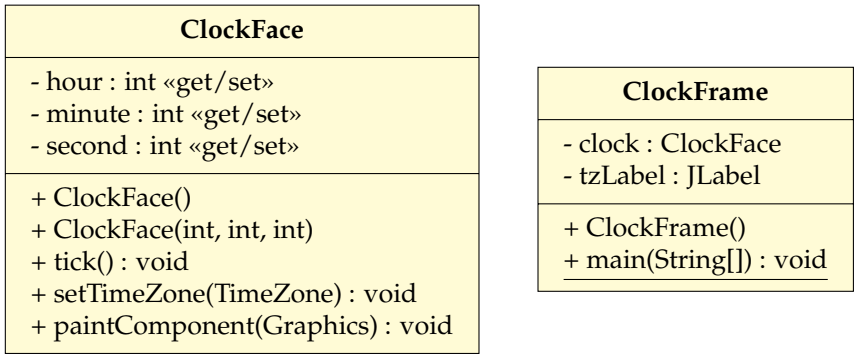


Figure 1: `https://xkcd.com/1328/`

Problems

Problem 1: Clock Timer

In the previous assignment, you wrote `ClockFace.java` and `ClockFrame.java`. In this assignment you will continue these two classes to create an animated clock that ticks every second.



Clock Face

`ClockFace.java` is the same as in the previous assignment, except for the addition of two new methods.

`tick()` moves the time forward by one second, then calls `repaint()`. Increment `second`, but if the result is `60` set it to `0` and increment `minute`. If the minute is then `60`, set *it* to `0` and increment `hour`. And when `hour` gets to `24`, wrap it around to `0`.

`setTimeZone()` takes a timezone object, (from `java.util.TimeZone`), and sets the hour, minute, and second to the current local time in that time zone. To do that, you can pass the time-zone to `TimeZone.setDefault()`, then get the time again from `LocalTime.now()`. Use `TimeZone.getDefault()` to save the system timezone before overwriting it, and set it back to its original value before exiting the method. Call `repaint()` after setting the time.

Clock Frame

`ClockFrame` inherits from `JFrame`. As in the previous version, it adds a `ClockFace` to the main window, and a `JLabel` containing the default timezone to the top of the frame. The private attributes `clock` and `tzLabel` should store these two components.

The next step is to create a new `Timer` object, to call `clock.tick()` once every second (1000 ms). The first argument to the `Timer` constructor is the delay, in milliseconds, to wait between events. The second argument is an instance of `ActionListener` – the code in its `actionPerformed()` method will run every time the timer goes off.

You can set up the `ActionListener` by adding a new inner class to the project. Alternatively, you

can use an anonymous class [1] to instantiate an instance of the interface on the fly. The syntax to declare an anonymous `ActionListener` would be:

```
ActionListener al = new ActionListener() {
    public void actionPerformed(ActionEvent event) {
        // event handling code goes here
    }
};
```

Add a new `JPanel` to the `SOUTH` of the `ClockFrame`, and add three `JButtons` to the panel. Each button will correspond to a different timezone ID (like `"America/Denver"` or `"Europe/London"`). You can choose any three timezone IDs you want, as long as they're valid. The buttons should each have an `ActionListener`, to do two things when the button is clicked:

1. Set the timezone of the `clock` to the timezone of the button
2. Set the text of the `tzLabel` to the display name of the button's time zone

You can convert the ID to a `TimeZone` object, with `TimeZone.getTimeZone(id)`, and use `tzLabel.setText(tzName)` to set the text of the label.

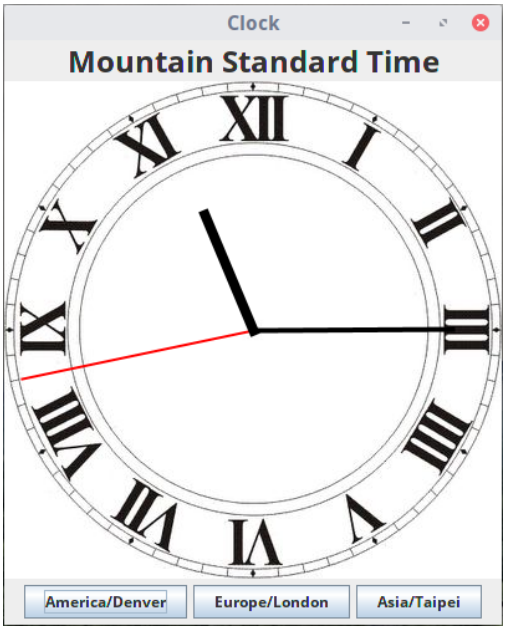


Figure 2: The finished product should look similar to this

Problem 2: Gomoku

Gomoku is a simpler variation of Go, originating in classical Japan. One player takes black, and the other player takes white. The two players take turns placing a stone of their color on an empty

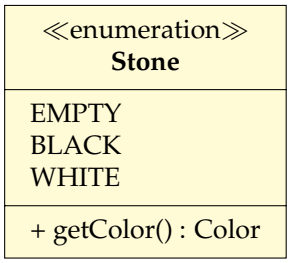
[1] Anonymous classes are similar in concept to lambda expressions – you can create collection of methods on the fly, whereas a lambda creates a single method. For more information, see: `https://docs.oracle.com/javase/tutorial/java/javaOO/anonymousclasses.html`

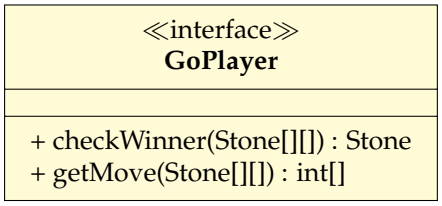
intersection of the grid lines on a Go board. The first player to connect five stones of their color – either horizontally, vertically, or diagonally – is the winner. Essentially, the game is tic-tac-toe five-in-a-row.

In this problem, you will write a simple AI to play Gomoku against a human player. The human player takes black, and gets to move first. The computer will respond to each move by adding a white stone to the board, until somebody wins.

Setup

To get started, add these two auxiliary files to your package:

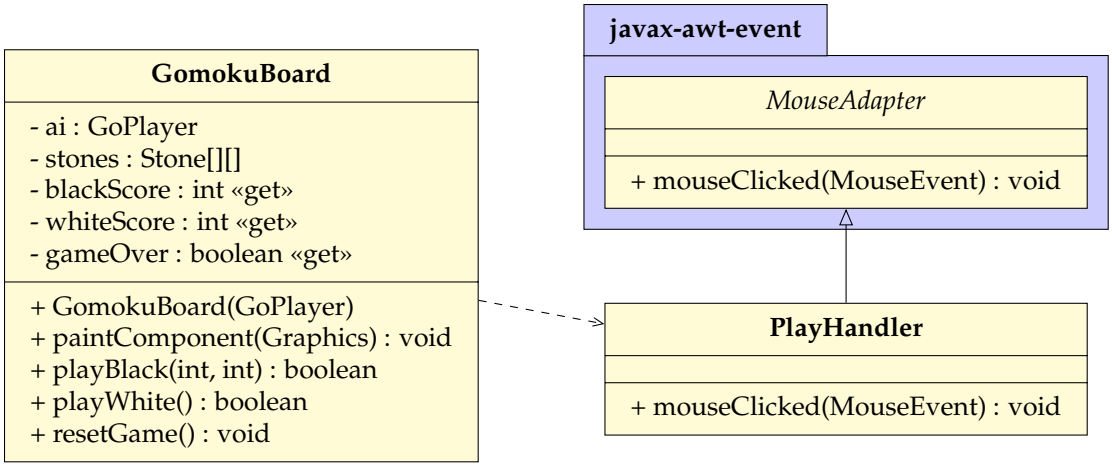




The enumeration `Stone.java` is to separate the concerns of drawing the Go pieces (the *view*) from the logic of using them in your program (the *model*). `getColor()` returns simply `Color.BLACK`, `Color.WHITE`, or `null`, depending on the value of the enum.

`GoPlayer.java` is simply an interface for Gomoku AI, requiring two methods, `checkWinner()` and `getMove()`. The advantage of using an interface here is that it allows you to test out different AI strategies, without having to change any code in `GomokuBoard.java`.

Gomoku Board



The class `GomokuBoard.java` is a continuation of the previous assignment's `GoBoard.java`. It also inherits from `JComponent`, and most of the drawing logic will be the same.

The constructor takes a `GoPlayer` as an argument, and sets `ai` to that value. It also sets the preferred size of the `JComponent` to 720×720, and adds a `new PlayHandler` as a mouse-listener. Set both player's scores and `gameOver` are to `0` and `false`, respectively, and initialize `stones` to be a 19×19 array with every entry set to `Stone.EMPTY`.

`paintComponent()` renders the contents of the `stones` array graphically. First, read `bamboo.jpg` and set it as the background image. Then draw an 18×18 grid of squares – each square has a width and height of 35 pixels, and the lower left corner of the entire grid is positioned at $(45, 45)$. For every color in the stones array, if `stones[j][i]` is not `EMPTY` it fills in a circle of the corresponding color on the board. Center each circle at the point $(45 + i \cdot 35, 45 + j \cdot 35)$, and set the radius to 16.

Note: If this method worked in `GoBoard.java`, there should be no need to change it!

`playBlack()` takes two integers, `i` and `j`, and attempts to add a black stone at `stones[j][i]`. If that space is empty, set the value to `BLACK` then call `repaint()`. If either coordinate is out of range (from 0 to 18), or if there is already a stone at that position in the array, do nothing. Also do not allow black to play if `gameOver` is set to `true`. Return `true` if a move was played, and `false` otherwise.

`playWhite()` passes the `stones` array to `ai.getMove()`, which should determine which position to play on the board. If the AI returns a valid pair of coordinates, `{j,i}`, then set `stones[j][i]` to `WHITE` and call `repaint()`. However, if the array is empty or the coordinates are invalid – if they are out of range, or that position on the board is not empty – do nothing. Also do not allow white to play if `gameOver` is set to `true`. Return `true` if a move was played, and `false` otherwise.

`resetGame()` resets every element of the `stones` array to `EMPTY`, sets `gameOver` to `false`, and calls `repaint()`. The player's scores are left alone, in case the user wants to start a new game.

Gomoku AI

Write a class called `GomokuAI.java` which implements the `GoPlayer` interface.

`checkWinner()` takes the 19×19 array of stones, and searches for any five-in-a-row streak of black or white. You will have to scan the board to search for sequences in four directions: horizontal, vertical, diagonal-up, and diagonal-down. If a row of five is found, then the player for that color has won the game. In that case, return the `Stone` with that player's color to indicating that they've won. Otherwise, return `Stone.EMPTY`.

`getMove()` takes a 19×19 array of `Stones`, and returns a pair of coordinates, `{j,i}`, signifying that it thinks the best move for the white player is at `stones[j][i]`. In the edge-case where there are *no* empty spaces on the board, return an empty array.

You are free to implement any strategy you think will give the computer player a good chance at winning. The only strict requirement is that the AI makes a *valid* move – the coordinate it returns needs to be empty. More information about implementing a Gomoku strategy can be found in the appendix.

Play Handler

Create a `private` inner class within `GomokuBoard.java` called `PlayHandler`. The class extends `MouseAdapter`, and overrides `mouseClicked()`. When a user clicks somewhere on the `GomokuBoard` component, Java will pass a `MouseEvent` to your `mouseClicked()` method. This event object will contain, among other things [2], the x and y coordinates of the user's click.

When the user clicks an empty point on the grid, this function will call `playBlack()` on that point. You can use this conversion to convert the coordinates on the `JComponent` to the nearest coordinate on the Go board:

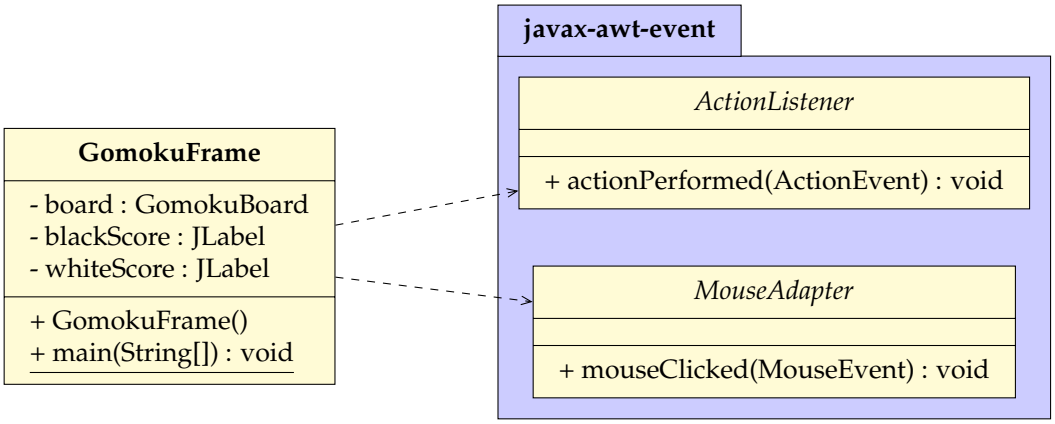
```
int x = (int) Math.floor((event.getX() - 27.5) / 35.0);
int y = (int) Math.floor((event.getY() - 27.5) / 35.0);
```

If `playBlack(x, y)` returns `false`, return from the method without doing anything else. If it returns `true`, call `ai.checkWinner()` to see if black's move won the game. If black has won, increment `blackScore` and set `gameOver` to `true`.

If `gameOver` is still `false`, call `playWhite()` so the computer player can make its move. Then call `ai.checkWinner()` one more time to see if *white*'s move won the game. If so, increment `whiteScore` and set `gameOver` to `true`.

Gomoku Frame

Finally, create a subclass of `JFrame` called `GomokuFrame.java`. The constructor should initialize a `new GomokuBoard` with an instance of your `GomokuAI`, and add it to the frame. Store the board component in the private attribute `board`.



The frame contains two `JLabels`, to display the player's scores. The text of the left label should be `"Black: " + board.getBlackScore()`, and likewise for the white score on the right. Add both labels to a `JPanel` (with a 1×2 grid layout) and add the panel `NORTH` of the `board` component. To the `SOUTH` of the `board`, add a `JButton` that says `"New Game"`.

[2] `https://docs.oracle.com/javase/8/docs/api/java/awt/event/MouseEvent.html`

Once the layout is set up, you need to add event handlers to the window. The first is an `ActionListener` to add to the `JButton`. Pass an anonymous class to `button.addActionListener()`, rather than creating a new inner class. Write the event handler so that when the button is pressed the game is reset with `board.resetGame()`.

Next, add a new mouse listener to the `board` component. Pass an anonymous `MouseAdapter` as an argument to `board.addMouseListener()`. One listener already plays moves for black and white when the board is clicked. This second listener runs *after* that, and simply updates the text of the two `JLabels`. If a player won after the previous move, then that player's score will be incremented in the top panel.

Finally, write a main() method that creates a new `GomokuFrame` and displays the window to the user. The user should be able to play a move by clicking an empty grid point on the board, and see the computer respond with its own move. The score panel keeps track of how many times each player has won, and the `"New Game"` button allows the user to clear the board. The program should exit when the window is closed.

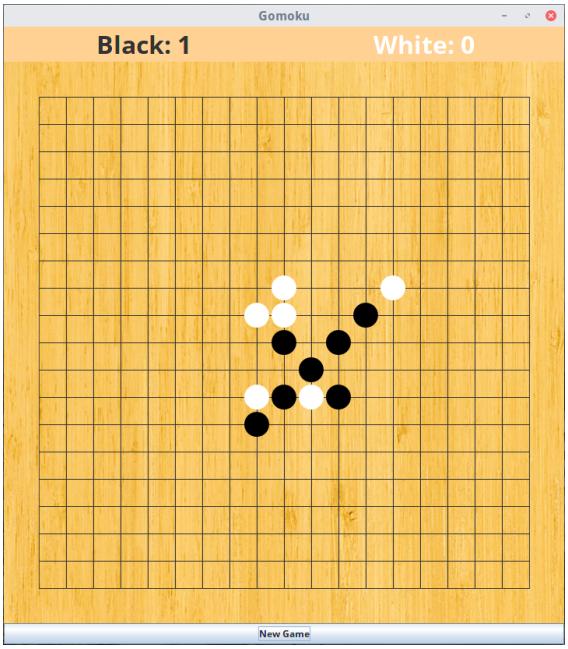


Figure 3: The finished product should look similar to this

Submission

Make sure that you have Javadoc style comments for every class and method in your source code, as described in Homework 0. Document any unresolved bugs in the Javadoc comments for each of your classes. When you are satisfied that your code is complete, create a TAR file containing all of the source code for this assignment called:

`cse213_<firstname>_<lastname>_hw8.tar.gz`

Upload your submission to Canvas before the due date.

Appendix: Gomoku Strategy

For the Gomoku AI, it would technically be valid to just choose a random empty coordinate on the board. However, since you already need to search the board for straight-line patterns in order to implement `checkWinner()`, it should be relatively easy to re-use that code to search for other important patterns in the game.

A good Gomoku AI should be able to detect *forcing* moves. These are moves that a player is forced to make, in the sense that failing to play them is essentially the same as forfeiting the game. In particular:

- If there is a run containing exactly four white stones and one empty space, then white *must* play on that empty space (to win the game)
- If there is a run containing exactly four black stones and one empty space, then white *must* play on that empty space (to block black from winning on their next move)
- If there are three black stones in a row, and there is an empty space on both ends, then white *must* play on one of those empty spaces (to block black from forcing a win in two moves)
- If there are three white stones in a row, and there is an empty space on both ends, then white *must* play on one of those empty spaces (to force a win by the next move)

No matter the strategy, the computer needs to scan the board to find patterns in horizontal, vertical, and diagonal directions. You may find it useful to add some private methods to cut small traces out of the array:

```java
private static Stone[] horizontal(Stone[][], int j, int i) {
    if (i < 15) { // [j, i] ... [j, i+4]
    } else { // [j, i] ... [j, i-4]
    }
}

private static Stone[] vertical(Stone[][], int j, int i) {
    if (j < 15) { // [j, i] ... [j+4, i]
    } else { // [j, i] ... [j-4, i]
    }
}

private static Stone[] diagonalRight(Stone[][], int j, int i) {
    if (i < 15 && j < 15) { // [j, i] ... [j+4, i+4]
    } else if (i >= 4 && j >= 4) { // [j, i] ... [j-4, i-4]
    }
}

private static Stone[] diagonalLeft(Stone[][], int j, int i) {
    if (i >= 4 && j < 15) { // [j, i] ... [j+4, i-4]
    } else if (i < 15 && j >= 4) { // [j, i] ... [j-4, i+4]
    }
}
```

Figure 4: One approach is to examine the immediate surroundings for each position on the board

This might make it easier to detect the forcing moves listed above. For example, if one of these methods returns `{BLACK,BLACK,BLACK,EMPTY,BLACK}`, then you know you *have* to play on the position corresponding to the fourth element of that array.

While scanning the board you can also keep track of shorter streaks or other patterns to help your AI build up a list of moves to choose from. In general, playing on the end of a black streak is *defensive*, and extending a white streak is *offensive* – the strategy for detecting and deciding between these moves is up to you.