Assignment 4: Flexible Boxes

- **This assignment is to be done individually.** Do not show other students your code, do not copy code found online, and do not post questions about the assignment online. Do not use solutions from previous semesters, courses, or offerings. Please direct all questions to the instructor or TA.
- See the marking guide for marking details.
- Assignment must be done in **IntelliJ;** submission generated **using Zipper addon**

1. Assignment Goals

- Demonstrate how inheritance allows the same code to work on multiple types of data.
- Demonstrate how composition allows runtime modification of behaviour.
- Demonstrate how narrow interfaces can improve cohesion and decouple code.
- Gain experience implementing a REST API.

2. Shapes

In this assignment you will implement a system for drawing boxes, using blocks (cells), in a graphical user interface (GUI). With the help of the provided GUI, your model will:

- Read a picture description out of a JSON file.
- Redact all existing objects (boxes) to hide information.

The GUI also supports writing a text representation of what is drawn to the screen. This is efficient for marking because we can diff each student's output to the sample solution.

The GUI (provided) includes the following classes:

- `Main`: Create the module & UI; starts app.
- `GUI`: The graphical user interface. Depends on your model, which will create and manage boxes.
- `PicturePanel`: Manages the process of asking shapes to draw themselves onto the canvas.
- `Canvas`: Draw to the screen, one cell at a time.

There are also two interfaces you'll need to use:

- `ShapeModel`: The main interface your model will implement so the GUI can interact with your code. See the sample `TrivialShapeModel` for an example.
- `DrawableShape`: Each of your drawable shapes (i.e., your box objects) must implement this interface so that they can draw themselves onto a `Canvas` when needed.

See the `TrivialShapeModel` file for an example of how to use these classes.

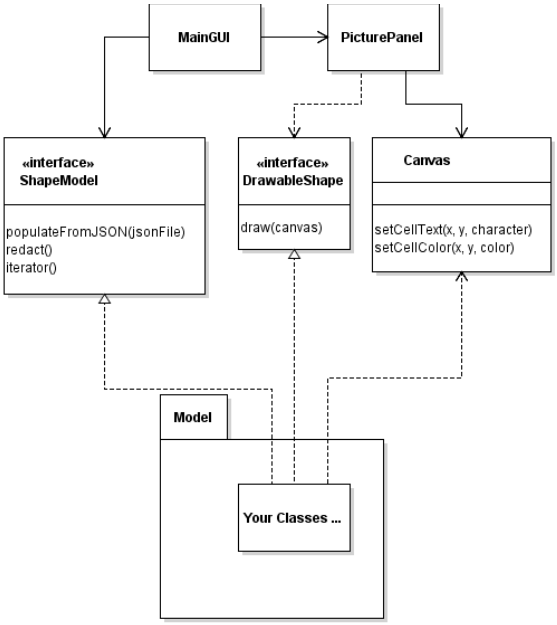

Figure 1: UML class diagram of provided code

2.1 Program Function

The `GUI` uses your model to show some "picture", which your code will read from a JSON file.

Implement the following features:

- When the user clicks on the "Load from JSON…" button, the GUI will call your model's `populateFromJSON()` method. You must use the description in that file to instantiate the necessary objects for the boxes.
- When the user clicks on the "Redact" button you must change all existing box objects to:
  - have a border of '+'s
  - be filled with 'X's
  - be light gray background
  - (There is no way to un-redact, other than to re-open the JSON file).
- The "Export to File" button exports the contents of the canvas to disk. This is fully implemented already, and will be used during marking to quickly ensure solutions are correct.

JSON File Format

- Each JSON file is an array of box shape descriptions.
  - `top`: The row number of the top of the box. Row 0 is at the top.
  - `left`: The column number of the left of the box. Col 0 is at the very left.
  - `width`: Number of columns wide; >= 1
  - `height`: Number of rows tall; >= 1
  - `background`: The type of background. Will be one of:
    - "`solid`": Entire background of box, including border and middle, is the `backgroundColor`.
    - "`checker`": Make a checkerboard pattern with the top left cell being `backgroundColor`, alternating with white to make a checkerboard.
    - "`triangle`": The top-right half of the box will be filled in with `backgroundColor`; the bottom-left will be white. Color the main diagonal the `backgroundColor`. Due to variations in computing what cells would be on the diagonal line, only square boxes will be tested for exact behaviour with the triangular colouring; however, your code must work on rectangular shapes and correctly fill in the top-right triangle.
  - `backgroundColor`: The colour for the background. Will be one of: "`white`", "`light gray`", "`gray`", "`dark gray`", "`black`", "`red`", "`pink`", "`orange`", "`yellow`", "`green`", "`magenta`", "`cyan`", or "`blue`".
  - `line`: The type of line to use for the border. Will be one of:
    - "`char`": Draw the border with the `lineChar` character.
    - "`ascii line`": Draw the border with ‖, ═, ╚, ╝, ╔, and ╗. The box you draw should look like it has a double-lined border. Use ■ for any box with width or height 1.
    - "`sequence`": Draw the border with the numbers 1 through 5. Starting in the top left with a 1, count up to 5 as you go around the box clockwise. When you get to 5, the next character will be 1. Do not reset counting between sides of the box.

- `lineChar`: A string, one character long, for the line character (used by the above "`char`" border line format). When needed, will always be 1 character long.
- `fill`: The type of fill character/text for the middle of the box. Will be one of:
  - "`solid`": Use the first character of the `fillText` string to fill each cell inside the box (not including the border).
  - "`wrapper`": Use the `fillText` string to fill the box full of text. See sub-section below for details.
- `fillText`: String used for filling the box.

- You can assume that the JSON files will always be correctly formed. However, you likely want to throw an exception if you detect a problem to help debugging.
- You can use GSON or another parsing library.

Word Wrapping

Laying out the text is non-trivial:

- If the message is too long to fit on one line, it must be split across multiple lines.
  - Strip leading and trailing spaces on each line of text in the box.
- To break the long text into parts, first attempt to break on a space if there is one; otherwise fill the entire line inside the box with text and break mid-word as needed.
- Each line of text must be centred horizontally inside the box.
  - If there are an odd number of extra spaces on the line place the extra space before the text, such as `"  Hi "`.
- Text starts at the top row inside the box.
- Text must not overlap the box's border.
- If there is more text than will fit inside the rectangle then some text will not be displayed.

Constraints

- Effectively use OOD to create well encapsulated classes.
  - **Favour composition over inheritance.**
  - **Favour narrow interfaces** (see assignment goals section!)
  - **Respect the Open-Closed principle**: adding new features should mainly involve writing new classes rather than deeply changing existing ones.
    Hint: adding a new border style is such a feature.
- When triggering the redact operation, you must use the existing box objects which you created when populating the system from the JSON file. These objects must be changed at runtime to change their border style, fill style, and background style. *You may not create new box objects when redacting*; however, you may create other objects, if needed, to be part of the redaction process as you modify the existing box objects.
- Separate your model from the UI code.
- There must not be much duplicate code in your program. Don't Repeat Yourself! (DRY)
- *Hint: Don't name anything "Box" because it will conflict with a class in Java Swing.*

2.2 Sample Outputs

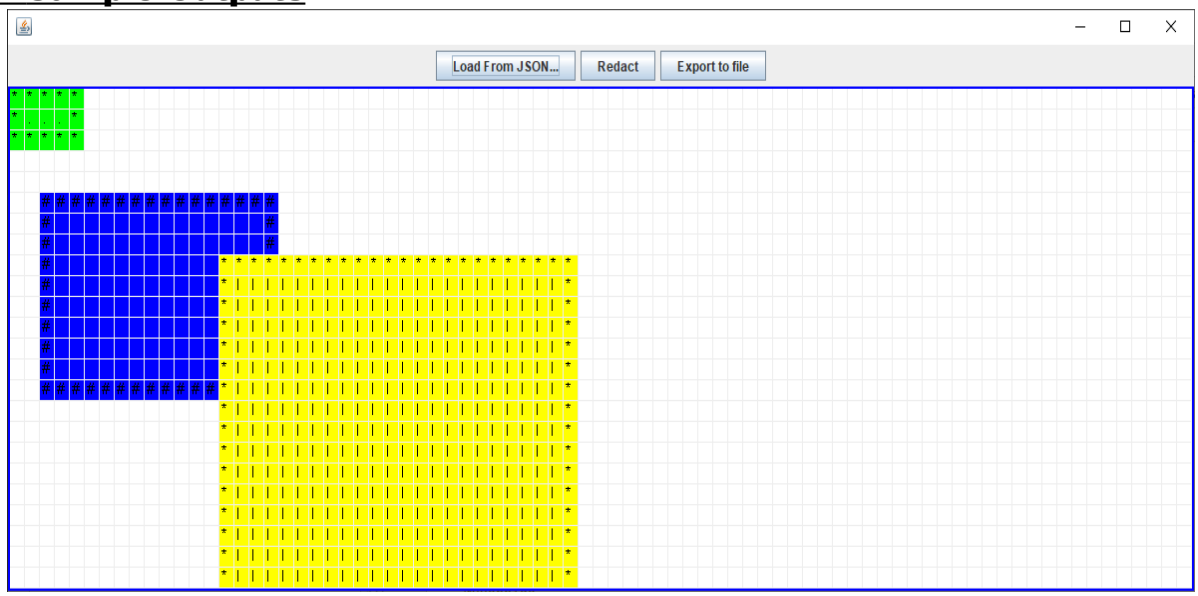

Figure 2: Sample program output for simple.json

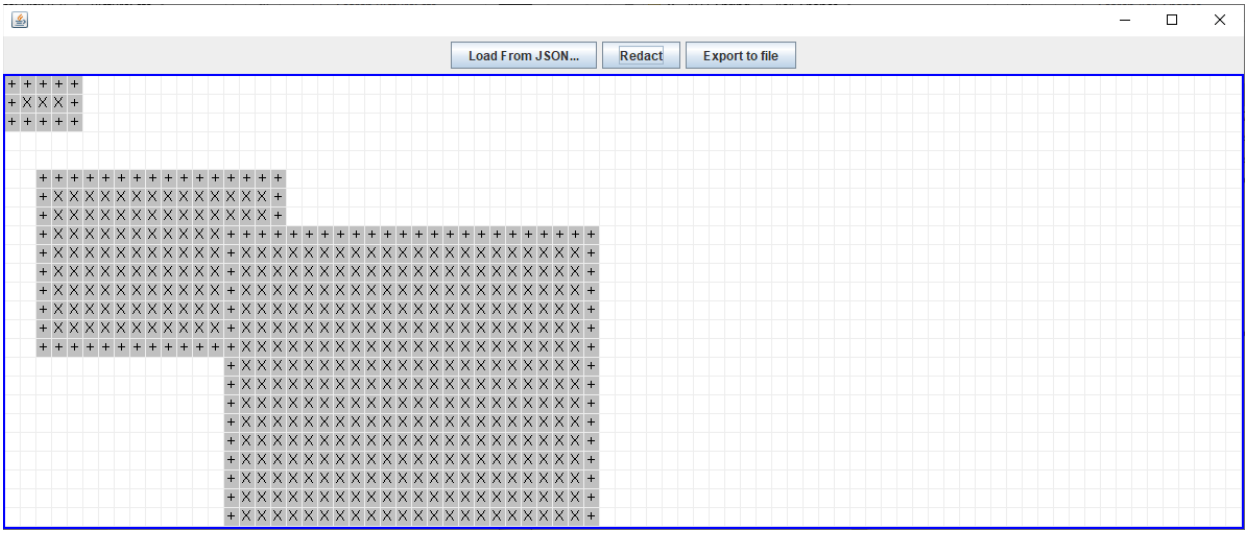

Figure 3: Sample program output for redacted simple.json

2.3 Implementation Suggestions

Suggested implementation order:

1. Create a model and hard-code a few boxes with a simple '*' outline.
2. Create a mechanism to, at runtime, change the border.
3. Implement the "redact" callback function for the boarder.
4. Handle the JSON files to create your boxes, supporting only the boarder to start with.
5. Add support for colour and fill (character / text). Ensure it is runtime changeable (redact).
6. Export pictures from sample data, compare to provided samples.

3. REST API for Assignment 3's Game

In this part of the assignment you will be creating a Spring Boot REST API for your assignment 3!

The course website provides a fully implemented web client front end that you can add to your project. Create a new Spring Boot project and extract the provided web client files into a `/public` folder in the root of your project.

When you run your project, Spring Boot will automatically serve up files in the `/public` folder to the web browser. So you should be able to browse to it at http://localhost:8080

The course website provides a number of resources to help you develop a Spring Boot server implementing the necessary REST API to support the web client:

- REST API document detailing the end points and data
- Data classes that the web client expects to be sent to it. These are data transport objects (DTO): nearly empty classes that just expose data. You will need to implement the static factory methods in each to fully populate an instance of a class before sending it to the web client. You should not modify any of the field names or types in the DTOs because this may cause problems for the web client.
- Video of full game being played, plus some tips on debugging.
- POSTMAN file with many of the HTTP requests configured.
- Full source code to the web client.

You should not need to modify the web client; however, you can if you desire. It should function in the same way it does in the video.

You may use either your assignment 3 solution (either from your individual work, or if you worked with a partner), or the posted sample solution. No change in marks either way. If your assignment 3 had bugs that prevented it from working correctly, then you should likely use the sample solution.

Tips:

- Create a Spring Boot project and copy in your model from assignment 3 (or sample solution).
- Create a `/public` folder for the web client; copy in the provided web files.
- Create a new Java package to store the provided API DTO classes.
- Have your controller class store a list of games (i.e., instances of your model). Have your REST API look up games in this list whenever needed.
- You may assume that the server processes one HTTP request at a time, but must support multiple games going on at once (i.e., in different browser tabs…)
- You may need to edit the model to support any new fetaures this version of the game requires.
- Don't use the `MakeSpringPrettyPrintJSON.java` file: it messes up serving the *`public/`* folder (causes a "Whitelabel Error Page" to show); just delete the file if present.

4. Deliverables

Submit two ZIP files to CourSys: https://coursys.sfu.ca

4.1 Shape

ZIP file of your project for part 1. See directions on course website.

4.2 WebApp

ZIP file of your project for part 2. See directions on course website.

Please remember that all submissions will automatically be compared for unexpected similarities.

