Assignment 4: Shapes and Game UI

- Late penalty: 10% per calendar day (each 0 to 24 hour period past due), max 2 days late.
- **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 via Piazza discussion forum.
  - You may use *ideas* you find online and from others, but your solution must be your own from this semester.
- See the marking guide for marking details.
- Assignment must be done in **IntelliJ**, and submission must be generated via File → Export to Zip file.

1. Shapes

In this part you will implement a system for drawing basic shapes using blocks (cells) in a graphical user interface. A `Canvas` class is provided to handle the basic drawing. Your shape classes will use an inheritance hierarchy to build up functionality without repeating code. The required shape class hierarchy is shown in Figure 1.

You will create a `Shape` interface and write classes that implement your `Shape` interface. Your concrete shape objects (`Triangle`, `Rectangle`, `TextBox`) will be able to draw themselves into the provided `Canvas` class.

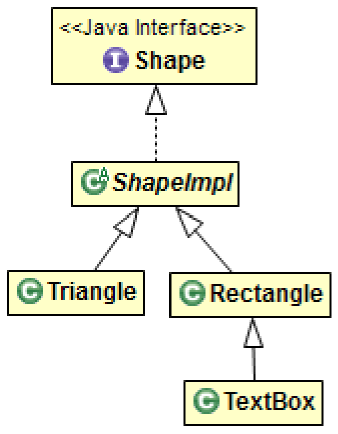

Figure 1: Shape class hierarchy.

1.1 Provided Code

The `MainGUI` class is the foundation of your application.

- It creates a GUI (graphical user interface).
- It exercise all your shape classes to draw some "pictures"

Some classes are provided to support drawing to the screen. You will work with a `Canvas` to draw your objects; the rest of these classes you do not need to know much about.

- `Canvas`: draws the boxes (or cells) of the display. Supports making each box a different colour and showing a character in the box. Your shape objects will call this when drawing themselves.
- `PicturePanel`: instantiated by the client code to hold and draw multiple `Shape` objects.
- `CanvasIcon`: a support class used to display a `Canvas` to the GUI.

Figure 2 shows a UML class diagram for these classes.

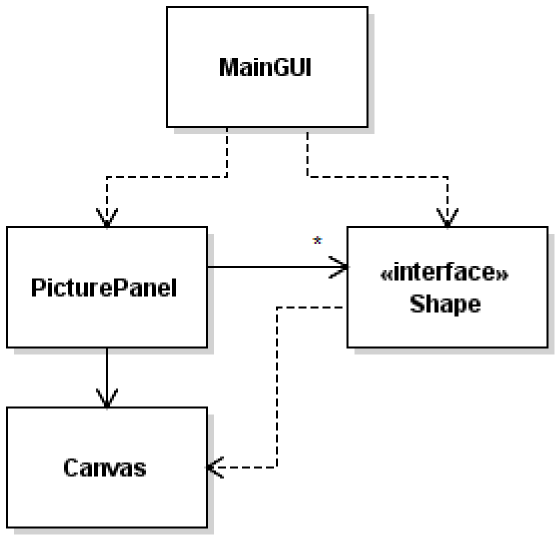

Figure 2: UML class diagram of shapes and client code.

1.1.1 Canvas Demo

Your shapes will use a `Canvas` object to display themselves to the screen.

The `DemoCanvasGUI` program shows how to draw on a `Canvas`. Its `sampleDraw()` method fills in a `Canvas` to look like Figure 3 by doing the following:

- Set the colour of a cell in the canvas.[1]
- Set the character in a single cell in the canvas.

`DemoCanvasGUI` only shows how to draw on the `Canvas`; it does not show making actual shapes. That comes next!

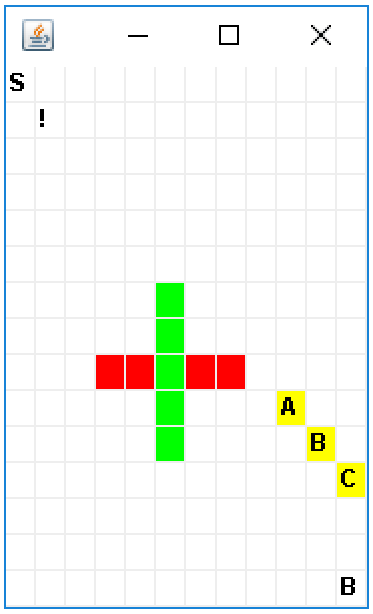

Figure 3: Canvas produced by DemoCanvasGUI

1.1.2 MainGUI

The `MainGUI` program uses your shape classes to show some "pictures". This application is structured to allow you to test each shape one at a time during development. For example, Figure 4 shows its output for the triangle picture.

The application's `main()` calls a number of `makeXYZPicture()` methods to generate some predesigned `PicturePanels`. You should not need to modify this file at all, other than un-commenting out the functions one-by-one as you implement the necessary shapes.

During testing you may comment out the calls to `makeXYZPicture()` to make the UI fit your screen.

When marking, we will likely replace this file with a clean copy of this file to ensure there were no inadvertent edits or add extra test cases.

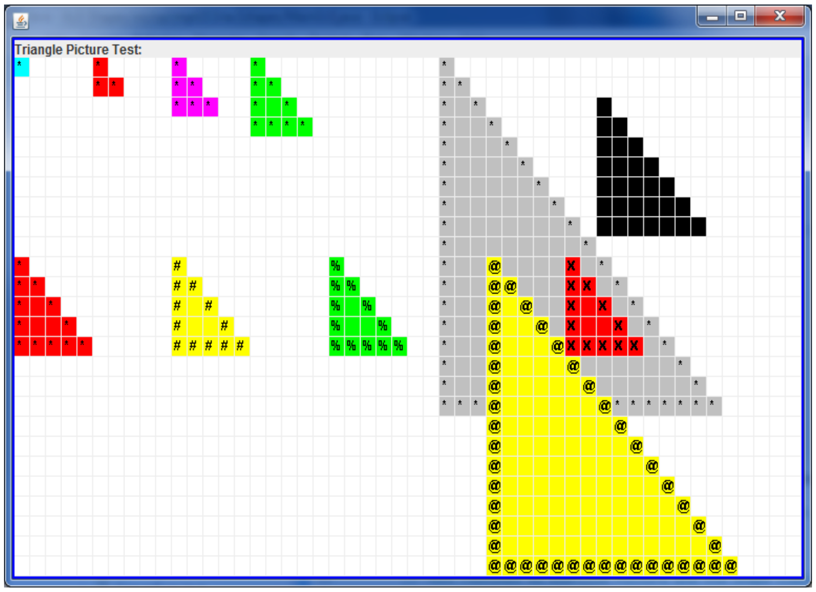

Figure 4: Sample program output for Triangles

To support marking, this program also writes each `PicturePanel` to a file. You can safely ignore these if you like; they are generated so we can quickly diff your output to the reference implementation.

1 Yes, colour has a 'u'; however, since Java does not spell it with a 'u', it is best to be consistent in code.

1.2 Required Object Structure

You must:

- Your project must have the following packages:
  - `package ca.cmpt213.as4`: Provided GUI program files (with `mian()`) will go here
  - `package ca.cmpt213.as4.UI`: Provided classes for drawing to screen (`Canvas`, `CanvasIcon`, `PicturePanel`)
  - `package ca.cmpt213.as4.shapes`: Your shape classes/interface should go here.
- Create a `Shape` interface
  - Each concrete shape must support the following method (required by `PicturePanel`):
    `void draw(Canvas canvas);`
- Create a `ShapeImpl` class which provides a base-level implementation of shared functionality required by derived classes.
- Create a `Triangle`, `Rectangle`, and `TextBox` class.
  - Each of these classes must extend the `ShapeImpl` class.
  - Each of these classes must work with the interface expected by `MainGUI`. Of specific interest is their constructors.
  - `MainGUI` must generate the identical output as shown online for each of the `makeXYZPicture()` test methods.
- There must not be much, if any at all, duplicate code between these classes. You must use inheritance effectively. Don't Repeat Yourself! (DRY)

The next sections of this document give some ideas on how you *may* want to structure your solution.

1.3 Suggested OOD

Figure 5 shows a *suggested* OOD for the shapes in this assignment. Each of these are described below. Both the diagram and the notes below are just suggestions; you may choose to do it a different (but at least as good) way.

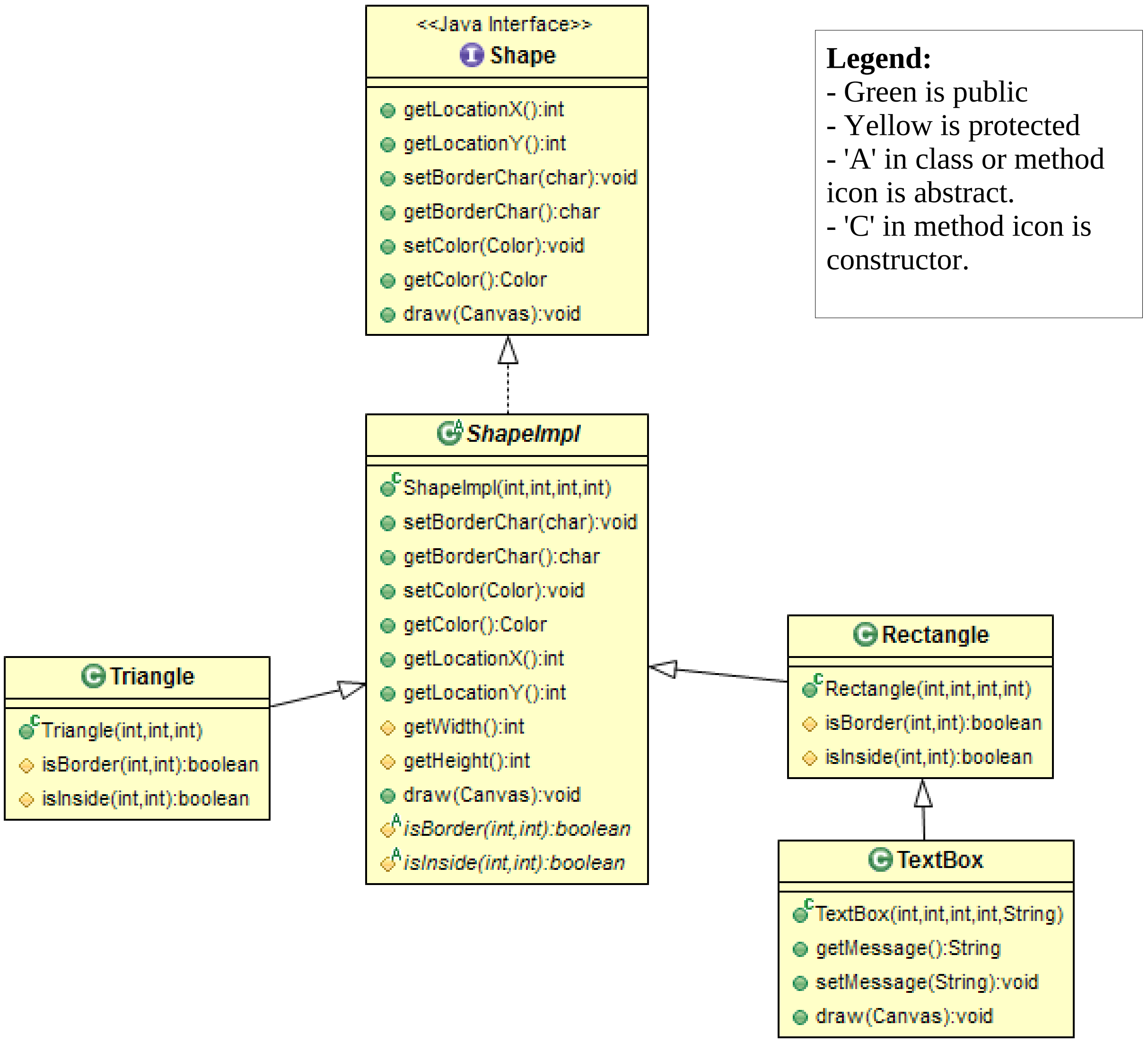

Figure 5: Detailed suggested UML design.

1.3.1 Shape Hints

Create a Java interface named `Shape`, as shown in Figure 5:

- Each shape stores its location as the top-left X and Y coordinates of the shape (integers).
  - The constructor of each class derived from `Shape` should set the coordinates.
- All `Shapes` are drawn with a border (the border character), and have a coloured background.
- The `draw()` function allows any `Shape` object to draw itself onto a passed in `Canvas`.

1.3.2 ShapeImpl Hints

This abstract base class implements much of the functionality specified by the `Shape` interface.

- It stores the location, border character, and colour as private fields.
  - Default character for the border is '*'.
  - Default colour is yellow (`Color.YELLOW`).
- It implements the required getter and setter functions for these fields.
  - If a derived classes needs access to its location, border character, or color it uses accessor functions (encapsulation).
- The constructor accepts the `x` and `y` location, plus the `width` and `height` of the shape.
- The `draw()` function draws the shape into the `Canvas` parameter. However, since this is the abstract base class, it cannot know exactly how to draw the specific shape (such as a triangle). To draw a specific shape, it must "ask" the derived class as follows:
  - The shape's `width` and `height` are used by `draw()` to scan through all positions which are between `(x,y)` and `(x+width-1, y+height-1)`.
  - For each position, `ShapeImpl`'s `draw()` asks the derived class if that position is:
    - a border cell (drawn with a coloured background and the border character) by calling `isBorder(xPos, yPos)`
    - an inner cell (drawn with a coloured background but no border character) by calling `isInside(xPos, yPos)`
    - otherwise it is neither a border nor inside and not drawn at all.
  - Note that this is the "Template Method" design pattern (more in class or online):
    - `draw()` is the "template method".
    - `isBorder()` and `isInside()` are the "primitive operations".
- `isBorder()` and `isInside()` are abstract methods which concrete derived classes override.

1.3.3 Triangle Hints

- `Triangle` draws an equilateral right-triangle where the left and base (bottom) edges are the same size. See sample output.
- `x` and `y` of the `Shape` class are the top left corner of the `Triangle`.
- Constructor takes `x`, `y`, and the triangle's size.

1.3.4 Rectangle Hints

- `Rectangle` stores a width and height.
- `x` and `y` of the `Shape` class are the top left corner of the `Rectangle`.

1.3.5 TextBox Hints

`TextBox` is-a `Rectangle` and adds a message inside it.

- When drawn on a `Canvas`, the `TextBox` places the text inside the box drawn by the `Rectangle`.
  - Use overriding to create a custom draw function, but use your base class's `draw()` implementation.
  - Do not repeat the drawing code from `Rectangle`! You wrote it once; it works; use it!
- 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 `TextBox`.
  - 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 `Rectangle` with text and break mid-word as needed.
  - Each line of text must be centred horizontally inside the `Rectangle`.
    - 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 `Rectangle`.
  - Text must not overlap the `Rectangle`'s border.
  - If there is more text than will fit inside the rectangle then some text will not be displayed.

1.4 Implementation Suggestions

Suggested implementation order:

1. Create `Shape` interface and `ShapeImpl` class.
2. Create `Rectangle` class.
3. Test using the `makeRectanglesPicture()` function.
4. Move on to `Triangle`, then `TextBox`.

Carefully compare your output to the sample output on the course website. The provided `MainGUI` should generate identical output with your code without any modification, other than commenting/uncommenting the appropriate `makeXYZPicture()` functions.

2. REST API for Maze Game

In this part of the assignment you will be creating a Spring Boot REST API for your maze game from 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 wrappers: 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 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 **must** use your Assignment 3's solution for the model in this assignment. If you worked with a partner for assignment 3, you can both use the solution. However, you may **not** work together to modify the model for use with assignment 4. If you did not complete assignment 3, or your solution is not usable for this assignment, you may use the sample solution but there will be a mark penalty (see marking guide). If you use the sample solution's model then you must include a `/doc/README.txt` file stating that you are using the sample solution's model.

If your Assignment 3 model had some bugs, then you don't need to fix them as they don't interfere with the game play. For example, it's OK to generate a low quality maze as long as the game is playable (most of the time); no further deductions will be applied for this. On the other hand, if your model was not able to detect player-wall collisions, then that must be fixed as that is a requirement for this assignment.

Tips:

- Create a new Spring Boot project and copy in your model from Assignment 3.
- Create a `/public` folder for the web client; copy in the provided files.
- Create a new Java package to store the provided API wrapper 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 your model to support any additional functionality this version of the game requires.

3. Deliverables

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

3.1 Shape

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

3.2 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.