CPSC 2151 - Lab 3

Due: Tuesday, September 19th before lab || Thursday, September 21st before lab

In this lab, you will practice writing some contracts for an example class. You will also draw some UML activity diagrams using Diagrams.net (formally Draw.io). The submission requirements are different for each part, so please pay close attention. You must work in groups of 3 or 4 students and only submit once on Gradescope per group. Remember, groups don't have to be the same as they were last week.

Part A

You will practice writing contracts for a class for this part of the lab. Create a Java file for the class and provide all constants, private data, method signature, Javadoc, and contracts. Note that you are doing the design work and writing contracts for these, so you do not need to provide the program code.

`Receipt.java`

This class is designed to track the cost for each item a client has bought and provides a method to compute the total amount they must pay. Note that there are two kinds of items: taxable items and items that are exempt from tax. This class must keep track of both subtotals and only add the tax to the taxable items when computing the final total for all items.

The following constructors and methods must be publicly available. They must have the exact names, parameters, and return types listed. You must include all Javadoc documentation and contracts, including any **parameters (param)**, **preconditions,** and **postconditions**.

- Constructor: `public Receipt(double taxRate)`
- `public void addToReceipt(double cost, int quantity, boolean isTaxable)`: add an item(s) to the appropriate subtotal depending on whether the item is taxable or not.
- `public double getNonTaxableSubtotal()`: returns the current subtotal of the non-taxable items.
- `public double getTaxableSubtotal()`: returns the current subtotal of the items that are subject to tax.
- `public double getTaxRate()`: returns the tax rate that will be applied
- `public double computeTotal()`: returns the total amount charged for all items, including any tax that needs to be applied.

Contracts and Formatting

Classes themselves, not just methods, also have contracts. These are known as **Invariants,** but since we haven't had the opportunity to talk about them yet in class, I won't require you do them for this lab.

You need to create contracts for every method in the class (preconditions and postconditions). Contracts should be formally stated when possible. We have not covered much formal notation, but if a parameter named row needs to be greater than 0, then the precondition should say "x > 0" not "[x must be greater than 0]."

Submission Requirements for Part A

Upload the created Java file to Gradescope. Include the name of all group members as a comment at the top of the Java file AND make sure to select their names when creating a submission on Gradescope.

Tips and other requirements

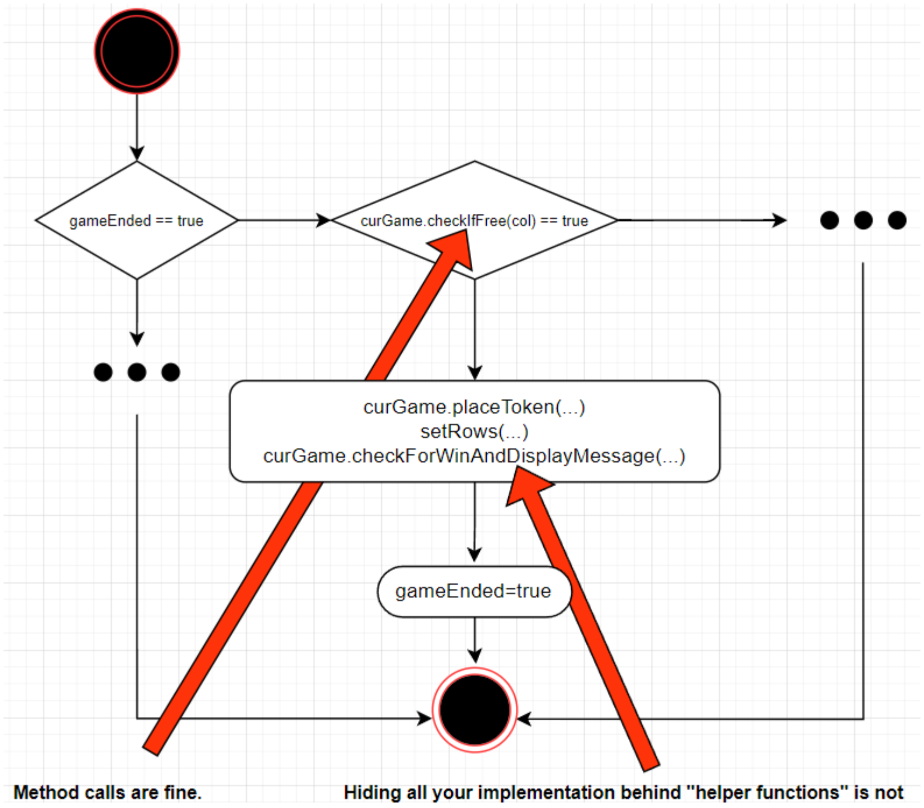
- Remember our "best practices" that we've discussed in class. Use suitable variables, avoid magic numbers, etc.
- Keep in mind that tax rates and money (in this context) cannot be negative.
- The @return clause describes the object/value we're returning.
- Remember, postconditions can (and should) **also** describe return values (in addition to Events and the State of my object). The way we describe a return value in the postcondition for a getter like getLength() would be like:

```
@post getLength = length AND ...
```

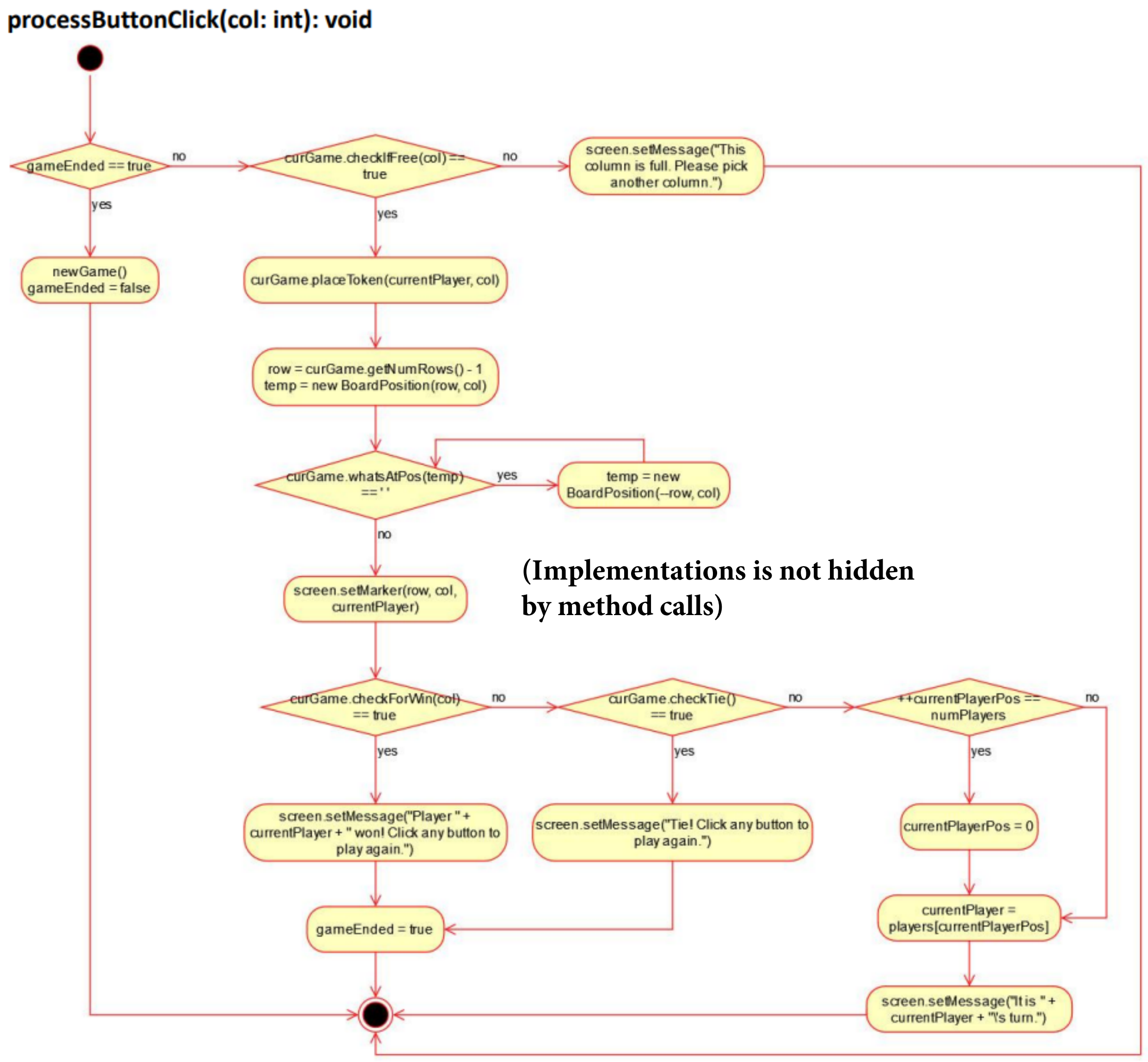
Part B

For this part of the lab, you will be drawing three activity diagrams. First, sketch out all the diagrams on paper. You may think you are finished with a diagram and then think of something you forgot once you have moved on to another diagram. Diagrams.net can take a little getting used to, so it's faster to sketch it out on paper and then make it in Diagrams.net when you are finished. Remember, when drawing an activity diagram, you do not need to have the activities or decisions written in code. You just need to show the entire process to solve it like you would in psudocode.

An "activity" in the diagram can be something that would take several lines of code, as long as that would not include any decisions. An activity can be calling another method, even if that method would contain decisions. We would assume that the method would have its own activity diagram. **However, you cannot delegate the main task to another helper method for this assignment (i.e., call a helper method (that doesn't exist / isn't defined in this project) and just have your primary method like isPalindrome call this helper method).** Examples of this are below. This is **not** acceptable:


gameEnded == true
curGame.checkIfFree(col) == true
curGame.placeToken(...)
setRows(...)
curGame.checkForWinAndDisplayMessage(...)
gameEnded=true
Method calls are fine.
Hiding all your implementation behind "helper functions" is not

This is acceptable:


processButtonClick(col: int): void
gameEnded == true
no
yes
newGame()
gameEnded = false
curGame.checkIfFree(col) == true
no
yes
screen.setMessage("This column is full. Please pick another column.")
curGame.placeToken(currentPlayer, col)
row = curGame.getNumRows() - 1
temp = new BoardPosition(row, col)
curGame.whatsAtPos(temp) == ' '
yes
temp = new BoardPosition(--row, col)
no
screen.setMarker(row, col, currentPlayer)
curGame.checkForWin(col) == true
no
yes
curGame.checkTie() == true
no
yes
++currentPlayerPos == numPlayers
no
yes
screen.setMessage("Player " + currentPlayer + " won! Click any button to play again.")
screen.setMessage("Tie! Click any button to play again.")
currentPlayerPos = 0
gameEnded = true
currentPlayer = players[currentPlayerPos]
screen.setMessage("It is " + currentPlayer + "'s turn.")

(Implementations is not hidden by method calls)

Activity Diagram #1: `ascendingOrderThree`
Create a UML activity diagram for an `ascendingOrderThree` method that takes in three integer values as parameters and outputs them in ascending order to standard out. **Note: You are not allowed to call any kind of sorting algorithm.**

Activity Diagram #2: `isPalindrome`
Create a UML activity diagram for an `isPalindrome` method that takes in a string as a parameter and returns true if it is a palindrome or false if it is not. A palindrome is a string that reads the same backward as forward. **Note: You must use a loop to check in place (i.e., not allowed to create a new reverse of the string and compare it).**

Activity Diagram #3: `calculateGPA`
Create a UML activity diagram for the `calculateGPA` method for the Student class from the last lab, which will return the GPA for the student.
When students take a class, they earn "points" based on their grades and the number of credit hours for the course. Points for the letter grade are:

Grade	Points
A	4
B	3
C	2
D	1
F, I	0

These points are then multiplied by the number of credit hours for the course. For instance, if a student gets an A in CPSC 2150, which is worth 3 credit hours, they will get 4 x 3 = 12 points.

To find the GPA, you add up all the points for all the enrollments for the student and then divide that by the total number of credit hours taken. Courses with a current or future enrollment are ignored. Courses with a grade of W or NA are ignored as well.

Tips and other requirements

- Refer back to my slides on UML diagrams. Many developers believe that UML should just be used as a rough outline for communication, and they skip over some of the notations. Because of that, many online sources may skip over some of the notation as well. **Note:** I don't necessarily think they are wrong, but I think there's a difference between knowing the full notation and ignoring some of it when it's not essential to the current conversation vs. not using the full notation because you don't know it.
- Make sure to export each diagram as an image from Diagrams.net. PDFs can cut off parts of the image from one page to another.
- Make sure your diagrams are readable so they can be graded. This may mean picking a particular layout for the activity diagram so you can fit it into your final document. Since you sketched it out on paper first, you should be able to think of how to arrange it before starting in Diagrams.net.

Submission Requirements for Part B

When you finish all three diagrams on Diagrams.net, export them as images (png or jpeg). You should place these diagrams under the "System Design" section and clearly label the method name. Use the lab report template to format your file. Make sure we can read your diagrams! **Do not make all the diagrams in one Diagrams.net file.** If you do, it will export as one image, which will have to be resized to fit on one page of the document, resulting in diagrams that are too small to read. Each diagram should get its own Diagrams.net file. **Include the name of all group members on the first page.** Do not submit the Diagrams.net files!

Convert your document into a PDF and upload it to Gradescope. Only one member per group needs to submit the assignment, but they should include all group members' names on the submission on the first page of the PDF AND make sure to select their names when creating a submission on Gradescope.

Groups

You can work in groups from 3 to 4 students. The groups do not have to be the same as they were last week. Remember to work collaboratively and do not try divide-and-conquer.