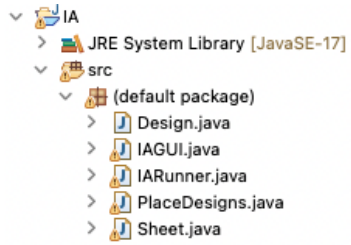
Criterion C: Development

Class Structure and Image

My project involves four main classes. There is a Design class whose instances are created for each unique design the user adds. The Sheet class is created to store the size of the sheet on which the user is attempting to place the designs. The PlaceDesigns class contains several methods that attempt to determine whether or not all the designs can fit on the sheet first through heuristics, then upon the user's request, exhaustively. Finally, the IAGUI class creates the GUI through which the user can input designs and sheet sizes and receive its output.



List of Techniques

- Recursion
- LinkedLists
- Algorithmic Thinking
- GUI Elements
- Encapsulation
- Sorting
- TreeMap

Recursion

Recursion in Java is when a method calls itself to complete a task on a given data set. I use recursion in my code to discover whether the given design will fit on the sheet. Since the possibilities of fitting designs on a sheet form a tree, recursion is used to explore that tree. My exhaustiveFit method explores the entire tree by first checking if a design fits on the sheet, then splitting the sheet into two new sheets, excluding the area that the placed design occupies. If no unplaced designs can fit on any of the sub-sheets, the method tells the previous method that called it to try placing a different design to explore a new branch of possibilities. This process continues until the code finds the branch that allows every design to be placed on the sheet, or the entire tree has been explored and no matter what branch is used, not all the designs can fit. Similarly, the method fitDesigns uses recursion in a similar way and for essentially the same purpose, however, it uses more heuristics to speed up the exploration of the tree by eliminating some branches.

```
public boolean exhaustiveFitDesigns() {
    if (!areasFit) {
        return false;
    }
    boolean go = false;
    boolean designFound = false;
    int sIndex = 0;
    int dIndex = 0;
    while(!go && dIndex < designs.size()) {
        Design d = designs.get(dIndex);
        if(!d.getFit()) {
            designFound = true;
            while (!go && sIndex < sheets.size()) {
                Sheet s = sheets.remove(sIndex);
            original orientation and horizontal cut
                if (s.fitsNoRotation(d)) {
                    d.setFit(true);
                    d.setPos(s.getX(), s.getY());
                    Sheet s1 = new Sheet(s.getWidth(), s.getHeight() - d.getHeight(), (int) s.getX(),
                            (int) s.getY() + d.getHeight());
                    Sheet s2 = new Sheet(s.getWidth() - d.getWidth(), d.getHeight(), (int) s.getX() + d.getWidth(),
                            (int) s.getY());
                    int origLength = sheets.size();
                    addSheet(s1);
                    addSheet(s2);
                    go = exhaustiveFitDesigns();
                    while (sheets.size() > origLength) {
                        sheets.removeLast();
                    }
                }
```

LinkedLists

LinkedLists are a Java Advanced Data Type that is composed of ListNodes which store three variables: the value that gets accessed, the place in memory where the next value in the list is, and the place in memory where the previous value is. This means that values don't have to be stored together in memory, making memory more efficiently packed. It also simplifies inserting and removing values, where the value of the variables for the next and previous location of values get changed, adding or removing a ListNode accordingly. I use a LinkedList to store the designs that need to get placed on a sheet because when they are added these designs get sorted and inserted in descending order of area, which a LinkedList can do quicker than an ArrayList. I also use a LinkedList while exploring the tree to store the smaller sub-sheets that get created after a design has been placed. Once a design has been placed on a sheet, that sheet is removed from the LinkedList, a process that is quicker than it would be using an ArrayList.

```
private LinkedList<Design> designs;
private LinkedList<Sheet> sheets;
```

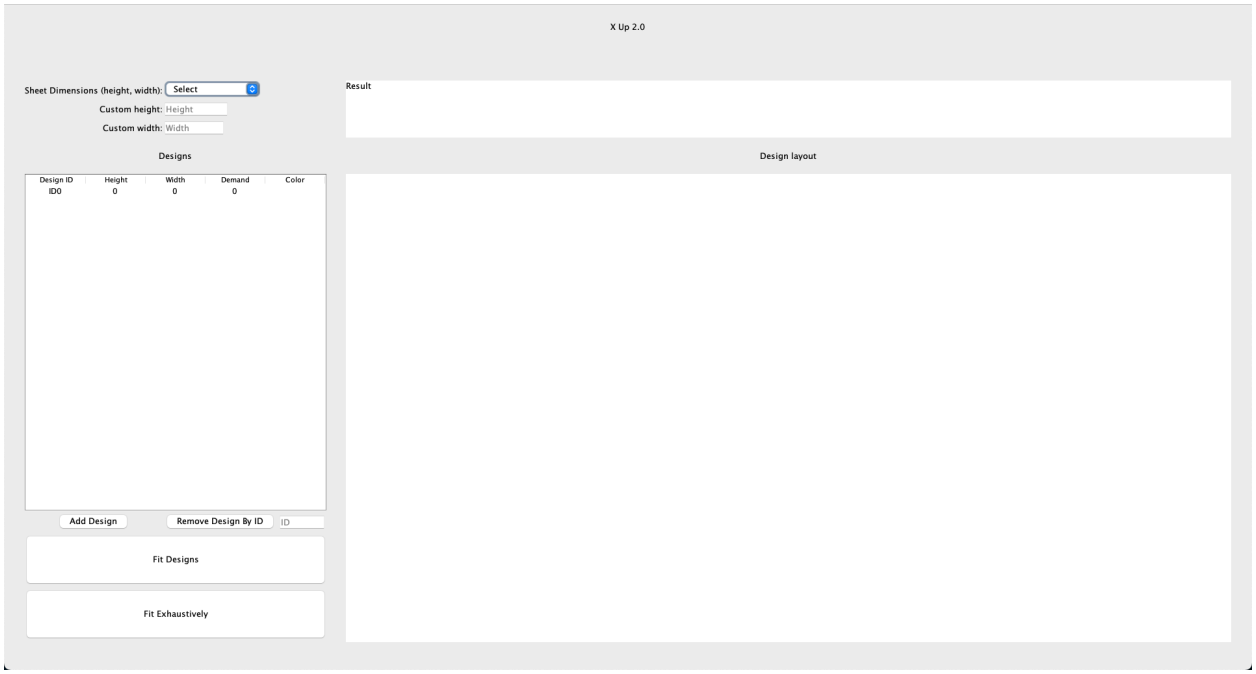
Algorithmic Thinking

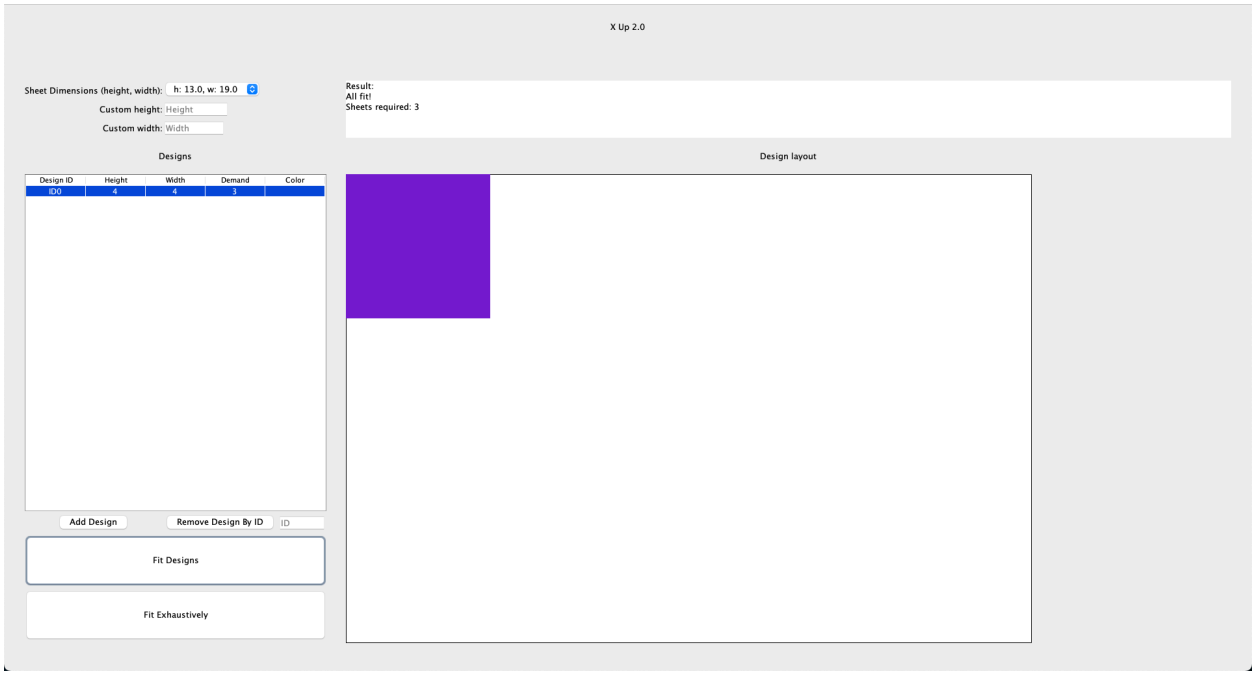
Algorithmic thinking is taking a large problem and breaking it down into easier, and simpler steps. I use algorithmic thinking when deciding which design should be the next to be placed in my method pickDesign. This method loops through every design and identifies the first design that hasn't been placed but keeps going through the designs. For every design the method checks each sheet, looking to see if a dimension of that design fills a dimension of the sheet. If it does then that design is immediately selected to be placed on the sheet, otherwise, the design chosen to be tried on a sheet is the first one encountered that hasn't yet been placed.

```
picks the first design that hasn't been placed, unless another design is able to fill a sheet's dimension
public int pickDesign() {
    Design d;
    int sheetIndex = 0;
    int designIndex = 0;
    int firstDesignIndex = designs.size();
    while (designIndex < designs.size()) {
        d = designs.get(designIndex);
        if (!d.getFit()) {
            if (firstDesignIndex > designIndex) {
                identifies the first design that hasn't been placed
                firstDesignIndex = designIndex;
            }
            checks all the sheets to see if this design is able to fill one of that sheet's dimensions
            while (sheetIndex < sheets.size()) {
                Sheet s = sheets.get(sheetIndex);
                if (s.fitsNoRotation(d) && d.getWidth() == s.getWidth()) {
                    return designIndex;
                } else if (s.fitsWithRotation(d) && d.getWidth() == s.getWidth()) {
                    return designIndex;
                }
                sheetIndex++;
            }
        }
        designIndex++;
    }
    return firstDesignIndex;
}
```

Graphical User Interface (GUI) Elements

GUI is an interface that allows the user to input and view results on a much friendlier screen. Without a GUI inputs and results are run through the Integrated Development Environment's (IDE) console, which only supports text. Consoles are a cumbersome and annoying way to get inputs as each input involves entering a precise word, phrase, or certain number, then hitting enter and either getting a new prompt or the code processing the inputs and returning a result. GUIs streamline this process allowing the user to enter information all at once in different areas then do some action like hitting a button and having the code process all the information at once. I use a GUI to receive information from the user, and then to display the result using visuals, something that the console is unable to do as well as text.





Encapsulation

Encapsulation in Java is where the private variables of a class are hidden from other classes and are only able to be accessed through public methods in the class which returns the value of the variable. This is useful as it prevents variables from being altered when they shouldn't be, which could happen if other classes access the variables directly. I use encapsulation for both my Design class and my Sheet class. Both of these classes have several methods that provide the values of variables should other classes or methods request them.

```java
returns width of the sheet (x-axis)
public int getWidth() {
    return width;
}

returns height of the sheet (y-axis)
public int getHeight() {
    return height;
}

returns total demand of the design
public int getDemand() {
    return demand;
}
```

(Example of encapsulation in Design class)

```java
returns the height of the sheet
public double getHeight() {
    return height;
}

returns the width of the sheet
public double getWidth() {
    return width;
}
```

(Example of encapsulation in Sheet class)

Sorting

Sorting involves ordering a set of values based on some quality. When designs are added to the LinkedList I use sorting to order them from largest size to smallest. The particular type of sorting I used is called Binary Sorting. Binary Sort is a very efficient way of sorting objects where the program eliminates half the possibilities where the object can be inserted on each check until there is only 1 possibility remaining. The result is a sorting method that has a maximum time of log base 2 of n, where n is the length of the list. Of course, for Binary Sorting to work effectively the list that the object is being sorted into must already be sorted, however since the list starts empty and as each design is added they get sorted, and the list is always sorted.

```java
sorts Design d into designs from smallest to largest area and returns LinkedList designs
public LinkedList<Design> addDesign(Design d) {
    int tooLow = -1;
    int highEnough = designs.size();
    while (highEnough - tooLow > 1) {
        // true: designs[too low] > d >= designs[high enough]
        int mid = (highEnough + tooLow) / 2;
        if (d.compareTo(designs.get(mid)) > 0) {
            highEnough = mid;
        } else {
            tooLow = mid;
        }
    }
    designs.add(highEnough, d);
    minDesignDimension = calcMinDimension();
    areasFit = areaCheck();
    return designs;
}
```

TreeMap

TreeMaps are a Java Advanced Data Type that stores a key and value in pairs. Each key is unique and relates to a certain value, which does not have to be unique. I use a TreeMap to identify what size sheet the user selects from a dropdown in my GUI. The keys for this map are the strings that appear as options in the dropdown while the values are certain sheets of different sizes. This TreeMap allows the code to easily read what size the user wants.

```
sheetMap = new TreeMap<String, Sheet>();
Sheet s1 = new Sheet(19, 13);
sheetMap.put(s1.toString(), s1);
Sheet s2 = new Sheet(18, 12);
sheetMap.put(s2.toString(), s2);
Sheet s3 = new Sheet(40, 28);
sheetMap.put(s3.toString(), s3);
Sheet s4 = new Sheet(45, 31);
sheetMap.put(s4.toString(), s4);
```

Word Count: 1044