COMP1406 - Assignment #7

(Due: **Monday, March 13th @ 10 AM**)

In this assignment you will create a re-sizable user interface for an application that will allow the user to design and visualize various floor plans in a building.

(1) The **Model** classes

The main class that represents the model is a **Building** object. Each building contains various **FloorPlan** objects which contain **Room** objects. Buildings also have **Exit** objects.

The picture on the right shows a single **FloorPlan** which contains 4 colored **Room** objects. The 3 red squares represent the **Building**'s **Exit**s. The white areas are corridors on that floor.

Since the goal of this assignment is to create an application with a resizable user interface that allows the user to create floor plans in a building, all of the model classes are given to you as completed. Please make sure that you understand all of the code that you are given here. You MUST NOT add any code to these model classes, nor change any of it. You MUST hand in these unaltered model classes along with your assignment.

An **Exit** object is simply a 2D point represented by a (row, column) coordinate with respect to the **Building**.

```
import javafx.geometry.Point2D;
public class Exit {
    private Point2D     location;   // The row/column coordinate of the exit in the building

    public Exit(int r, int c) {
        location = new Point2D(r,c);
    }

    public Point2D  getLocation() { return location; }
    public void setLocation(Point2D newLoc) { location = newLoc; }

    public boolean isAt(int r, int c) {
        return (location.getX() == r) && (location.getY() == c);
    }
}
```

Room objects maintain an array of square tiles that make up the room's surface area. The tiles are simply 2D points represented by (row, column) coordinates with respect to the **FloorPlan**. Tiles should be added and removed by using the given methods.

```
import javafx.geometry.Point2D;
public class Room {
    private static final int    MAX_ROOM_TILES = 400;

    private int             numTiles;      // The number of tiles that make up a Room
```

```
    private Point2D[]       tiles;         // All the tiles that make up a Room
    private int             colorIndex;    // The color index of the Room

    public Room() {
        tiles = new Point2D[MAX_ROOM_TILES];
        numTiles = 0;
        colorIndex = 0;
    }

    public int getColorIndex() { return colorIndex; }
    public int getNumberOfTiles() { return numTiles; }
    public void setColorIndex(int c) { colorIndex = c; }

    // Add a tile to the room (up until the maximum)
    public boolean addTile(int r, int c) {
        if (numTiles < MAX_ROOM_TILES) {
            tiles[numTiles++] = new Point2D(c,r);
            return true;
        }
        return false;
    }

    // Remove a tile from the room
    public void removeTile(int r, int c) {
        // Find the tile
        for (int i=0; i<numTiles; i++) {
            if ((tiles[i].getX() == c) && (tiles[i].getY() == r)) {
                tiles[i] = tiles[numTiles -1];
                numTiles--;
                return;
            }
        }
    }

    // Return whether or not the given location is part of the room
    public boolean contains(int r, int c) {
        for (int i=0; i<numTiles; i++)
            if ((tiles[i].getX() == c) && (tiles[i].getY() == r))
                return true;
        return false;
    }
}
```

FloorPlan objects maintain an array of walls that make up the inner boundaries of the various rooms and corridors and the outer boundaries of the building. An array of **Room** objects is also maintained. Rooms should be added/removed by using the given methods. The walls are also to be added/removed using the appropriate methods.

```
public class FloorPlan {
    private static final int    MAX_ROOMS= 12;

    private String          name;               // name of the floor plan
    private int             size;               // # of rows and columns in table
    private boolean[][]     walls;              // Grid indicating whether a wall is there or not
    private Room[]          rooms;              // Rooms defined in the floor plan
    private int             numRooms;           // Number of rooms defined in the floor plan

    // Yep, this is a constructor.  It assumes that floor plans are always square in shape
    public FloorPlan(int rw, String n) {
        name = n;
        size = rw;
        walls = new boolean[size][size];
        rooms = new Room[MAX_ROOMS];
        numRooms = 0;

        // Set the grid to have empty space inside, but walls around border
        for (int r=0; r<size; r++)
```

```
        for (int c=0; c<size; c++)
            if ((r==0)||(r==size-1)||(c==0)||(c==size-1))
                walls[r][c] = true;
            else
                walls[r][c] = false;
}

// Some get/set methods
public int size() { return size; }
public String getName() { return name; }
public int getNumberOfRooms() { return numRooms; }

public boolean wallAt(int r, int c) { return walls[r][c]; }
public void setWallAt(int r, int c, boolean wall) { walls[r][c] = wall; }

// Return the room at this position, if this tile position belongs to that room
public Room roomAt(int r, int c) {
    for (int i=0; i<numRooms; i++) {
        if (rooms[i].contains(r, c))
            return rooms[i];
    }
    return null;
}

// Return the room with the given color index, if there is one
public Room roomWithColor(int index) {
    for (int i=0; i<numRooms; i++) {
        if (rooms[i].getColorIndex() == index)
            return rooms[i];
    }
    return null;
}

// Start a new room at the given location
public Room addRoomAt(int r, int c) {
    if (numRooms < MAX_ROOMS) {
        Room room = new Room();
        rooms[numRooms++] = room;
        room.addTile(r, c);
        return room;
    }
    return null;
}

// Remove a room from the floor plan (assumes the room index is valid)
public void removeRoom(Room r) {
    // Find out which room it is first
    for (int i=0; i<numRooms; i++) {
        if (rooms[i] == r) {
            rooms[i] = rooms[numRooms - 1];
            numRooms--;
            return;
        }
    }
}

// Create and return an example of a FloorPlan object
public static FloorPlan floor1() {
    FloorPlan fp = new FloorPlan(20, "Main Floor");

    int[][] tiles = {
            {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
            {1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,0,1},
            {1,0,0,0,0,0,1,0,0,0,1,1,1,1,0,1,1,1,1,1},
            {1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1},
```

```
                {1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1},
                {1,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,1},
                {1,1,1,1,1,1,1,1,1,1,1,0,0,0,1,1,1,1,1,1},
                {1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1},
                {1,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,0,0,1},
                {1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1}};
        for (int r=0; r<20; r++)
            for (int c=0; c<20; c++)
                fp.setWallAt(r,c,tiles[r][c]==1);

        return fp;
    }
}
```

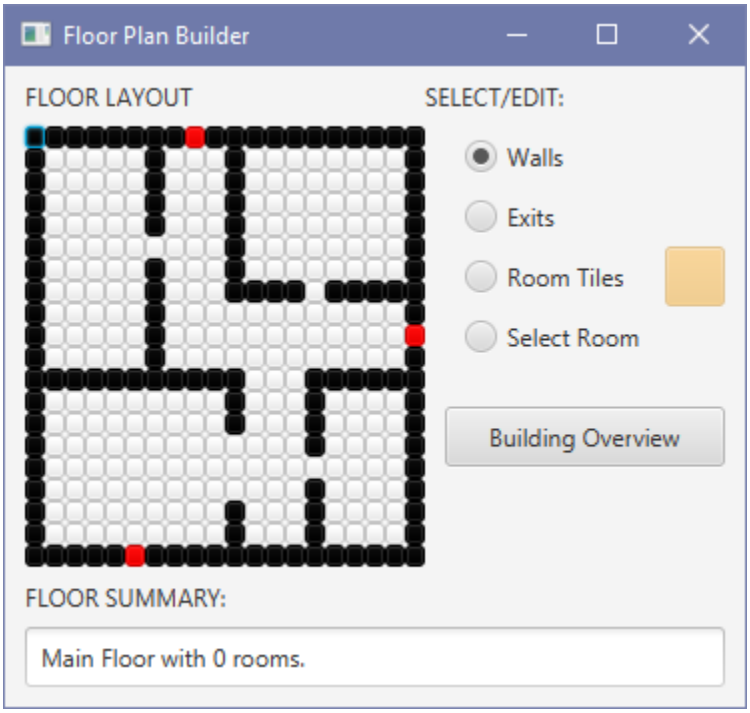
Building objects maintain an array of **FloorPlans** and **Exits** that make the building. **Exits** should be added/removed by using the given methods. An example building with a single floor has been provided, which will be used to test your code.

```
public class Building {
    public static final int MAXIMUM_FLOORPLANS = 5;
    public static final int MAXIMUM_EXITS = 8;

    private FloorPlan[]   floors;           // The floorPlans of the building
    private int           numFloors;        // The number of floorPlans in the building
    private Exit[]        exits;            // The Exits of the building
    private int           numExits;         // The number of Exits in the building

    public Building(int fCount, int eCount) {
        floors = new FloorPlan[Math.max(MAXIMUM_FLOORPLANS, fCount)];
        numFloors = 0;
        exits = new Exit[Math.max(MAXIMUM_EXITS, eCount)];
        numExits = 0;
    }

    // Get/set methods
    public Exit[] getExits() { return exits; }
    public Exit getExit(int i) { return exits[i]; }
    public FloorPlan[] getFloorPlans() { return floors; }
    public FloorPlan getFloorPlan(int i) { return floors[i]; }

    // Get the exit at this location
    public Exit exitAt(int r, int c) {
        for (int i=0; i<numExits; i++)
            if (exits[i].isAt(r,c))
                return exits[i];
        return null;
    }

    // Retirn whether or not there is an exit at this location
    public boolean hasExitAt(int r, int c) {
        return exitAt(r, c) != null;
    }

    // Add an exit to the floor plan (up until the maximum)
    public boolean addExit(int r, int c) {
        if (numExits < MAXIMUM_EXITS) {
            exits[numExits++] = new Exit(r,c);
            return true;
        }
        return false;
    }
```

```
    // Remove an exit from the floor plan
    public void removeExit(int r, int c) {
        // Find the exit
        for (int i=0; i<numExits; i++) {
            if (exits[i].isAt(r,c)) {
                exits[i] = exits[numExits -1];
                numExits--;
                return;
            }
        }
    }

    // Return an example of a building with 1 floor and 3 exits
    public static Building example() {
        Building b = new Building(1, 3);
        b.floors[0] = FloorPlan.floor1();
        b.numFloors = 1;
        b.addExit(19, 5);
        b.addExit(0, 8);
        b.addExit(9, 19);
        return b;
    }
}
```

---

(2) The View

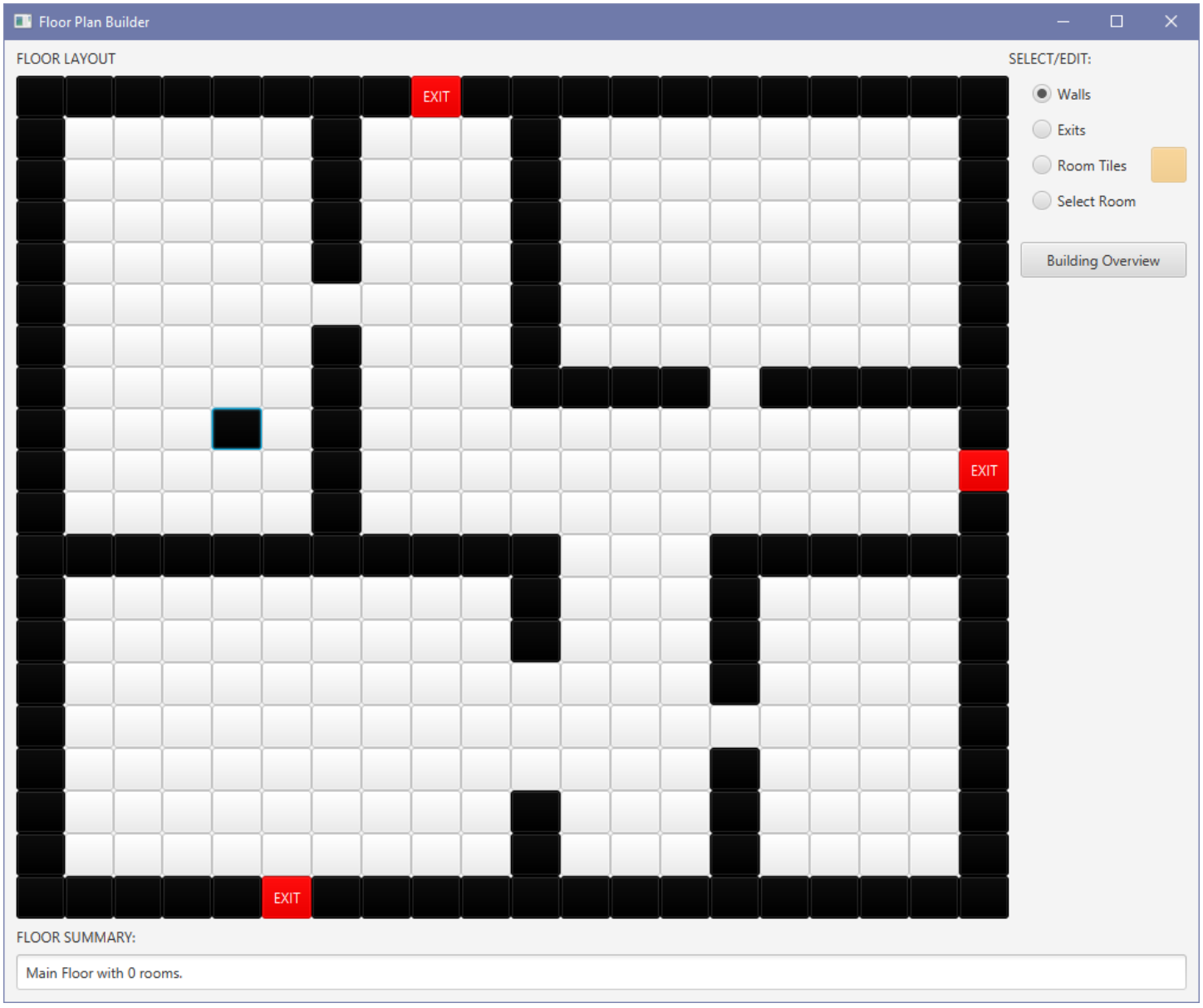


When the application first opens, it should appear with the size and appearance shown here , which displays the first FloorPlan of our `Building.example()` model. The window should be resizable. The next page, shows an image depicting how nicely the window resizes. (Note: Everything looks smaller in the image since the image was shrunk to fit into this document's boundaries).

You must create a class called **FloorBuilderView** which represents the *view* for this application. Your class MUST extend **GridPane**. Here is an explanation of what is required in this view:

- Each square in the grid is a **Button** object. Walls must be shown as BLACK-colored buttons and exits must be shown as RED-colored buttons, while everything else should be shown as WHITE.
- Exit **Button**s must be labeled "Exit", but you won't see the labels when the window is small.
- There are 3 **Label** objects: *"FLOOR LAYOUT"*, *"SELECT/EDIT"* and *"FLOOR SUMMARY"*.
- There are 4 **RadioButton** objects, only one should be on at any time, Walls being the default selection upon startup.
- There is one **Button** object: *"Building Overview"*.
- There is another **Button** (light orange in image) object representing the color of the room currently being created. Note that there is no text on this button but that its color will change when the user clicks on it. This button should only be enabled when the Room Tiles **RadioButton** is selected, and should be disabled otherwise.

- A **TextField** summary field is shown at the bottom of the view.
- All window components MUST be declared **private**. You will need to create some public **get** methods to access them from the controller class.
- The constructor for this class MUST take a single parameter of type **Building** which represents the model that this view will be displaying. You can determine the number of Buttons to use in the grid by using: `model.getFloorPlan(0).size()`
- Pay attention to the margins around the components ... which remain the same as the window is enlarged.
- When the window is resized, the only growth should be on the grid of Buttons as well as the summary field at the bottom of the view, which should only grow horizontally.
- At all times, the summary field should show the name of the **FloorPlan** as well as the number of **Rooms** currently defined on that floor.



(3) The **Controller**

We will now create the **Controller** for the application which incorporates the event handlers. Note that there should be NO EVENT HANDLERS in the view, they should all appear in this class. You must create a class called **FloorBuilderApp** which represents the *controller* for this application. Your class MUST extend **Appication**. Here is an explanation of what is required in this controller:

- The square light orange **Button** that you created in the view which is beside the Room Tiles **RadioButton** will represent the *current room color*. You should use the following fixed array to represent you 12 possible room colors:

```
public static final String[]    ROOM_COLORS =
       {"ORANGE", "YELLOW", "LIGHTGREEN", "DARKGREEN",
        "LIGHTBLUE", "BLUE", "CYAN", "DARKCYAN",
        "PINK", "DARKRED", "PURPLE", "GRAY"};
```

  As the user clicks on this *current room color* **Button**, the Button should change colors to match the next one in the above array. That is, as the user continuously clicks on the button, the color will cycle through the above array, wrapping around to the start again. Note back in the model classes that each **Room** maintains an index into this array as its colorIndex.

- When a grid button is pressed, the action to take will depend on which of the three **RadioButtons** has been selected:
  - If the Walls button is selected, the **Button** that is clicked on will toggle from being a wall to a non-wall or vice versa. Make sure to alter the model and update the view accordingly. If this **Button** clicked on happens to be a tile which is part of a **Room**, then you must change the model to reflect the fact that this tile is no longer part of the **Room** but is now a wall.
  - If the Exits button is selected, the **Button** that is clicked on will toggle from being an exit to being a non-exit or vice versa. Make sure that the model is updated accordingly to remove/add any exits accordingly. As with the walls, if this **Button** clicked on happens to be a tile which is part of a **Room**, then you must change the model to reflect the fact that this tile is no longer part of the **Room** but is now an **Exit**.
  - If the Room Tiles button is selected, the **Button** that is clicked on will toggle from being a tile in a **Room** to NOT being a tile in a **Room**, and vice versa. If the **Button** clicked on is already a wall or an exit, then do nothing, since you cannot make a wall or exit location to be part of a **Room** unless you first remove the wall or exit. Otherwise, the action that you perform will depend on the *current room color*. One of three situations can occur:
    - The **Button** represents a tile that is not yet part of a **Room**. In this case, you need to create a new **Room** object with the *current room color* as its colorIndex, add it to the **FloorPlan**, and add this tile to that **Room**.
    - The **Button** represents a tile that is already part of a **Room** that matches the *current room color*. In this case, you simply remove this tile from that **Room**.
    - The **Button** represents a tile that is part of a **Room** that does NOT match the *current room color*. In this case, the tile should be removed from the **Room** that it is already a part of, and it should then be added to the **Room** of the *current room color*. If this is the first tile for that *current room color*, then you need to

create a new **Room** object with the *current room color* as its colorIndex, add it to the **FloorPlan**, and add this tile to that **Room**.

- If the Select Room button is selected, do nothing.

- Make sure that all of your event handlers follow the clean "change the model" and "update the view" strategy. Each time that you update the view, it would be good to indicate the current FloorPlan number and the *current room color* so that you know what to display. Also, make sure (in your *view* code) that each tile is colored according to the **Room** that it lies in.

- You will not be adding an event handler for the *"Building Overview"* button.

IMPORTANT SUBMISSION INSTRUCTIONS:

Submit your ZIPPED **IntelliJ project file** as you did during the first tutorial for assignment 0.

- YOU WILL LOSE MARKS IF YOU ATTEMPT TO USE ANY OTHER COMPRESSION FORMATS SUCH AS **.RAR**, **.ARC, .TGZ**, **.JAR**, **.PKG**, **.PZIP**.

- If your internet connection at home is down or does not work, we will not accept this as a reason for handing in an assignment late ... so make sure to submit the assignment WELL BEFORE it is due !

- You WILL lose marks on this assignment if any of your files are missing. So, make sure that you hand in the correct files and version of your assignment. You will also lose marks if your code is not written neatly with proper indentation. See examples in the notes for proper style.
