Project 5: Fractals & GUIs ("Bubbles and Bedlam")

1 Understand

1.1 Know the Objectives

After completing this project, you should have a solid grasp of these topics and how to code them:

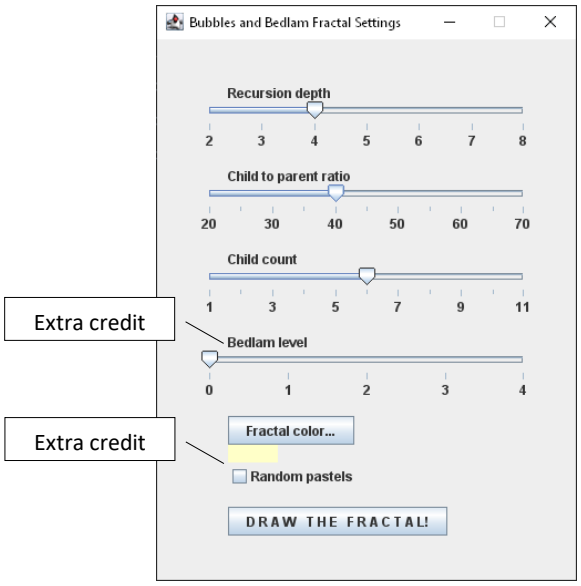
- Fractals (created using recursion)
- Observer design pattern

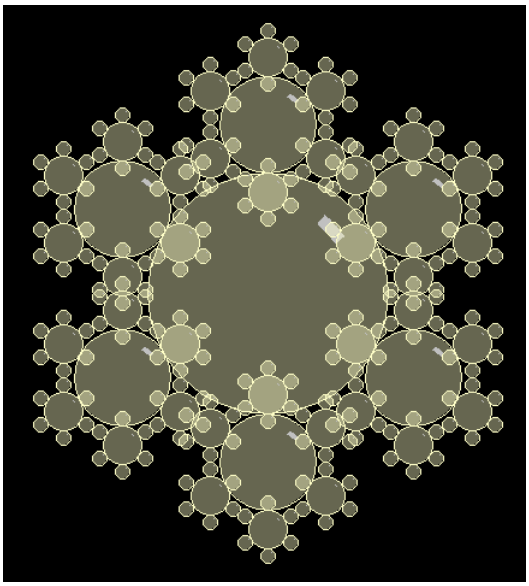
1.2 Understand the Problem: Bubbles and Bedlam Fractals

Fractals are often drawn using recursion, usually starting with one drawn pattern, then branching off to smaller "children" patterns, smaller and offset in some way; this continues until the desired recursion depth has been reached or the objects become too small to bother drawing.

Drawings will start with a central bubble and proceed outwards, with children around the parent (first child directly above, spaced equidistantly apart), touching the parent tangentially.

The GUI lets the user control many aspects of fractal drawing. The expected GUI (yours should be similar) is shown below, at left, with the corresponding output shown at right, which must be drawn on a *single* separate window (i.e., not a new window each time it is drawn), with a black background. When the GUI appears, it should have defaults that render a nice image, with an image shown on the display.





2 Design

Before you code, create appropriate design documentation and obtain feedback. Update designs per feedback, then use them during the rest of the development process and submit them as part of your project. Recommended tool: Violet, which creates the diagram types we need, along with others.

~~2.1 Draw a Class Diagram~~

We will work through this design together; please be prepared to fully participate in that discussion.

~~2.2 Draw an Object Diagram~~

An object diagram is not due for this project.

2.3 Sequence Diagram

Together, we will look at a UML Sequence Diagram that will be useful in understanding how classes communicate in this project. You do not need to submit a version of this diagram.

3 Code

3.1 Understand the Details

3.1.1 Core Classes

- FractalGui is responsible for *gathering user data* via the settings dialog. When the user changes *any* settings, *all* settings information is sent to the Subject (real-time updates).
- FractalGenerator is responsible for *generating fractal data* and storing it in an ArrayList.
- FractalDrawing is responsible for *drawing the fractal* using the fractal data in an ArrayList. This must be a separate class from the GUI. But to make this more universal, have the fractal elements *draw themselves* when asked[1]. Consider a parameter to the draw method to receive drawing space dimensions so the fractal elements can be placed appropriately.
- Main: since this is a full-featured application, you will need a Main class separate from the others. But main here has little code in it; it should be about three code statements.

3.1.2 Observer

We will use the Observer design pattern; the FractalGenerator will be the concrete Subject; the FractalDrawing, the concrete Observer. The FractalGui won't participate in an observer/subject model; it will simply send data to its FractalSubject (via setData) when the user clicks the Draw button.

Remember that in Observer, we can implement incredibly *loose coupling*; none of the three classes mentioned above should hardcode the class names of the other two. Only main needs to know and use these, to do the introductions/handshakes.

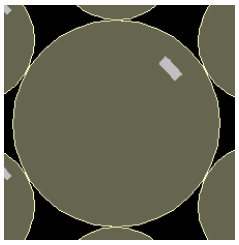
We will use a lazy pull model. "Pull" means that the Observer update message won't be laden with data; it is just an announcement that data is available. The concrete observer will then ask for data (via getData), which will retrieve the ArrayList of FractalElement objects to draw. "Lazy" means that you shouldn't generate data unless it is requested; only when getData is called should you generate it.

3.1.3 FractalElement Interface

Your fractal generator will be creating both Bubble and Arc objects. To abstract this out, we'll use a FractalElement interface that has one method: draw. This method will need a Graphics reference parameter. Also pass it the dimensions of the display window so fractal elements can draw themselves at the right spot (e.g., the first bubble is centered on the display). Implementing classes are responsible for drawing, not calculating per se (except to convert from Cartesian to Graphics coordinates); the core calculation work is done in the fractal generator class.

[1] You would not want the whole application to work only with circles; it should be flexible enough to work with other shapes, though you are not required to do that work.

3.1.4 Bubble (Record)



This class will implement the FractalElement interface. Bubbles are circular (use the Graphics method drawOval), with their outsides being thin, drawn in a solid (non-transparent) color. The insides of bubbles are transparent; find the Color constructor that allows the specification of four parameters (red, green, blue, and alpha). RGB values are percentages of 255, which is a bit unusual. Use a transparency of .4 for all bubbles.

3.1.5 Arc Class (Record)

This class will implement the FractalElement interface. Each bubble has a little reflection arc (use the Graphics method drawArc) on it. Use setStroke (available in Graphics2D, cast your Graphics reference) with a stroke width of radius/10, making larger bubbles have thicker reflection arcs. Calculate the position of the arc by using 75% of the circle's radius, with a start angle of 40 and arc angle of 10, in a light gray color. Make the arcs transparent, too, again using .4 transparency.

3.1.6 Use These

- GUI dialog created in Java Swing
- Graphics drawn on a JPanel (for the display)
- Observer design pattern
- Recursion

3.1.7 Don't Use These

A monolithic approach where everything is in one class (or only a couple of classes).

3.2 Use the Interface(s)

Use the provided interface(s) for the project. Your class(es) should implement these and must exactly follow the signatures in the interfaces. Add helper methods to your classes, of course, but do not alter the interface in any way.

3.3 Implement Other Requirements: Hints

3.3.1 Remember the Basics

- Build your fractal-element generation code incrementally, verifying at each step that the results are exactly as you expect. Failure to do this creates A Huge Mess that you will have to unravel.
- Remember the standard mathematical practice of carrying around the maximum precision for as long as possible, then *round* at the last moment; that is good advice, here.

3.3.2 Understand Coordinate Systems and Measurements

- In the end, circles must be drawn from the upper left-hand corner of their bounding box. But that is a detail that you can put off until the last minute. Instead, focus on the *center* in earlier thinking.
- Standard math focuses on the four-quadrant cartesian coordinate system, with (0,0) at the center. But most computer graphics programming does *not* use that system, but instead sets the origin to the upper left-hand corner. As you move right, X gets bigger; as you move down, Y gets bigger. Ponder that last detail and the implications of having two distinct systems.
- Unit-circle geometry assumes as radius of one. We will start there but will quickly need to move outside of that confinement. Scaling merely requires multiplication.

- In geometry and trigonometry, sometimes we talk about angles, sometimes radians. They are not identical; make sure you are clear on the difference. I recommend focusing on radians throughout.

3.3.3 Other Recommendations

- Build incrementally. Check your work at each step.
- I recommend you stay in Cartesian land (not Graphics land) until later in the process (see below).
- Refresh yourself on unit circle geometry[2] and radians vs. degrees; consider using radians everywhere in your project[3]. This refresher should allow you to calculate the point at which the parent and child circles are tangential; it is also helpful in figuring out the child's orientation. At the start, assume the parent circle is centered at (0,0) for ease of calculation.
- Once you feel confident about your unit-circle results, expand those to cover non-unit circles. This will help you calculate the centers of child circles. This should be sufficient to generate the fractal elements.
- When the elements want to draw themselves, they must first handle the Cartesian-to-Graphics shift; that is a simple mathematical one, but an important one. When the display object calls the fractal element's draw method, have it pass in the width and height of the display, so the FE can place itself properly (or find some other way to make sure the drawing knows about the display dimensions).
- The last thing elements must handle is to calculate their positions in graphics land. Circles, for example, will need to calculate the upper-left hand corner of their bounding boxes, which is simple thing to do given you know the coordinates of their centers and their radii.

3.3.4 Overall Approach

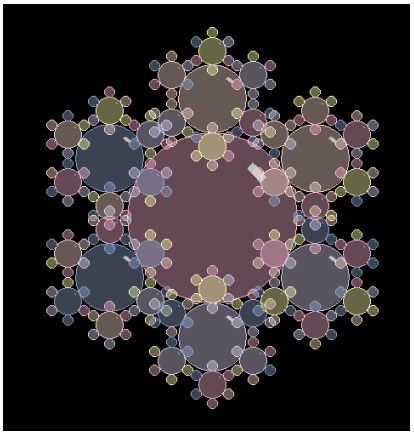
It is tempting, and not unrealistic, to first build the program in a monolithic fashion. For example, you could write code that gathers setting information from the console/keyboard, generates the data, and draws the fractal on a DrawingPanel. But then this code will need to be split apart to fit into the model we will design; that is not a small feat, requiring some mental paradigm shifts. Another approach would be to start with the model and make sure your signaling mechanisms are working properly; then write code in stages and check carefully that the state is right at each step.

4 Extra Credit

4.1 Random Pastels Checkbox

Add a checkbox with text "Random pastels". If the user clicks it, instead of using the current color, create each bubble of a random pastel color from this list (adding transparency, of course):

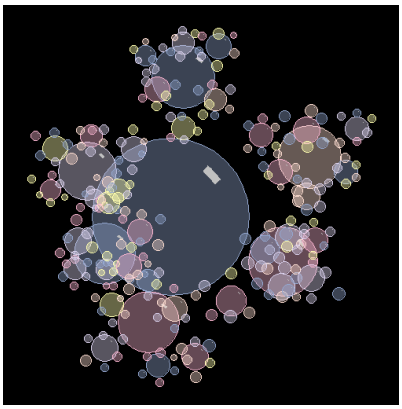
- yellow (255, 255, 176)
- blue (148, 168, 208)
- purple (221, 212, 240)
- pink (251, 182, 209)
- orange (255, 223, 211)



[2] Do an online search for "unit circle geometry," finding resources to refresh your knowledge of the subject. Videos like this one might be helpful.

[3] Research Math.toRadians() if you feel compelled to convert between them (I would personally stick with radians).

4.2 Bedlam



Beautiful, even, pristine bubbles are lovely. But a bit of chaos is fun too! Add a Bedlam control, where level 0 means no chaos and level 4 means maximum chaos.

Explore Random's nextGaussian() and do a little math to turn the number it generates into a usable result. Start with the equation $Z = (X - \mu) / \sigma$ (where Z is standard normal distribution value (returned by nextGaussian()), X is original distribution value (the value we need), μ is the mean, and σ is the standard deviation of the original distribution value). Solve for X.

The bedlam settings should influence the standard deviation. At level 0, standard deviations are 0 (causing perfectly even output). At level 4, the standard deviation should be:

- Child radius – 10% of the child radius
- Child angle – 10% of the typical child angle delta
- Child distance – 25% of the typical calculated distance

5 Document

5.1 Follow the Style Guide

Follow the Course Style Guide, which is linked in the Reference section of the Modules list in Canvas. Failure to do so will result in the loss of points.

5.2 Write JavaDoc

Write complete JavaDoc notation for all classes in this project. This means that an -Xdoclint:all comes back with no errors or warnings. If there are errors or warnings, you will lose points.

6 Test

~~6.1 Write Unit Tests~~

Take a break from testing on this project.

6.2 Verify Code Meets Acceptance Criteria

Work will be returned to you for rework (with a deduction for late work) if it doesn't meet the acceptance criteria listed here:

- …compiles with all required Xlint and Xdoclint command line additions.
- …runs without throwing any exceptions.
- …has no failing unit tests.
- …allows the user to choose fractal options.
- …draws the resulting fractal upon request.
- …has results that are at least a vague facsimile of what is expected.

7 Demonstrate

A working GUI with fractal output is the only demonstration necessary.

8 Measure Success

Area	Value	Evaluation
Fractal drawn/colored properly	15%	Was the fractal drawn correctly, given the settings? Did subsequent drawings look good? Were screen sizes appropriate and friendly?
GUI layout	10%	Does the GUI look like the example? Do widgets function properly?
GUI class implementation	15%	Was the class set up as requested? Was it successfully coded using good practices?
Fractal generation class	20%	Was the class set up as requested? Was it successfully coded using good practices?
Drawing class implementation	15%	Was the class set up as requested? Was it successfully coded using good practices?
Observer/Subject implementation	15%	Was the Observer model properly and successfully used? Were all the handshakes properly coded and used? Was loose coupling implemented as we designed[4]?
JavaDoc	5%	Did you use JavaDoc notation where requested, and use it properly? Did the Xdoclint run come back clean?
Style/internal documentation	5%	Were other elements of style (including the Style Guide) followed?
Extra credit: random pastels	1%	Does the random pastel checkbox work? Does selecting some color using the color chooser uncheck the random pastel box?
Extra credit: bedlam	2%	Does the bedlam control exist, and does it work as requested?
Total	**103%**	

9 Submit Work

Follow the course's Submission Guide when submitting your project.

[4] A good rule of thumb: the three core classes should have no knowledge of the names of the other core classes; they should only use interfaces as static references to their core colleagues.

10 Recommended Milestones

1. Create Observer interfaces.
2. Create three core classes (GUI, Generator, Drawing) in skeleton form to get basic communication working. For example, GUI might have a single button on it, "Draw". Generator receives settings from GUI, notifies observers. When requested, sends an empty ArrayList. Drawing receives empty list, draw a circle in a random spot on the screen.
3. Flesh out GUI. Ensure user settings are sent to the Subject when *anything* changes.
4. Create FractalElement interface, plus Circle and Arc classes that implement it.
5. In Generator, do recursive work, with shapes created and added to the ArrayList.
6. In Drawing, ask each of the FractalElements in the ArrayList to draw themselves.
7. Do extra credit (if time).