# PART #1:

Question 4

1. To begin, the technical debt included in the starter code mainly has to do with the difficulty of drawing the grid with an unordered data structure. The starter code provided us with a set of vertices which was contained within a hash set. The biggest issue with a hash set is that the elements are in no particular order. This presents a challenge as it is very difficult to connect two neighboring vertices when their places in the data structure holding them are totally unrelated. This resulted in lines being drawn totally randomly as our first instinct was to create a segment from the ‘ith’ index to the ‘i+1th’ index. This obviously did not work because, as was mentioned before, the hash set stores the vertices in totally random order. Once potential solution to this was to create a hash function to essentially decrypt the random order being given to the vertices. Since no one in our group has experience with that, we decided to change it to an array list since the data structure will be altered later anyways. The other main source of technical debt in the starter is the fact that it is primarily set up to accommodate a square grid as the vertices are drawn specifically in a square and the variable which generates the x and y values (square size) is tailor made to create squares which will not be viable as we move to irregular meshes.
2. If we were to stay at the immutable data structure level, it would stop us from implementing a lot of functions in the future that would be much easier if we were to change it up. For example, we found that the dots stored as hash tables were not too effective when drawing the segments because it all became very random. We changed the data structure to be an array list as we found having more order in our segments would help us in finishing our first step of business logic. If we were to have an immutable data structure, it would be incredibly difficult to try and work around it instead of just changing our data structure. Even more problems would arise later on if we did not alter the data structure at all, specifically if we were stuck working with unordered vertices and could not store our data as polygons. Once we move on to irregular meshes, if we were still stuck using a set of unordered vertices, it would essentially be impossible to efficiently draw segments between the consecutive vertices of each polygon. By storing the vertices in a polygon, we are able to keep track of which vertices need to be connected (and in what order) which greatly simplifies a process which would not be viable if we had to manually identify each vertex in our polygons which will be randomly generated in future iterations.

# PART #2

Diagram

Description automatically generated

In the visualizer, there is only one class to which we use in any way which is the graphics renderer and therefore the visualizer cannot exist without this class. This class has a single method called render which requires a mesh to be passed through. As a result of this, the graphics renderer must use the data stored in mesh data to render anything at all (in reality it also depends on dot gen to return the mesh but that is too specific for our purposes). Additionally, this one is correct because we see that we do not have to really alter our final Visualizer class. This means that this will be the best diagram we can offer,

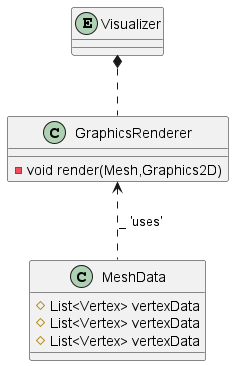
Diagram

Description automatically generated

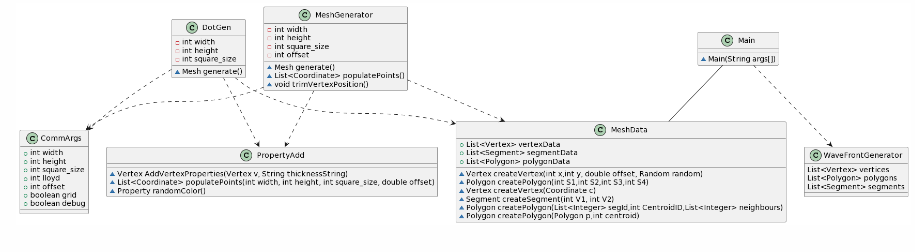
In the generator, the entry point is the main class which instantiates and object of the dotgen class and therefore references it. The dot gen class in turn uses the MeshData class to store what it generates and the MeshData class leverages the DotGen business logic to gather its data. Finally, the generator itself at the end of the day uses the data stored within the MeshData class to generate the actual mesh. Out of the two diagrams this one is not correct. We believe that there is still more to add and this will be evident in our final class diagram.

1. To fix some of our flaws with the starter code, we first changed our data structure that would store each dot. It was a hash table. With has hash tables there is no order to them. It only has two corresponding values, and the tables keys can be in any order. This made drawing the segments impossible and forced us to change everything to an array list. Having an array list allowed for a much better data structure to store our dots and segments. Likewise, we fixed up the problem with having multiple dots being drawn on each other by completely altering the code that was provided and adding if statements for some error that was occurring when drawing the segments.
2. To support requests from the user we made everything a variable that can be easily changed in the command line argument. Likewise, utilized object orient programming to make it sure that if a class was not needed in our function, we were able to just not utilize the class and our code would not be impacted because of the nature of our composition in our code. Since only the classes that can be affected by invariants use composition, this means that our code can be manipulated and changed accordingly.
3. The testing activity that we used helped us a lot. Mainly we used a lot of our testing to check where and how we could have problems arise in our code. However, we didn’t utilize it enough when trying out our code, but this means that we can learn from our error and hopefully utilize it more in the final step of our code. Mainly, we used it to narrow down where the error in our code was and used it to narrow down the area that we needed to work on. Overall, we came to learn the importance of testing and we are now changing our mistakes and testing cases that can occur.

# PART #3



For the visualizer code, we didn’t change anything from step two. This is because our project had no business logic that required us to do anything to the our code that would alter it in any way. Overall, there is one class that creates the render and then uses the data stored in the mesh data to create our graphics. We believe that out of the two diagrams because it is separated into two classes which utilizes interface segregation with our mesh data.



The Generator subproject is mainly composed of the classes DotGen and MeshData. The entry point to the subproject is Main.java, which parses command line options using Apache CLI. Our design attempts to separate the two different mesh types into their own categories, Grid based meshes and Irregular meshes which have their own distinct ways of being generated. Additional classes MeshData and PropertyAdd exist to handle the numerous ways mesh data can be interacted with.

1. When the Voronoi diagram was implemented, the graphic renderer class which is the main aspect of our visualizer code was changed. However, this was not necessarily needed as both the current and previous versions operate by accessing each polygon’s data and generating shapes by drawing segments and vertices using the information contained within each polygon object. The changes made improved our design as it now stores the vertex information in arrays of x and y values and uses the ‘fill polygon’ tool to fill in the entire polygon which makes coloring the polygons much easier. The vertices and segments are drawn in the same way, accessing the data within the polygon, and drawing lines and points with the 2D graphics tools available.
2. Fortunately, the given JTS library already does a little bit of encapsulation for us. We do not need to do the calculations needed to produce the geometry of a Voronoi diagram, the JTS library already has tools which do this. All data, functions and calculations are hidden from us when we generate the Voronoi diagram. Access to this information is being restricted, the only thing we can see is the public interface which tells us what each method does as well as the output that it returns. Additionally, we encapsulated the complexity of creating polygons, segments, and vertices by placing them in a method located inside a ‘Mesh Data’ class so that the actual generator simply must call a method called ‘vertexCreate’ or ‘segmentCreate’ which only require one or two parameters and hide the complexity of actually building these objects.
3. Of course, creating an entirely new kind of mesh comes with its own set of calculations that must be built from the ground up. No matter how streamlined our design, there must be new code created to meet the specifications of where each vertex is to be drawn, under what conditions, what the polygons should look like, etc. However, if this new mesh uses the same principles of polygons, segments, and vertices to be generated, the MeshData class will certainly provide a useful starting point. Given that a grid mesh and an irregular mesh require slightly different methods of creating their mesh, we have used method overloading to provide multiple ways to create vertices and polygons automatically. This offers a decently wide array of strategies for gathering data to produce vertices and polygons and it would hypothetically grow larger and become more comprehensive as more different types of grids were made. Generally, our design encapsulates the process for creating vertices, segments, and polygons and makes adding those to the mesh data structure quite high-level to allow the developer to focus on the logistics of producing the triangular tessellations.
4. Currently we only have a few sources of technical debt which need to be repaid before the conclusion of the project. The first and most significant is that our meshes still draw duplicate segments where polygons touch. This needs to be resolved to streamline the mesh and make it as efficient as possible. A proposed way to resolve this is to implement some sort of function that checks if a vertex/segment already exists with the given properties (v1 and v2 indexes or x and y coordinates for example). The other main source of technical debt is the structure of our code. While the modularity of our code isn’t too bad now, it has room for a lot of improvement. We have made a mesh ADT as well as two mesh generation classes (one for each type of mesh) which provide the data for the ADT and a class which provides the various generator classes with information on the command line argument inputs. However, this should be further modularized as some classes contain too much information. For example, the mesh ADT has methods for adding properties which our team thinks could be further decomposed to create a class which has the sole responsibility of adding properties. Additionally, the dot generation classes are very messy and we are likely going to encapsulate their methods into separate classes so that the dot generation classes are very minimal and just have methods such as (produceVertices(),produceSegments(),AddVertexColor() etc…).
5. Single Responsibility: In our design there are three classes which bear a significant burden in producing the mesh. The first is the DotGen class which is responsible for doing the calculations to determine the position that each vertex, segment, and polygon should be created at as well which vertices should be referenced by each segment and which segments should be referenced by each polygon. This is the DotGen class’s main responsibility (it sort of has a second responsibility of adding properties to mesh objects which it could be argued should be another class but that is not a super significant part of the code). The second class is the MeshData class which is responsible for building and storing all data related to mesh objects and does not try to do anything else, it only has that singular responsibility. The final class is responsible for rendering the mesh as it is the one ‘In the knowing’ since it has access to the completed mesh and can break down its objects into parts that are drawable and uses 2D graphics tools to realize this.

Open/Closed: The main instance in which this principle is applicable to in the Mesh Data ADT class. It contains a space to hold data for all of the different objects of the mesh which will never be modified as long as the structure of building the mesh does not change. Additionally, it contains methods which create these mesh objects given different parameters and parameter types. These are closed for modification as they serve a fixed purpose that is immutable. The method for creating a vertex given a coordinate will never change as that purpose which it fulfills is unchanging. However, given new approaches to creating a mesh, there may be different ways of creating these mesh objects (different parameters and parameter types for example) which need to be implemented to allow some new implementation of a type of mesh to be realized. In this way, the class is openfor modification as new methods can be added to accommodate for any new strategies of generating the mesh that arise.

Liskov’s Substitution: Liskov’s substitution generally references a relationship between classes and subclasses. It states that subclasses should only add information and should never reduce or restrict the attributes of the superclass. By doing this an object of the subclass should be able to do everything the superclass can. In our design, we do not really make use of the subclass concept, so Liskov’s principle is not super applicable, but we will learn from this and try and add this principle into future projects.

Interface Segregation: Our design consists of many different classes, each with its own variety of methods. However, we tried to focus on only putting methods in each class that is necessary for the class to operate. For example, creating a mesh data structure is always going to require storing vertices, segments, and polygons and therefore, the methods for creating them should be contained within a single class (which it is). Similarly, we have a properties class which is generally used for assigning colors to mesh attributes and therefore contains all methods and data required to create and add properties to the mesh elements. The final major way we segregated our interfaces was by making a dot gen class for each specific type of mesh as the methods contained in each corresponding class are not useful for creating the other mesh type and therefore should be separated.

Dependency Injection: The most significant way this principle is realized in our code is when creating the mesh elements. The dot generation classes produce the required information to instantiate a vertex/segment/polygon object so that the class which it belongs to does not have to provide that information itself. Additionally, when generating and assigning properties in the property creation class, the object which the properties are being added to are passed into the class and do not need to be created by the class itself.

# PART 4

1. To start, the biggest thing that went wrong in development was how long it took to familiarize ourselves with the project. Since these builder methods and even the 2D graphics tools were entirely new to us, it took a long time to figure out what exactly all of the starter code did and how it worked. The first part of the project took well over a week and mostly consisted of us being confused and making silly mistakes. Another similar aspect of the development which went somewhat poorly was us spending too long making the grid mesh. After figuring out what was going on in the project we were able to develop the grid mesh and most of its specifications generally quickly, but did not think to look ahead to the third step of the project and realize that it requires entirely different business logic to implement and spent a good portion of time trying to make our grid mesh code modifiable to accommodate for Voronoi diagrams (which is not really feasible as the Voronoi diagrams make use of an entirely different geometry library). In contrast, one major thing that went was our consistent incremental delivery over time. This was likely somewhat forced due to it being a group project but throughout the whole project one member would figure out how to implement a specific feature and then someone else would take that and build upon it to create the next feature, streamline it, or fix an aspect of it that was not working correctly. In general, development was a team effort and it felt as though every time one group member got stuck on an issue another member was able to figure it out which allowed the team to not get stuck on any one aspect of the development for very long which is often a problem in individual projects.
2. Our team believes that version control systems are superior to any other form of file sharing used in the past as it not only shows exactly what was changed each time the version is updated, it also keeps a log of the history of every commit. We found that a big reason this was so useful for us is because it gave us the option to try something really radical if we had an idea for implementing a feature that held the potential to completely destroy our code. If a group member completely ruins their code by trying something, it is totally fine because they can just look at the most recent version of the working product and copy the code from that commit and it was as if nothing happened. In this way, it allows us to be better engineers because we can experiment with new (and potentially very bad) ideas for a design solution which we would not think was feasible to try if we were not able to easily revert our code back to any past working version. Additionally, the version control system eliminates the need for meetings and consolidating work which slows down the design process. If a group member has an idea for solving a problem in the code, they do not need to run it by the rest of the team, they can just get work and if their idea works, update the version control system and just like that, they have contributed to the project. On the other side of things, keeping the version up to date, every group member immediately reaps the benefits of any other member’s work as the feature implemented or bug fixed is updated in their code as if by magic (assuming they are pulling frequently).
3. In this assignment, our team’s structure was one which had very little structure. All of the features of the project as well as the report were somewhat of a free-for-all with everyone attempting random features and no concrete person who had the responsibility to complete it. Furthermore, the order in which the features were delivered was very out of order. While we do think this strategy has some merit, allowing all team members to contribute to anything they think they are able to, it led to some issues with our codes design and each member’s understanding of said design. Given that the delivery was not purely incremental, the code required some restructuring to implement features that should have been delivered long ago. Since responsibilities were vague, many group members were unsure of what task to do and did not quite understand code they had to work with as another member had done it instead (we aren’t great at commenting). In the future, it would be helpful to assign responsibilities for features concretely to each member and potentially even give a member an entire domain of the project to oversee to maximize fluency with the code they are interacting with.
4. Nathan: 34% -> Report work, supplementary code,

Aidan: 34% -> Minimal report work, took on main workload of part 3,

Jeremy: 32% -> Report work, supplementary code, scripting, bonus.