Part 1

Describe the technical debt associated with the starter code (generator and visualizer).

The technical debt associated with both the generator and visualizer starter code consists of multiple items. In both the generator and visualizer starter code, nothing is abstracted, and the code is lengthy and difficult to read. Within the generator directory, the vertices are stored within a HashSet, which we can't keep track of since they have no order. These technical debts will be required to be reimbursed in the future to reduce the need for further modification and ensure a product is produced with a quality infrastructure.

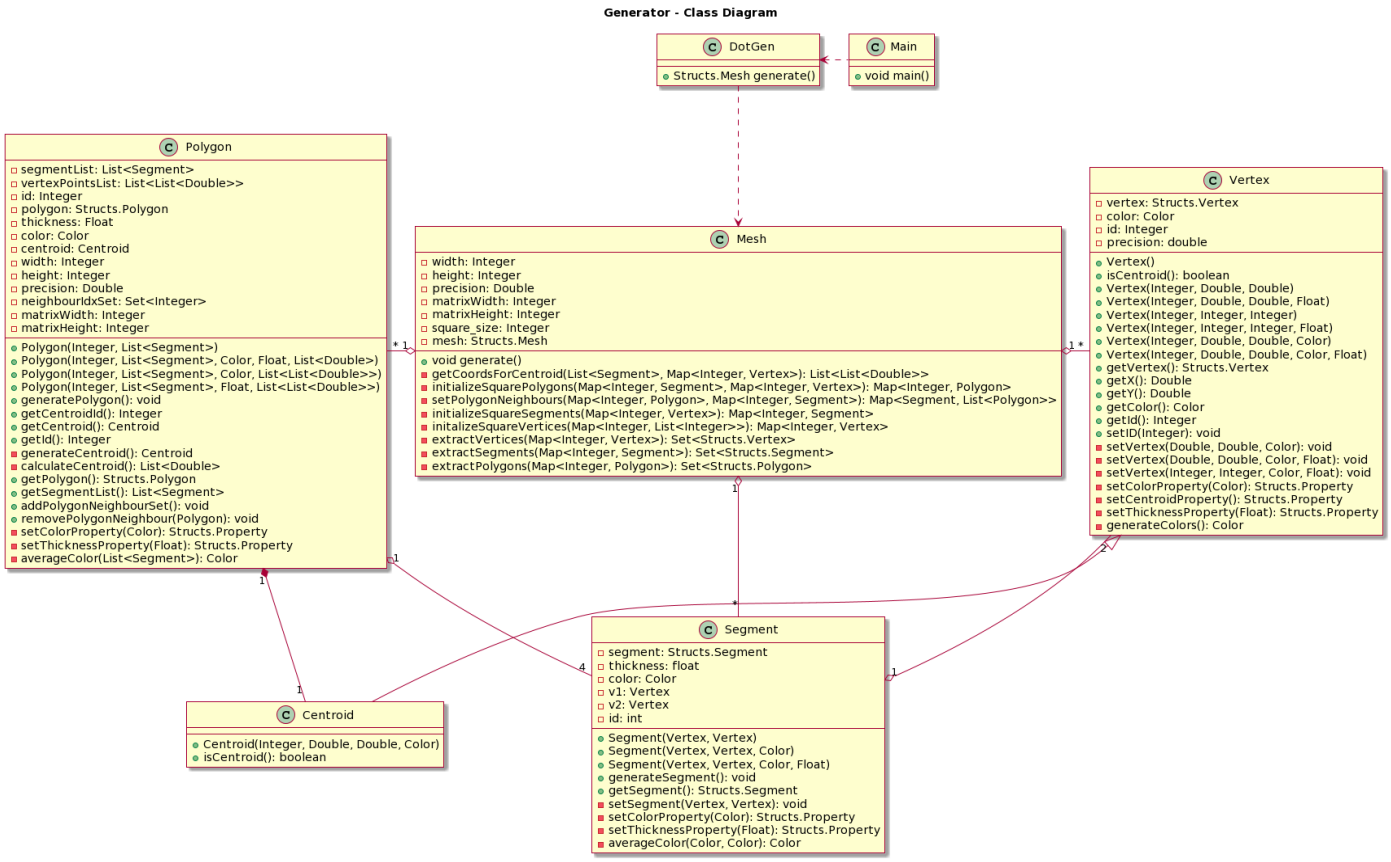
Reflect on the difficulties in controlling the evolution of both parts if staying at the immutable data structure level.

If we stay at the immutable data structure level it will be quite difficult to control the evolution of both the generator and visualizer. For instance, the vertices are an immutable data type. In the starter code, the vertices were initialized without colour. Adding colour to the established vertices requires new vertices to be created due to the immutability. Also, it required a lot of refactoring to simply generate the missing segments. This required us to create multiple new methods, and to rethink the entire flow of generating new vertices/segments, and associating properties.

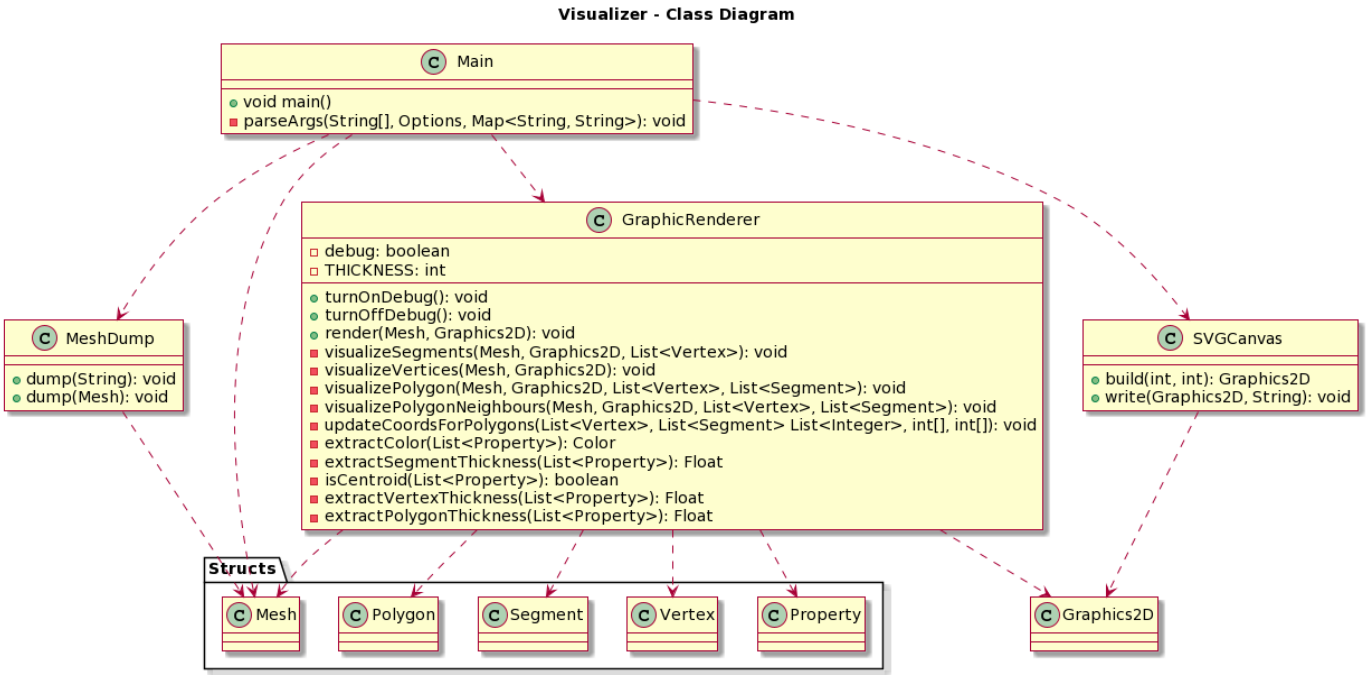
Part 2

Part 2 Class Diagrams and Explanation

Generator:



Visualizer:



Based on the class diagrams shown above, we believe our design for part 2 is correct for a number of reasons. This design attempts to ensure single responsibilities are assigned for each class within the program. Rather than combining multiple responsibilities to one mesh, the modular design of our program supports easier implementation of new features and requires little modification if needed. The class diagrams also demonstrate high cohesion and modular decomposition as all the separate modules work together towards achieving a goal. Additionally, minimal coupling is also ensured as the main goal was to only associate classes with their necessary counterparts. As seen in the lectures, we wanted to minimize the fluff (YAGNI) when we came to our design decisions. We also strategized to make the visualizer sub-project universal to any mesh shape as long as the mesh followed according to the strict IO sub-project guidelines. In conclusion, we believe that this design is on the right track to becoming a robust and optimized mesh generator. We will continue to develop and improve this design in part 3 so that it may adhere to all SOLID principles and fine-tune the design development route that we chose.

How did you fix the flaws identified in the starter code?

The main flaw with the starter code was that nothing was abstracted, primarily due to the fact that a lot of the starter code did not follow Demeter's Law (one dot rule). Many of the function calls had a chain of additional function calls which made it very hard to see what was primarily being implemented. We fixed this main flaw through creating separate Vertex, Segments, and Polygon classes that would handle the low level components of creating those elements through functions that were easy to read. For instance, when we create a new Vertex, we pass in data (x, y coords, color, thickness) then that info gets passed into a "setVertex" method that contains

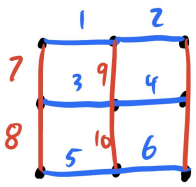
the low level methods of generating a vertex (utilizing the IO library). We had similar design patterns for segments and polygons alike. This is sort of like the decorator pattern as we had a base object (the Structs Geometries) and then added functionality onto of it without changing its core behaviour (it's still a vertex, but with higher cohesion, lower coupling and easier to work with). Additionally we created a Mesh ADT that combined the data structures defined above (Vertex, Segments, Polygons) and provided an abstraction layer that supports the specifications from the IO library to fully generate the mesh.

For the Mesh ADT, how do you ensure in the code the invariants supporting the requests from the user

The main two invariants that the user requests are preventing redundancy of geometries, and maintaining consecutiveness.

Redundancy refers to avoiding creating overlapping vertices, segments, or polygons. For our vertices, we initialize our vertices using the "initializeSquareVertices" method and store each vertex in a hashmap with a unique id as its key to prevent duplicates. Similarly, we utilize a similar storage principle with our segments and polygons by using a hashmap.

To maintain consecutiveness, we drew horizontal segments first, followed by vertical ones. An example of such ordering looks like the figure to the right.



Through using that method of storing the segments, we can use mathematical patterns to store segments in a consecutive order that abides by the invariant by only looping through the horizontal segments. In the above example, we would only iterate through all of the blue segments (1..6), and for each segment we would compute the other 3 segments that make up the polygon. For segment 1, the polygon would be (1,7,3,9) and this maintains the consecutive order as all the segments are adjacent to each other.

By adhering to our design choices, we can guarantee that the invariants supporting user requests are upheld in the Mesh ADT.

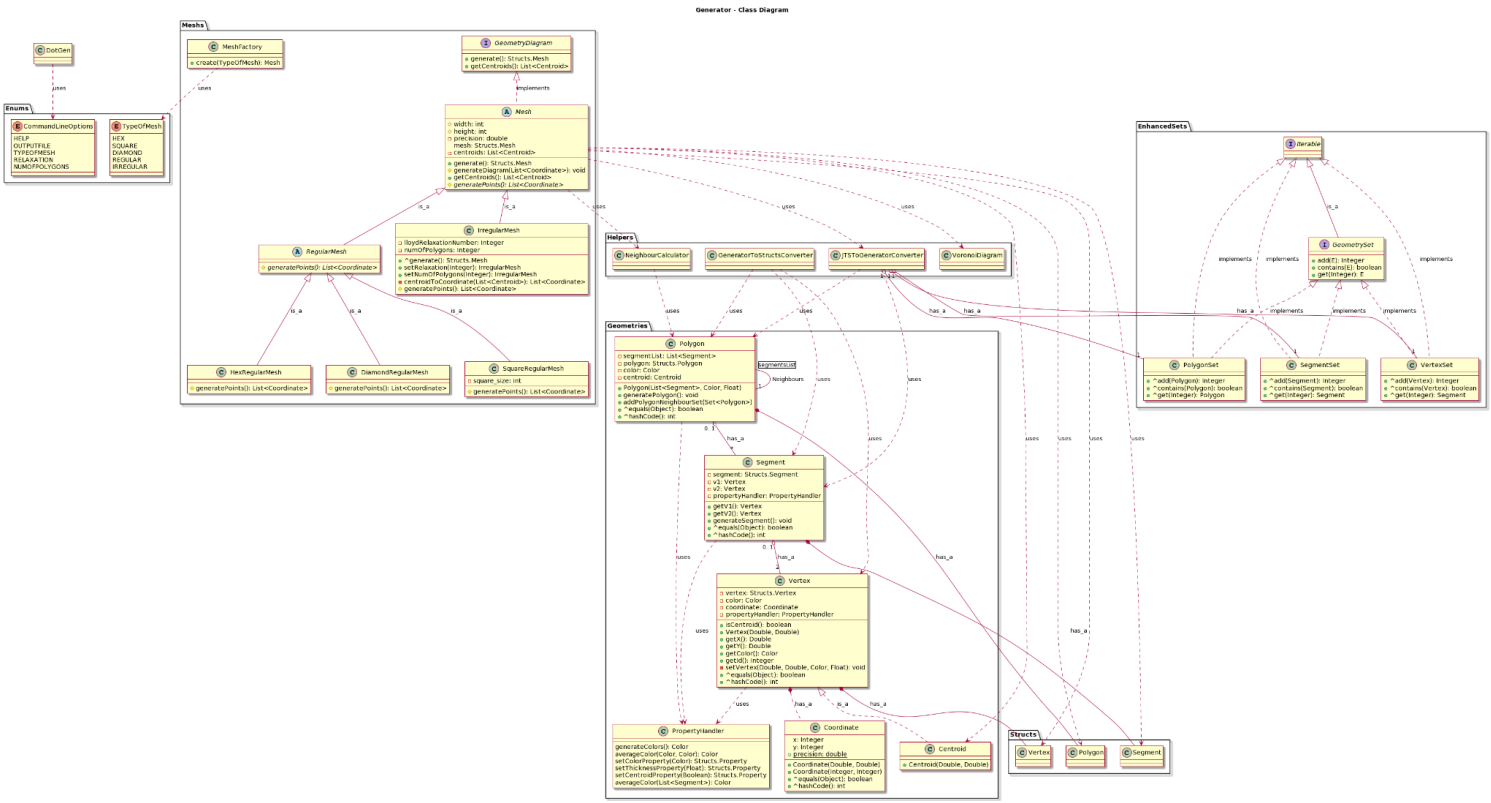
To what extent the testing activity helps the release of features in your product?

Testing is crucial for ensuring the quality of the features that we develop and release for our product. Currently, we rely on running the code utilizing different input scenarios to check if it generates the expected output, but we plan to incorporate formal testing frameworks like JUnit in the future. By testing our code, we can catch bugs and issues early, minimizing the need for future refactoring when new features are implemented. Additionally, testing helps us identify specific lines of code that are causing the program to break, allowing us to focus our efforts on tackling the root cause of the problem. By utilizing proper testing, we can ensure that our code is robust and of high quality, meeting the needs and expectations for the user. Furthermore, through unit testing we can ensure our new features don't interfere with the already implemented features, which is key for larger scale collaborative projects. If everyone adds tests for new features as we go, we can make sure code one doesn't fully understand is still working and if it breaks, it's easier to debug as long as the tests follow: context, action, assertion.

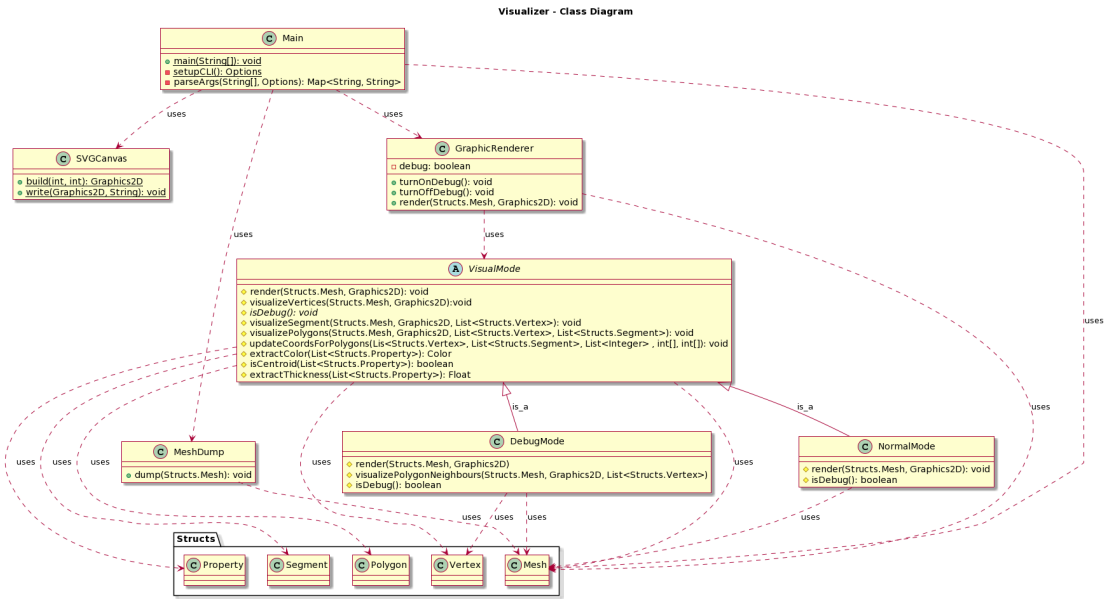
Part 3

Part 3 Class Diagrams and Explanation

Generator:



Visualizer:



*Note: Within our repository, under “umlDiagrams” you can view the images there if it is too small on the report

We believe that this updated and refactored product is ultimately the correct design for this project based on the class diagrams shown above. The updates that we implemented into our code better utilizes the SOLID principles learned in lecture. As the class diagrams show, we've decomposed our modules even further to ensure single-responsibility amongst each sub-project and as a result in addition to abstraction, the open/closed principle is guaranteed to be prevalent.

The generator is now segmented into separate packages which increases readability for the program. Enhanced sets are now introduced to ensure that the minimal mesh is achieved. With these sets, no duplicate vertices, segments or polygons are guaranteed to be within all mesh types. Each of the enhanced sets now implement the iterable interface which allows for easy, immutable access to its contents.

The visualizer has also been further decomposed, implementing an abstract class called VisualMode with child classes NormalMode and DebugMode. The DebugMode class acts as a decorator class, adding in the visualization of each polygon neighbour. Should new visualization modes be implemented within the future, the modular decomposition of the visualizer sub-project allows for easy addition without the need for heavy modification.

Our team stepped back and focused on implementing more GRASP patterns, especially *information specialist* and *creator*. Our MeshFactory class acts as the specialist to switch between the different types of mesh that can be generated and knows which classes to access in a particular situation. In this specific case, this is an implementation of the factory pattern, which takes in user input and returns the correct mesh object. These are still all exposed as a Mesh type, but it's runtime type holds the ability to generate the proper mesh. Each of our Mesh classes act as a creator and expand upon the now abstracted parent *Mesh* class, being responsible for creating instances of composed points that will generate based on the type of shape wanted.

We tried to favour the use of composition rather than inheritance in certain cases, such as with geometries, however when we did come to use inheritance, we made sure to uphold Liskov's substitution principles (anything the parent class can do, the subclass can do). For instance, all of our Meshes can generate a mesh, however, they just all implement generating the starting set of points differently. Inheritance was favoured in this situation because each subtype is defined as an "is-a" relationship with the regular mesh.

Furthermore, we used realization in order to keep our code business-oriented with high cohesion. The two specific cases in which this was important were for EnhancedSets and GeometryDiagram (Mesh). For the former, it was necessary to specify the general functions needed to implement a set (such as contains, add…). For the latter, it was necessary to make sure our Mesh class implemented functions that define it as a mesh (ex. it needs to be able to generate a mesh).

In conclusion, this class diagram displays the appropriate steps that our team has taken to achieve a product that follows the necessary software development principles learned in class.

Though it is impossible to find the perfect solution to a problem, our team believes that we have found the “least-worst” solution given all of the tools provided to us.

Did you have to modify your visualizer for this part?
During the process of completing step 3, the implementation of the visualizers business logic was not modified. However, we did end up refactoring and cleaning up the visualizer sub-project to ensure that all proper object-oriented design principles were followed particularly with the different visualization modes. We ended up creating an abstract class “VisualMode” that contained generalized behaviours that the two visual modes (Normal and Debug) would later generalize. This abstract class supports the Open/Closed principle, as we can easily add a new class that extends the abstract class if we want to support a new visualization mode.

How did you encapsulate the complexity of geometrical computations in your design?
We encapsulated the complexity of geometrical computations in our design by segregating them into different classes / abstract classes while attempting to create high cohesion and reduce coupling using modular decomposition. Each class has its own single responsibility that it needs to complete, and despite being somewhat more complicated, the program is able to work in a more modular way and still allows for extension while preventing the need for modification.

If one has to generate another kind of mesh (e.g., triangular tessellation instead of square based), how would your design support it?
Implementing a new kind of mesh, such as triangular tessellation, would be a very simple task to do due to the object-oriented design of our code and our program adhering towards SOLID principles. You would first need to create a new class that would extend the regular mesh abstract class. Once created, the generatePoints() method would be implemented and generate the required coordinate points according to the desired shape of mesh that is needed. As a final step, the new mesh would require the addition of a new enum value to support the calling of the mesh via command-line arguments from the MeshFactory switch statement. This easy implementation demonstrates that our code follows the *O* in SOLID principles, being open for extension and closed for modification.

Where is your debt? How to repay it?
Though there is much less technical debt within our code than before, some debt does still exist. Rather than in the business logic sense, a majority of our debt comes from the readability of code. We plan to repay our debt by refactoring any code that may be duplicated or redundant, as well as add some comments to help segment each block of code.

How does your code respect SOLID principles?
As stated in previous questions, our code respects SOLID principles in a variety of ways. The single-responsibility principle is displayed throughout the modular decomposition that we’ve developed. Each class has its own responsibility to fulfill and uses cohesion with other classes to complete a bigger goal. The open/closed principle is shown through the implementation of new mesh types, such as the diamond and hex tessellation. Previous code did not need to be modified in order to add these new patterns, which greatly increased our development speed. Lastly, Liskov’s substitution principle interface segregation were both respected by our code as seen mainly through Mesh and Enhanced set. Dependency injection was also implemented as

we relied on our interfaces in the rest of the code (ex. returning a Mesh in MeshFatory). We also favoured composition over inheritance where possible and modularized our design accordingly.

Conclusion

What went well in the development? What went wrong?
The development within this project was quite a roller coaster to say the least. While there were many things that went well, there were also plenty of things that went wrong. First of all, the teamwork and collaboration that we had enabled us to complete features in a timely manner. Not only did this improve quality as we did not need to rush production, but it also gave us time to make modifications within the program if necessary. Our goal was to start early, learn early and fail early to prevent unnecessary revisions in the future. However while this was our goal, a lot of refactoring and cleaning up of code needed to be done after features were implemented. A major part of our design process was to get an initial feature working correctly first and then clean it up. This led to a lot of refactoring as the code would be very messy and would often be in one class or even within one method. This refactoring slowed down our development time and often created frustration during development when things weren't going according to plan. In the future, we should apply the BoyScout or Broken Glass principles in order to make sure the codebase remains tidy throughout development. In addition, some features took longer than expected to complete and may have been due to limitations within our code design. An example of this occurring was during the implementation of polygon neighbours within the irregular meshes utilizing Delaunay's Triangulation. Despite the frustration that our team endured, these mishaps are a great learning experience for the team. They allow us to enhance and evolve our software development skills so that we may not make these mistakes in the future.

Elaborate on using a version control system as collaborative support, instead of screen sharing or exchanging files through Discord. What does it bring to your development from an engineering point of view?
From an engineering point of view, using a version control system, such as GitHub, as collaborative support enables the ability for each member within the team to work on their own features and tasks in separate branches without requiring immediate review from other members. This method allows for working in parallelism, which as a result increases production speed and lowers time needed to complete features. It prevents any unwanted changes being implemented within the main branch of the program and it can be easy to revert back to a previous version of the program if a bug or error is found. In addition, using a version control system shows who wrote (therefore who is responsible for) each line of code, which can assist in tracing back errors effectively. Overall, using a version control system rather than screen sharing or exchanging files on Discord creates an easier workflow for the team, allowing efficient production and debugging of code while also ensuring a quality learning experience for each member.

How would you reorganize your team to approach the next assignment better?
To approach the next assignment better, our plan to reorganize our team involves a variety of steps. The first step is to ensure that throughout each feature being implemented, that we keep in mind the Broken Glass Principle and the Boy Scout Rule - keep things cleaner the way that

we found it. This will ensure that the program is not only always clean, but also give our team an easier time when debugging and fixing errors. The next step is to continue to identify each member's strengths and weaknesses so that we can assign new features accordingly. If we properly assign features to members who are best suited for them, we complete features efficiently and also prevent minor errors in code that can add up overtime. Finally as a last step, we will attempt to exhibit continuous unit and service testing to verify and validate the quality of our program.

Indicate how the workload was shared among the participants in terms of effort percentage. Your effort should be in the 30-40% bracket for a team of three.
The workload among the participants in terms of effort percentage was relatively even amongst the three team members. No member slacked off or let one person complete all of the work; rather, each member pulled their weight and assisted other members if one was lost within their assigned features. Our team continuously collaborated and pushed new ideas that may benefit our program to the table. We all exhibited even amounts of effort and steadily continued our way to the final product.

Bonus

We chose the Tessellation extension, and provided a few more ways to tessellate the 2D space:

In general, as explained in part 3 question 4, our design choices (adhereing to Open/Closed principle) made it easy to extend these new versions, rather than modifying our old code to support these new tessellations. It requires simply adding a new class for the tessellation and then adhering to the interface by generating the points to create the voronoi diagram around.

The 3 extra tessellations we chose were:

Diamond	Honeycomb	Hexagonal