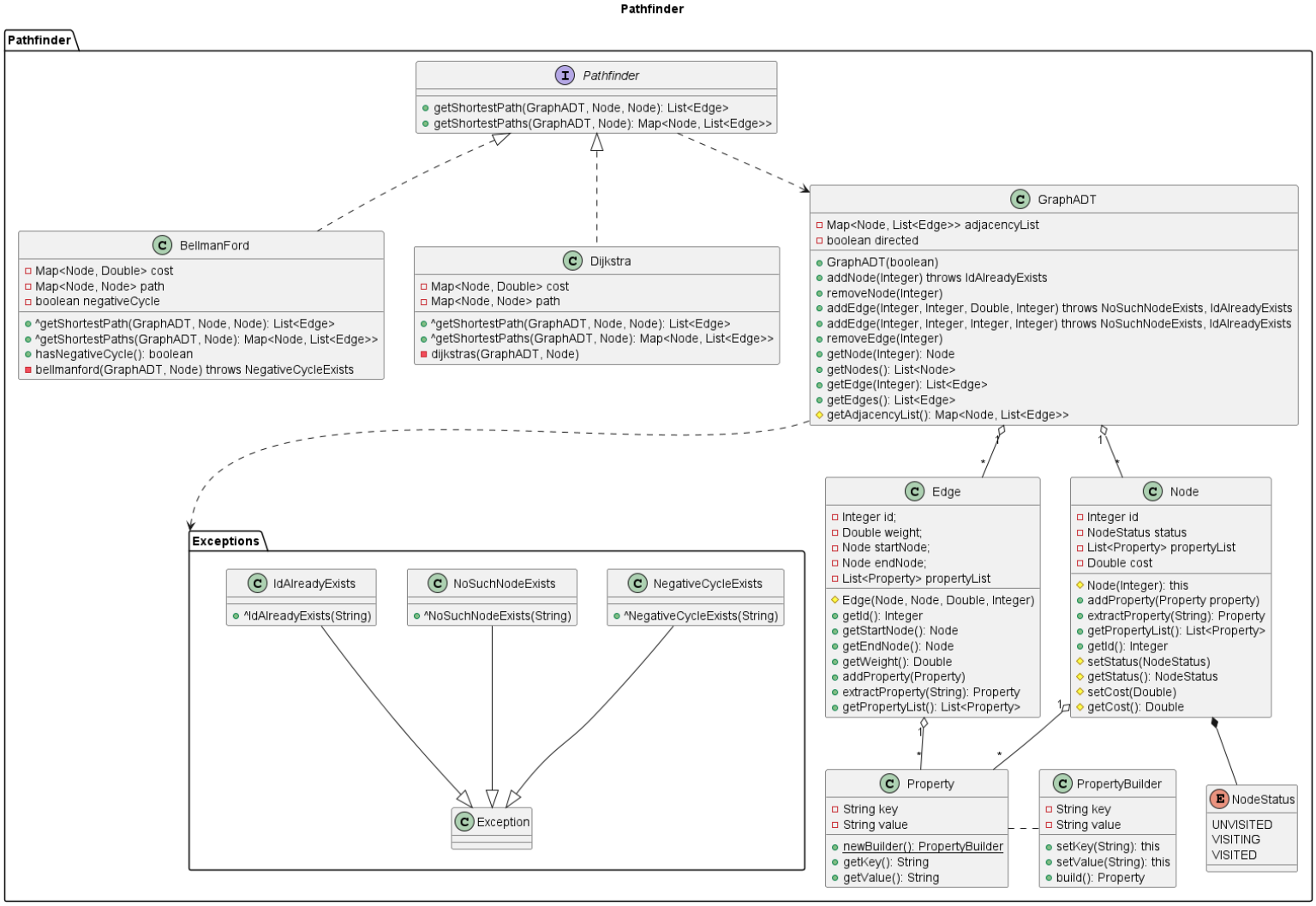
# Part 1

**Provide a class diagram of your pathfinder library**




**Explain why your code is SOLID, and locate your technical debt. If you have used any pattern, describe them and justify your decision.**

To begin, all classes exhibit the single responsibility principle. The graph, node and edge class all represent their discrete math counterpart. They simply house the information a graph needs, while the GraphADT exposes the necessary methods to interact with the graph. Pathfinder was extracted as a Visitor pattern as adding pathfinding algorithms, or other graph algorithms should not require to edit the definition of a Graph, Edge or Node. The Open/Closed principle is also respected through this as adding algorithms does not require to modify any code whatsoever; one must simply add a new class that implements Pathfinder, then the user can use that algorithm in their code to compute shortest paths. By programming to the Pathfinder interface, it can also take in any implementation of a graph and compute shortest paths, making it very loosely coupled to GraphADT. This anticipates the scalability of the Pathfinder library, since it allows an easy extraction of only Pathfinder algorithms to another sub-module. Furthermore, to ensure single responsibility principle, nodes and edges do not hold attributes for Island-specific properties, instead they can hold a list of Properties, which can be used in any way by the user. This allows the user to pack as many properties as they'd like into a node or edge.

Regarding technical debt, the Edge requires a weight to be associated with it, which can only be stored in the form of a Double. Trying to avoid premature optimization, this was the correct decision as the user required nothing more than using it for the Island generation process, however, this might arise as an issue later in the development of the library. When it becomes an issue, the Edge can adapt to accept any weight type that implements Comparable.

**Explain why your tests are "reasonable" (refer to CORRECT/BICEP if needed)**
When testing the GraphADT, I made sure to test all public functionality extensively. Cardinality was used to check if you can add and remove 1 or more of each element. Existence was used to check when creating an edge, if the inputted node ids actually exist within the graph. If not, a personalized error should be thrown. Conformance was used to check if user input (node ids and edge ids) conformed to the proper API. In other words, you cannot create two of the same element with the same id, else an error would be thrown. For pathfinding algorithms, Crosscheck was used against 2c03 slides to ensure both Bellman-Ford and Dijkstras algorithm were finding all the correct paths. The same scenario was used for both algorithms since the odds that both passed the tests incorrectly is close to impossible. Boundary was used for Bellman-Ford algorithm to check if it could detect negative edge cycles correctly. The example graph for that was also cross checked with a 2c03 graph example.

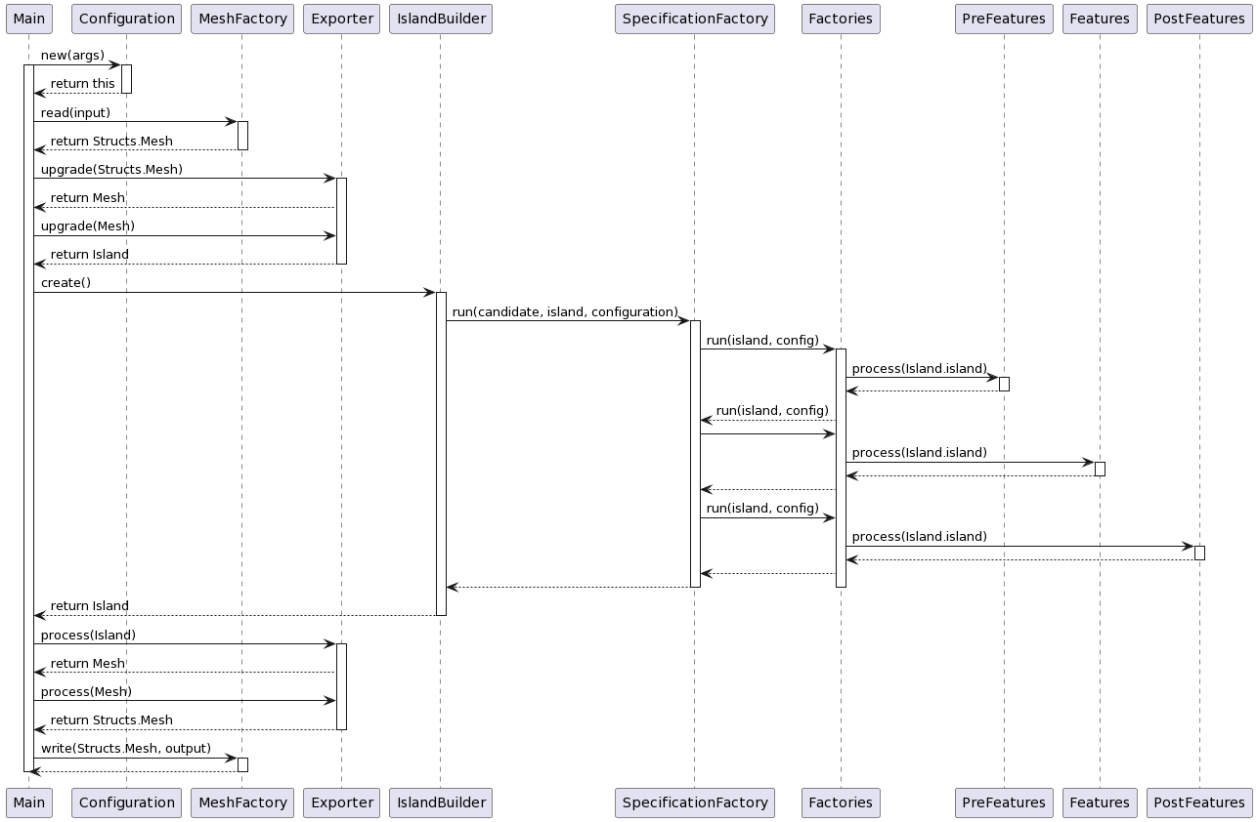
## Part 2

**Which design patterns have you used to support integrating your pathfinding library into your generator?**
I used the Adapter pattern to integrate the pathfinder into the island generator. This adapter class handled translating the centroids to nodes and neighborhood relationships to edges. It could then allow for the client class to use that graph to compute the shortest paths then translate the information back to island representation via the ids. Furthermore, I used dependency injection to pass the Network into the CityGenerator. This is good as it is the CityFactory's job to instantiate the right Network, then pass it to CityGenerator. CityGenerator doesn't need to know about the internal information of how it works, just that it must call the Network to generate the roads between cities.

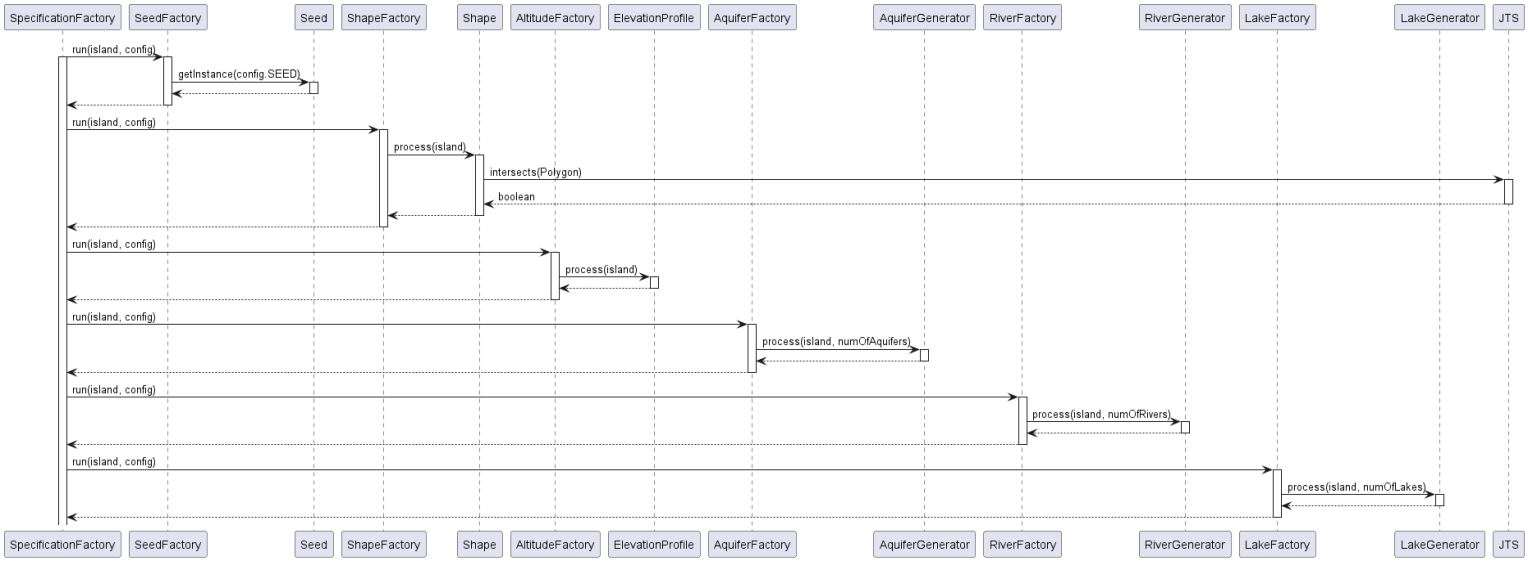
**What about performances? Do you envision any scalability issues in your road network generation?**
I used an adjacency list for the graph representation making Dijkstra's algorithm (O(NlogN + E)) for computing the shortest paths a good choice for performance and scalability. Adjacency lists are efficient for sparse graphs, which is what I have in the island since each centroid (node) only has around 5 neighbors (edges). Even though Dijkstra's is reasonable in most cases, there could still be scalability issues if the input data size grows significantly. This could occur if the user begins asking to expand from an island to a larger environment, such as a country, thus requiring more and more polygons.

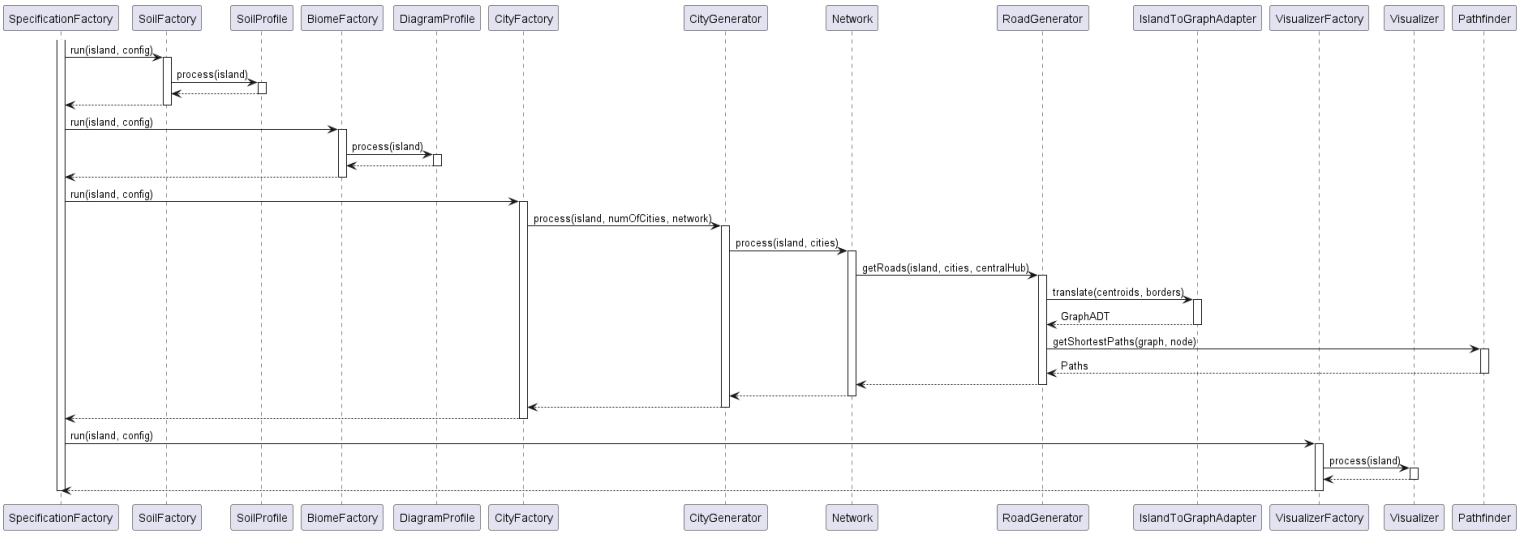
To address these potential scalability issues, I could consider using the A* algorithm, which is the one Google Maps uses, or optimizing my Dijkstra's. I could also try using parallelization, distributed computing, or using heuristics, such as only storing necessary edges to reduce the size of the input data.

**Draw a high-level sequence diagram explaining how your generation process works from a coarse-grained point of view. The point is to document the whole process, not only the urbanism part.**




*Prefeatures: Seed. Features: Shape, Altitude, Soil, Aquifers, Rivers, Lakes, Biomes. PostFeatures: Visualizer.* The Factory part of the generation is expanded on in the next two diagrams.






**Locate your technical debt in the generator. How can one extend it by adding a new generation mechanism in the generator? For example, if one plans to introduce resource production mechanisms (e.g., forests produce wood, lakes produce fish, ...), how would it be supported by your project?**

In the Specification, all the features are ordered based on PreFeature, Feature and PostFeature, however, the features within these subcategories are only ordered based on their fields within the Configuration class. This is an issue as some features that depend on others could be generated before them. For example, if a new developer comes in and moves the biome field before soil or elevation, there will be issues as biome depends on both elevation and moisture being generated. I have documented this technical debt within island.md, however, if a developer doesn't read this, they might fall into the mistake of adding a feature in the wrong order. Apart from that, in order to introduce a new feature is quite simple. The developer must create a new field for it in the Configuration class, add the binding in the Specification factory, then create a new factory to handle how to instantiate and run the generator for the Resource production mechanism. All the rest of the code (the code that actually does the resource generation) can be isolated into its own package, thus will not interact with any of the rest of the subsystem. In this way, the code is open for extension, but close to modification and loosely coupled to the rest of the system.

**Bonus**

The hardest part of the bonus was analyzing how I wanted to implement it. This was documented in Island.md. Since the code respected the SOLID principles, notably, I coded to the interfaces and used dependency injection by passing in the network to the city generator, adding a new network was isolated to making the new networks class and then attributing the responsibility of which Network to instantiate to the factory.

## Self Reflection

**Backward: What process did you go through to produce this result?**

To produce this result, I went through a process of gathering requirements for the system (messaging/reading the Teams channel for things that needed clarification), designing the architecture and components, implementing the code, testing the system, and iterating on the implementation to improve its performance and functionality. This involved a significant amount

of time and effort, but ultimately resulted in a functional and effective island generation system with urbanism.

**Inward: What were your standards for this piece of work? Did you meet your standards?**
My standards for this piece of work were to complete the assignment with proper software architecture, assuming I will be returning to work on the codebase later in life. I also aimed to get the bonus complete as it is a good test for the extensibility of the codebase. I met my standards in both cases and continue to strive to improve on implementing GoF patterns to maximize the cohesion and readability of my code. I also aimed to successfully document all my work in a nice and visual manner, which was complete in all the markdown files within my codebase and the UML diagrams.

**Outward: What is the one thing you particularly want people to notice when they look at your work?**
When looking at my work, I would appreciate it if someone looked more deeply into the Specification/Configuration aspect of the project. It is very satisfying to see how well the Open/Closed principle was respected, and how it makes it easy to enhance the library with more features. It simply takes adding a few bindings and a Factory to make the feature, then the rest of the code to implement the feature is COMPLETELY isolated from the rest of the subsystem. It usually just takes in an island and some configuration specific to the feature. The bindings could even be extracted to a properties file. That level of responsibility separation and extensibility is very satisfying and it is something I haven't even seen in most of the codebase at my previous co-op. Thank you 2aa4 :)

**Forward: What lessons will you keep from this assignment in your professional practice?**
DOCUMENTATION. Documentation is a very important and key part to software development. Without documentation, the life of others and your future self will be detrimentally affected. Furthermore, coding an MVP and then incrementally improving on it is a good strategy to end up with a SOLID codebase. What I did throughout these assignments is similar to the agile methodology, where I would begin to work on the projects incrementally, allowing me to ask for clarification or feedback in the teams channel when needed. This is a very good habit as it is prominently used in the workforce. Also, I will carry forward all the SOLID principles and GoF patterns. These are useful, especially for segregation of responsibility. At the co-op I worked last summer, I ran into issues while trying to incorporate new features into the codebase where I would need to refactor a section of the codebase due to the lack of SOLID principles. However, back then, I did not know the different patterns so I was trying to refactor instinctively. Now knowing the patterns and principles, I can not only discern where the issues are, but I can apply strategies to fix them.