2AA4 Team 01 A3 Report

Was your Feature Plan adequate?

The feature plan that our team designed was an adequate plan that assisted us with the scheduling and delegation of required features. The document provided us with a basic structure that we could follow and keep track of our progress.

A factor that we should've implemented within our feature plan was flexible deadlines for the completion of each feature. The enforcement of such deadlines would have created better organization for our team as during some moments of development, the arrangement of which features to prioritize at that given moment was somewhat convoluted. In addition, the feature plan had some modifications compared to the initial submitted document. Some features were removed due to being no longer needed, such as the choosing of the island's minimum and maximum elevation, while other features were combined together.

The assignment of features for each person was also different than initially planned. This was due to members having available time, as well as working together on a feature when difficulties arose. Nevertheless, we ensured equal distribution of effort was achieved and the feature plan still was of great use to our team.

What were the challenges in this assignment?

Throughout the duration of this assignment, our team faced many difficulties and challenges. When first dealing with the creation of the assignment, we created a dependency between the generator and island sub-projects.

This however created some issues as sometimes when executing the program created a NullPointerException. Ultimately, we decided to fix this issue by extracting our MeshADT from the generator into a new sub-project rather than directly depending on the generator. This ultimately reduced the heavy coupling that would've occurred if the adt was not extracted.

In addition to extracting this ADT, we modified the MeshADT so that we could create our mesh to hold polygons, vertices and segments, rather than exporting straight to Structs. We understand that the code we refactored in this sub-project is not the best due to the limited time constraints, but we ultimately fixed the code to the extent that we can show that if we were to do a full refactor, our island sub-project would be ideal.

Our team additionally encountered several other difficulties during the assignment. One of the issues we faced was determining why Liskov's substitution principle wasn't working when it came to extending a tile from a polygon. When we wanted to hold a list of polygons, the list also should've been able to hold tiles as well, however that was not the case. Through many struggles of trying to fix this, a decision was made to follow a design principle to favour composition over inheritance and have a tile hold the polygon instead.

Another challenge we faced was determining how to respect the open-closed principle with each feature. We had to carefully consider each feature and ensure that we could add new features without modifying the

existing code. Starting a feature and building it in the correct manner also posed a challenge, as we had to consider various factors such as complexity, maintainability, and scalability.

Testing also proved to be a challenge. We had to make sure our code was not over-encapsulated so that we could test it effectively. Throughout the design we were working with a full mesh and had to test the business logic of the program, which was a challenge in itself. However, we were able to successfully test the necessary features to ensure program correctness.

Overall, though these challenges were stressful and hard to manage at times, we were able to overcome these challenges to design the final product.

How does your code respect the SOLID principles?

Our team wanted to utilize the SOLID principles in a variety of ways within our program. Our code respects these principles in the following ways:

Single responsibility is implemented by ensuring that each of our classes has its own delegated task, rather than a multitude of them combined. An example of this is within our shape feature package, where we have a class for each shape that determines its own land boundaries. Within that package exists a shape generator that has the sole task of generating the land tiles based on its specified boundaries. Each of our features have a similar layout to this for the most part; therefore, this ensures that our program respects the single responsibility principle.

Continuing with the SOLID principles, we also ensured that our code respects the Open-Closed principle by designing our system to be extensible without having to modify existing code. For example, the addition of new Whittaker diagrams does not require modification of existing classes. If a new implementation of a Whittaker diagram was introduced, all that needs to be done is to extend the abstract class *WhittakerUtil* and implement the following abstract methods. This also goes similarly with the rest of our features. Though we attempted to adhere closely to the open-closed principle, there is one instance where this is violated in our code. Each biome was decided to be an enum rather than a class as we determined that holding a new class for each biome would be too complex, as biomes only hold a colour. This means that in order to introduce a new biome, modification to the enum would be required. We made a conscious decision to consider the services that a Biome provides. Our final decision was that all biomes offered the exact same service (associating with a specific colour), and a polymorphic approach was deemed to be unnecessary. Although this feature violates the open-closed principle to some extent, we made a conscious decision to prioritize tradeoffs for our design. Furthermore, in our Specifications, we added a binding hashmap that binds a user command with its respective class. Although you are technically adding to the code, it is considered to be closed for modification as we could extract this bit of code into a properties file and it would serve the same purpose.

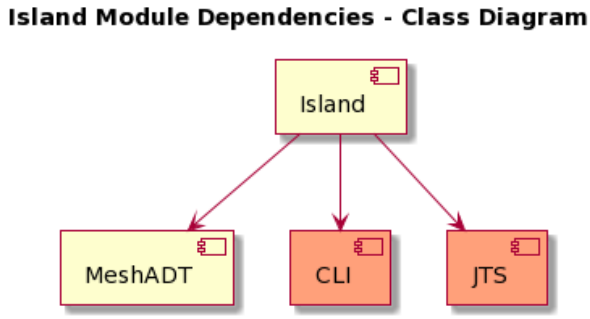
Liskov's substitution principle is also respected in our program. An example of this in effect is within our *“Water”* features. A *LandWaterGenerator* abstract class (which extends the *BodyOfWater* interface) is implemented for each body of water (lake and aquifer). Therefore a lake can technically be substituted for

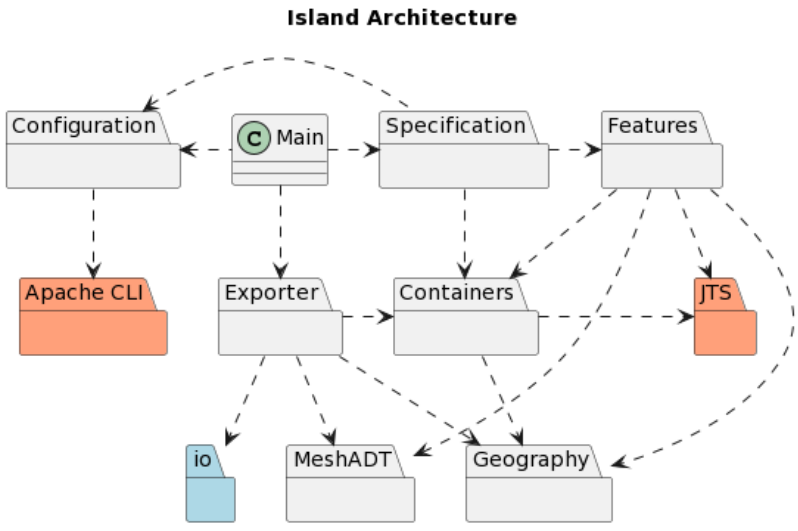
an aquifer, and the program will still be able to behave in a correct manner. Like the previous principles, this holds for the majority of our features.

The next SOLID principle that we've respected is the interface segregation principle. Throughout the program, multiple interfaces are implemented for each unique business logic. Shapes, elevation, water, whittaker diagrams all have their own interface. This ensures that any class does not implement a method that it does not use. We also programmed to interfaces in our Specifications as all subclasses of a feature adhered to its respective interface. This helped us keep our code closed for modification as no code was needed to be changed, you just needed to add the user command with its class in the bindings map, then we would pick the correct one and call the interfaces operation.

Lastly, our team utilized dependency injection by favouring composition over inheritance. This is shown throughout our tile, border and vertex decorator classes. Rather than having each of these classes extend their respective mesh geometry and implementing more methods, our team ultimately decided for each island geometry to hold these mesh geometries since an island is separated from a mesh and the only thing tying them back together is our exporter. This ensures that if the structure of these mesh geometries are modified, minimal disturbance occurs to the island geometries.

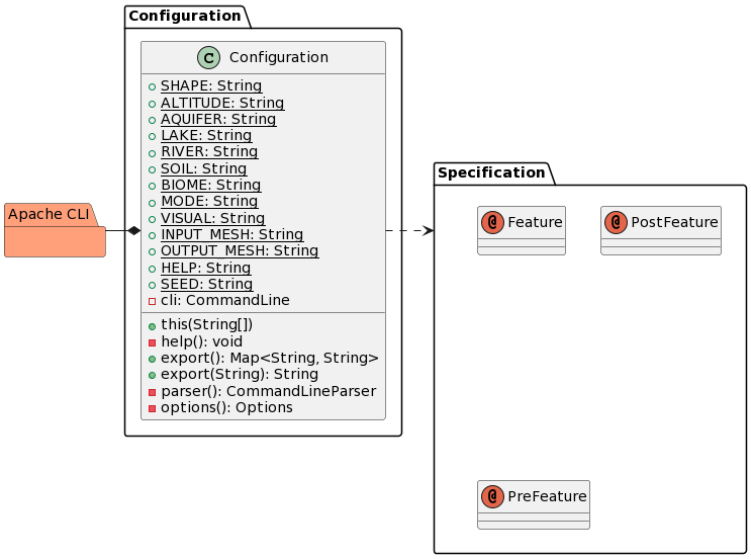
In order to visually display our code respecting these SOLID principles, class diagrams have been provided. The full resolution diagrams can be shown on our repository under umldiagrams/a3:



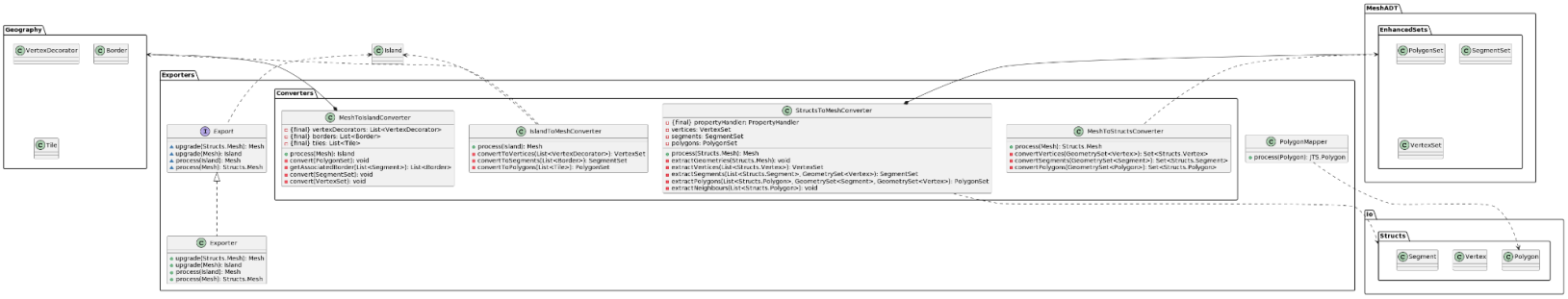


2AA4 Team 01 A3 Report

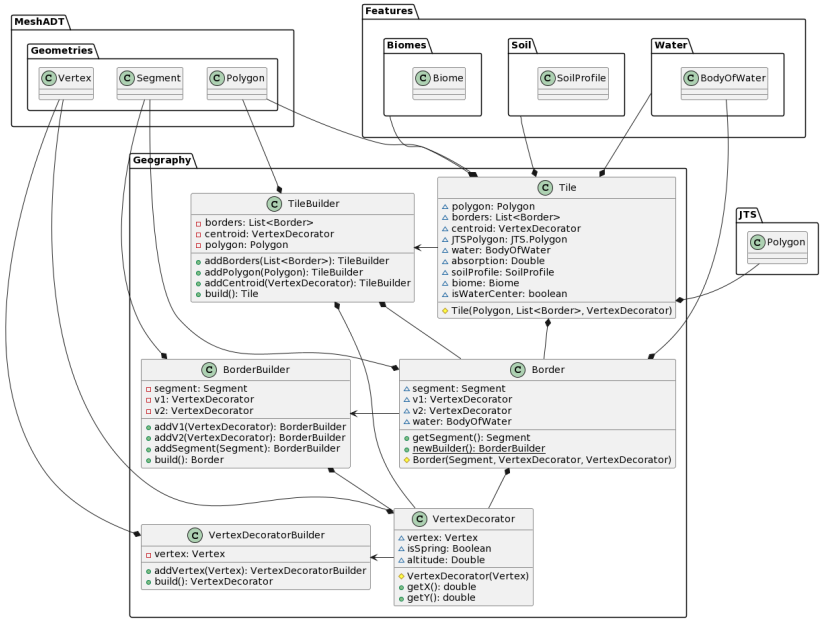
Island: Configuration



Island: Exporters:

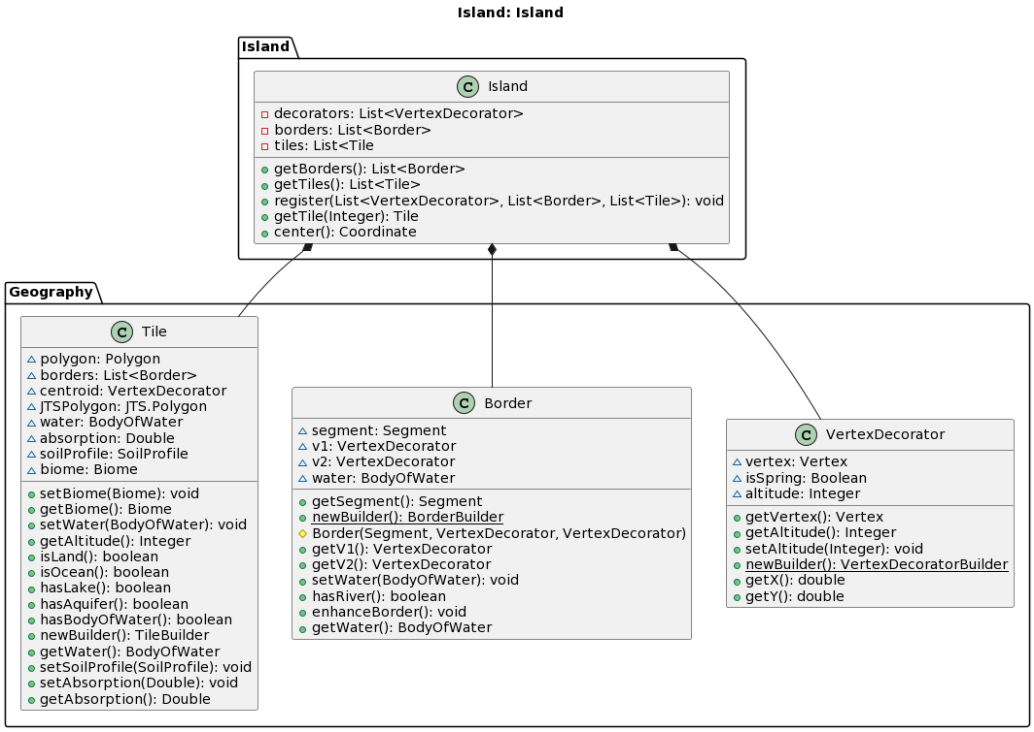


Island: Geography

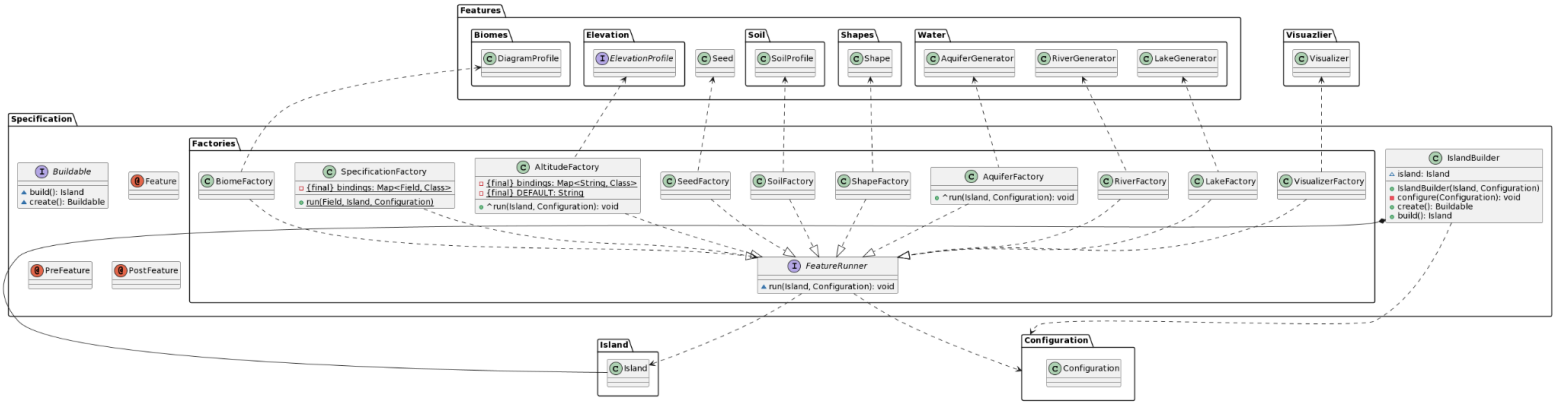


2AA4 Team 01 A3 Report

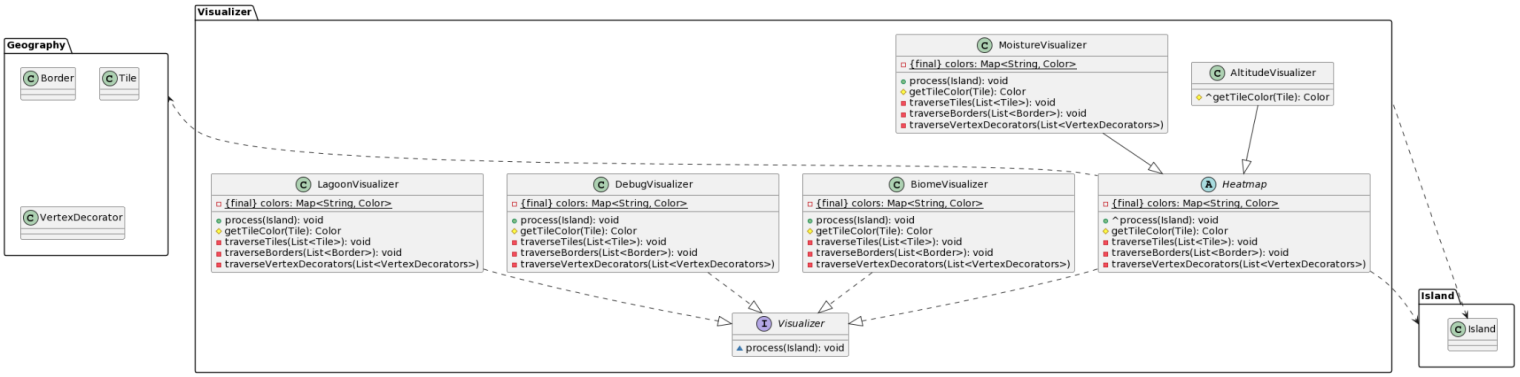
Island: Island



Island: Specification



Island: Visualizer



Which GRASP patterns have you used when attributing responsibilities?

Within our project, our team utilized the following GRASP patterns when attributing responsibilities: creator, information specialist, high cohesion and polymorphism.

For the creator pattern, this is shown throughout the generators we have in our project, such as the *ShapeGenerator* and *LandWaterGenerator*. These generators act as the creators for their designated features; they are the objects that are responsible for creating the other objects.

The information specialist within our project is the Island class. The island class is the specialist that has access to all of the information required for all of the tiles, borders and vertex decorators. From there, the responsibility of looking for any of these objects is fulfilled through the iteration of these collections.

We aimed to utilize the high cohesion pattern by ensuring that multiple classes ensure a single goal. This is shown throughout each feature such as lakes, rivers and aquifers, where rather than one class doing all of the work, the feature is achieved through modular decomposition. Though this may be more complicated in some aspects, this allows each feature to work in a more modular way and allows for maintainability of this code to be easier.

Finally, our team utilized the GRASP pattern of polymorphism. This example is shown throughout a majority of our features, but can be seen particularly in our shapes. Based on the different types of shapes, a different shape can be used interchangeably within our methods and still produce an output that can be used by our island.

We also attempted to minimize coupling within our program as much as possible. However, due to time constraints and the scope of the project, we made the tradeoff of coupling our tiles with the JTS library to utilize its functions such as the intersects method to implement some features, like shapes. While we understand that this design decision may not be the best, we believe it was worth it given the circumstances. But overall, our team aimed high to incorporate as many of the GRASP patterns as possible, which led to a suitable design for this project.

Which GoF design patterns have you used?

Our team implemented various types of GoF design patterns within the final design. We utilized the creational design patterns, factory, builder and singleton, the structural design patterns, adapter and decorator, and the behavioural design pattern, strategy.

In terms of the creational design patterns, factory, builder and singleton were utilized. The overall project had an IslandBuilder class, which generated the island based on its configuration. From then, each designated feature had its own builder, which generated the appropriate feature (shapes, elevation, whittaker diagrams, etc). Our program then consisted of a factory which would create new instances of the appropriate feature builder / object without needing to specify the exact class to create. The singleton pattern was used for our seed generation. We chose to use this pattern because we needed global access to ensure the reproducibility of a terrain given the specific seed. These creational design patterns that we

used ultimately assisted with us adhering to the open-closed principle as if a new feature was added the factory would not need to specifically indicate which class to run.

The structural design patterns of adapter and decorator were also implemented in a particular fashion. The decorator pattern can be seen within the vertex decorator, border, and tile classes, which takes in a vertex, segment or polygon and extends the behaviour of each object dynamically. Whether it's color, altitude, biome, these island geometries add on top of the current mesh geometries. The adapter in our program is the exporter class, which converts incompatible types to work with one another. This allows for easy implementation of required features.

The exporter class can convert:

Structs → Mesh → Island & Island → Mesh → Structs

Finally, the behaviour design pattern of strategy was used by each feature in our program. Each feature within the program has its subtypes segregated into different classes, for example with shapes there are circle, square, triangle, etc…. Despite this, each subtype can be interchanged without causing any interference or problems. This allows for polymorphism to occur and an easier adherence to the SOLID principles.

As a result of using some of the GoF design patterns, the design of our program is more easily maintainable and allows for better implementation of other important principles when it comes to software design.

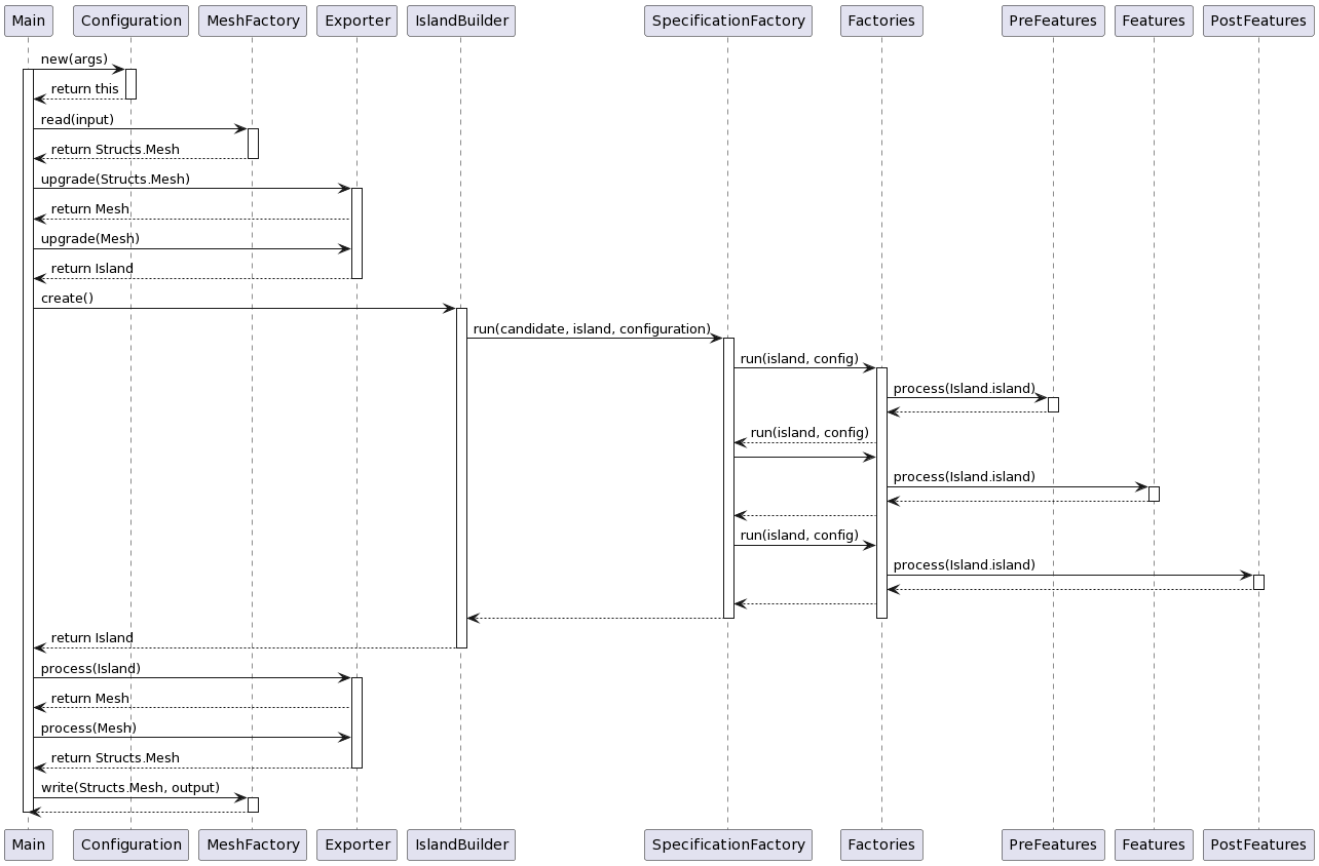
How did you design your test suite?

Our team designed the test suite we would be using for our program to focus on the necessary business logic of each feature. An example includes making sure that the correct biome is determined for a tile based on the specified Whittaker diagram, altitude and moisture levels.

Throughout each of our unit tests, we also focused our testing for the defining attributes of each feature such as correct tile definition, elevation profile and lake specification. In addition, determining which features and unit tests relied on each other was crucial to ensuring proper testing was achieved. We decided to mostly create one test suite per class we were testing. This ensured our tests were independent and cohesive.

At some moments, there were some challenges in designing this test suite and making sure the over-encapsulation or abstraction leaks did not occur in our program. Throughout designing the test suite, our team realized how important testing is to ensure the correctness of our program as well as a way to make debugging easier. In the future, we plan on implementing TDD which will guide us to make our code more testable as instead of testing written code, we write tests for what we want our code to do and then have to adhere to that as we code for green tests. This also helps with business logic as we decide from a user perspective how we want our code to behave before writing the code.

Island-Building Mechanism Sequence Diagram



Prefeatures: Seed. Features: Shape, Altitude, Soil, Aquifers, Rivers, Lakes, Biomes. PostFeatures: Visualizer. In our sequence diagram, Factories represents every factory we have for each feature. Each FeatureFactory then called the respective object that can process that feature based on the configuration (ex. if the user input -shape circle, then ShapeFactory would call Circle.process(island)). Therefore, for each FeatureFactory, it also calls Configuration to get the users input for its respective feature, but we omitted that from the Sequence Diagram as well for readability purposes. Also each feature is called sequentially but for readability, we listed them in the caption.

Bonus

We found incorporating the heatmaps a very easy task as we had visualizer use the Visitor pattern, which means each new heatmap we incorporated just needed to implement its own operation upon visiting each tile. It was also considered a feature, so adding it to the bindings list (which could easily become a properties list), is considered to be open for extension, closed for modification.

Appendix

Updated feature plan: Removed details are highlighted in red and modified details are highlighted in blue.

*ID's are continued from our current Backlog

Feature ID	Feature Name	Dependency	Who Implements It?
F19	Oval and Lagoon Shaped Island	N/A	Cyruss
F20	Scattered Shaped Island	N/A	Richard
F21	User can choose the shape of the island	F19 or F20	Hady, Richard, Cyruss
F22	User can choose the max and min altitude	N/A	~~Cyruss~~
F22	Support for ThreeCircle (Complex) Shaped Island	N/A	Hady
F23	Altimetric Profile for Volcano, Mountain	N/A	Richard, Hady
F24	Altimetric Profile for Arctic, Prairie	N/A	Hady
F25	User chooses the Altimetric Profile	F23 or F24	Cyruss, Hady, Richard
F26	Add randomly spawning lakes of random size in a range within the island	F19/F20	Cyruss, Hady, Richard
F27	Randomly spawn rivers that flow from high to low elevation, ending in a lake/ocean or lowest point (creating a lake)	F19/F20	Richard, Hady
F28	User can choose the max number of lakes	F26	Richard, Hady
F29	Rivers can merge and multiply	F27	Richard, Hady
F30	User can choose the number of rivers to spawn	F27	Hady
F31	Aquifers randomly generated based on user input	F19/F20	Cyruss
F32	Add soil types that get moisture based on their type and the humidity	F19/F20	Richard, Cyruss
F33	Add Dry Soil Type	F32	Cyruss, Hady

F34	Add Moist Soil Type	F32	Cyruss, Hady
F35	User can choose the absorption profile	F33/F34	Richard, Cyruss
F36	Add a American Whittaker Diagram	N/A	Hady, Richard, Cyruss
F37	Add a second Asian Whittaker Diagram	F23 or F24/F33 or F34	Cyruss, Richard
F38	User can choose which whittaker diagram to follow	F36/F37	Richard, Cyruss
F39	Regions are split into biomes based on the environment and whittaker diagrams	F23 or F24/F33 or F34/F36 or F37	Richard, Cyruss
F40	Randomly generate islands based off of a seed	N/A	Hady
F41	Return the seed to the user and allow them to pass in a seed (this should produce the same map every time)	F40	Cyruss Hady
F42	Add Altitude Heatmap Visualization	F23/F24	Hady
F43	Add Moisture Heatmap Visualization	F33/F34	Hady
F44	Add Debug Visualization (show spring sources and aquifers)	F26/F27/F31	Hady
F45	User can choose visualization type	F42/F43/F44	Hady