Assignment Report

SFWRENG 2AA4 (2026W)

Assignment 1

Author(s): S^4

Shifa Zaman	zamans27@mcmaster.ca	zamans27
Syaan Merchant	merchs8@mcmaster.ca	merchs8
Sanika Surose	suroses@mcmaster.ca	suroses
Lakshmi Saranya Alamanda	alamandl@mcmaster.ca	alamandl

GitHub URL

`https://github.com/syaanmerchant/Settler-of-Catan_2aa4.git`

February 13, 2026

Original template credits: Samuel Zurowski and Charles Barone. Adapted for McMaster SFWRENG by Dr. Istvan David.

1 Executive Summary

This assignment focuses on the design and implementation of a simulator that imitates the behaviour and conditions of a real world game of Settlers of Catan. The objective is to apply software engineering principles to transition from conceptual design to program code, while sticking to a defined specification and rule set.

The simulator models a simplified version of the game, including a valid game board, four randomly acting agents, and turn based gameplay with the official rules. The simulation executes for a user-defined number of rounds or until a player reaches ten victory points, printing all agent actions and game state updates to the console in a specified format.

The development process follows a structured engineering workflow. First, the design is created in Papyrus, which highlights object oriented programming principles. This model is then developed into program code through model driven techniques, which allows for consistency between the design and implementation. Generative Ai tools were also used to create another variation of the program code and are then analyzed for their strengths, limitations, and applicability in large scale software engineering projects.

The final implementation satisfies the functional requirements of the simulator and includes a demonstrator program that showcases the key mechanisms and invariants of the system. The assignment concludes with a reflection on the engineering process, talking about the design process, teamwork, time management, and the distinction between programming and software engineering.

2 Requirements Traceability

Req ID	Status	Implemented in	Design considerations
R1.1	Implemented	`MapSetup.java`	The software first sets up a valid hard-wired map using the specified tile and node identification mechanism. The map configuration is separated into the `MapSetup` class to simplify testing and debugging and is reused consistently across simulations.
R1.2	Implemented	`Simulator.java`	The simulator supports four randomly acting agents operating on the map generated in R1.1. Agent behavior is encapsulated to allow independent random decision-making during gameplay.
R1.3	Implemented	`GameEngine.java`	The simulation enforces the rules defined in the rulebook, excluding explicitly removed features. Core gameplay logic is centralized to ensure rule consistency across all turns.
R1.4	Implemented	`Config.java`	The simulator runs for a user-defined number of rounds (up to 8192) or until an agent reaches 10 victory points. These parameters are loaded from a configuration file to allow flexible experimentation.
R1.5	Implemented	`Simulator.java`	Execution halts automatically when one of the termination conditions from R1.4 is met, ensuring controlled and predictable simulation endings.
R1.6	Implemented	`RuleValidator.java`	Game invariants are enforced through validation checks, including road connectivity, city replacement of settlements, and minimum distance constraints between players' structures.
R1.7	Implemented	`ConsoleLogger.java`	The simulator prints encoded agent actions to the console and reports victory points at the end of each round to provide transparent progress tracking.
R1.8	Implemented	`Agent.java`	Agents holding more than seven cards attempt to spend resources by prioritizing construction actions, encouraging active gameplay progression.
R1.9	Implemented	`Demonstrator.java`	A dedicated demonstrator class contains a `main` method that runs representative simulations. Extensive comments explain the showcased functionality and expected behavior.

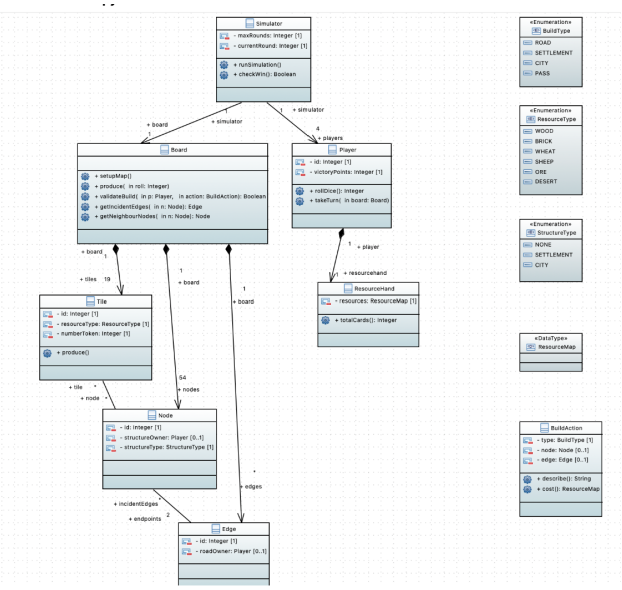

Figure 1: UML Diagram

3 Design and Domain Modeling

Question 1: Why this modeling stage was useful and its benefits compared to the PDF specification?

The specification in the assignment PDF describes the game in plain understandable language (turn flow, build rules, resource production, win conditions, etc.), but it does not explicitly define what objects exist, how they relate, or where each rule should be implemented. Creating a domain model converted the narrative into a shared blueprint of classes, relationships, and responsibilities. For example, the Board owns the map structure (Tiles/Nodes/Edges) and is where all map-dependent logic such as resource production and build validation would be implemented whereas the Simulator class coordinates rounds and termination conditions. Compared to relying on the PDF alone, the UML reduces ambiguity and provides a consistent structure for implementation, making it easier to code.

Question 2: How does the natural language description in the rulebook motivate conceptual modelling? If you believe it was beneficial, name three concrete benefits.

The rulebook is written in natural/plain language so it's easy for humans to read, but that same flexibility makes it easy to misinterpret terms like "valid build," "connected," or "adjacent" as they can be understood differently. That uncertainty is what motivates conceptual modelling instead as translating the narrative rules into a domain model makes it easier to explicitly decide what objects exist and where each rule belongs. While this step isn't strictly necessary for a small program, it becomes increasingly important as systems grow more complex because the number of interacting rules and edge cases increases.

In our project, modelling was beneficial in three concrete ways. Firstly, it clarified responsibility boundaries such as the Simulator controlling rounds, Board validating builds and producing resources and ResourceHand managing resources. This helped support parallel development when designing all the classes and would be implemented better as logic in our code. Secondly, it surfaced missing or conflicting assumptions early such as optional ownership on nodes/edges and representing pass actions with optional node/edge fields. This is way easier to fix in UML than after implemen-

tation into code where it can get complicated. Lastly, it improved testability by making invariants and checks explicit such as edges have exactly two endpoints, the map has fixed counts of tiles/nodes, and production/build rules depend on adjacency. This gave us a clearer basis for verifying correctness on the design.

Question 3: There are at least two aspects of the problem not modeled. What are they and how would they be modeled?

One of the aspects not modeled was the turn/phase control. We did not formally model detailed turn phases such as first rolling the dice, producing resources, deciding an action and then building/updating the board. This could be modelled by adding a playTurn(Player) operation in the Simulator or Player class. If needed, we could also create a TurnPhase enum that explicitly encodes the ordered steps of a turn. The second problem not modeled were certain rules to be seen as formal constraints. In the UML class diagrams, we represented the data needed for rules such as graph connectivity, ownership and structure types but we did not encode rules like the distance rule, road connectivity, and upgrade constraints as formal UML constraints. These could be expressed by introducing an explicit Validator or RuleEnforcer class that would keep track of these rules throughout the game.

Question 4: How OO mechanisms helped the design

OO design mapped game concepts to code in a maintainable way. Encapsulation keeps responsibilities local (resources in ResourceHand, map structure/rules in Board, simulation flow in Simulator, etc). Abstraction separates the overall world state (Board/Tile/Node/Edge) from the time progression that can be seen in Simulator rounds and player turns. Composition/associations clarifies containment and the lifecycle such as the Board containing tiles/nodes/edges and Player owning a resource hand. We used enums BuildType, ResourceType and StructureType as well as BuildAction to represent variation in actions/buildings while keeping the model simpler. If the system were expanded, polymorphism via Building subclasses would be a natural extension.

Question 5: How SOLID principles were used in the design

Single Responsibility (SRP): Each class has a focused job: Simulator manages rounds/end conditions, Board manages map data and rule checks, Player represents an agent and turn decisions, ResourceHand manages card counts, and BuildAction packages an action and cost. This improves readability and testing.

Open/Closed (OCP): The system can be extended with new action types or new validation rules without restructuring the entire model. For example, you could extend BuildType and handle new cases in action/rule logic or introduce new validator components.

Liskov Substitution (LSP): Our simplified model uses minimal inheritance, reducing LSP risk. If we later add Building subclasses (Road/Settlement/City), we would ensure all subclasses obey the same contracts such as consistent cost/placement behaviour.

Interface Segregation (ISP): We kept APIs narrow and purposeful. For instance, Board provides targeted query methods (incident edges, neighbour nodes) rather than exposing internal collections everywhere.

Dependency Inversion (DIP): The design can support DIP by separating "agent decision logic" from the simulator (e.g., an AgentPolicy interface), allowing us to swap random agents for smarter agents or a demonstrator without rewriting the whole Simulator class.

4 Translating Engineering Models to Program Code

Question 1: What are the translational semantics of this step? (What in your design model maps to what in your program code?)

In Task 2, after building our UML class diagrams in Papyrus, we used Papyrus' code generation to translate the model into Java. The output is mostly structural, meaning it creates the skeleton of the program (types, fields, and method headers), but not the full behaviour.

First, UML classes translate directly into Java class files with the same names. For example, classes like Board, Simulator, Player, Tile, Node, Edge, BuildAction, and ResourceHand each became their own .java file. Next, UML attributes become Java fields inside those classes. For instance, our BuildAction class in the UML produced fields such as type, node, and edge in Java. Similarly, the Node class produced fields like structureOwner and structureType. This shows that data we model in the UMLs are preserved fairly directly in the generated code. Then, the UML operations become Java method signatures, but Papyrus does not automatically generate the internal logic. Instead, it produces method stubs with empty bodies but the correct name, parameters, and return type. Examples include Simulator.runSimulation(), Board.validateBuild(Player, BuildAction), and Player.takeTurn(Board). For types like enums, the translation is very clean as UML enumerations become Java enums.

A major limitation appears when the UML model includes a type that the generator cannot represent well in Java. For example, we wanted ResourceMap to behave like a real Map<ResourceType, Integer>, but since we couldn't model that directly in Papyrus, we used a UML «DataType» called ResourceMap. Papyrus translated this into an empty placeholder class (public class ResourceMap) rather than a real map implementation. This preserves the type name so the rest of the generated code still compiles structurally, but it means the actual "map behaviour" must be written manually. Finally, associations and multiplicities in UML represent important relationship constraints (such as one board has many tiles), but they do not always turn into actual Java collections automatically. Whether you get a generated List/array field often depends on navigability and generator settings. Overall, the generated code is a consistent starting framework that matches our model's structure, and we build the real simulation logic on top of it in later tasks.

Question 2: What are the benefits of this step?

The biggest benefit of doing this step is speed and consistency. Papyrus instantly creates a complete set of files, class names, enums, fields, and method headers that match the design, which reduces boilerplate and prevents early design confusion that may exist between our group. It also acts like a design sanity check as once you inspect the generated code, gaps become obvious. For example, the ResourceMap which we intended to act as a separate data type to replicate a map, appears as empty in the generated code. Additionally, many stub methods further showcase where real logic must still be written in. Overall, code generation helps reduce the conceptual divide between UML and implementation by ensuring the code structure directly reflects the model, even though behaviour still needs to be implemented manually.

Questions 3: When would you use and when would you skip the modelling part and start writing program code immediately?

In general, modelling + code generation is most useful when the system has many interacting entities and rules, like a board-game simulator with a board, players, resources, actions, and rule constraints. In that kind of project, a UML model gives the team a shared blueprint for what objects exist, what they store, and how they relate, which makes it easier to divide work (for example, one person focuses on board validation, another on player decisions, another on the simulation loop) without everyone inventing their own structure. Code generation then helps by producing a consistent starting skeleton with classes, fields, and method signatures so the group can spend most of their time implementing behaviour rather than setting up boilerplate. This approach works best when the overall structure is fairly stable, even if the details of the methods will take time to implement.

I would skip modelling and start coding right away when requirements are changing quickly

or when the program is small enough that a diagram adds little value. I would also keep the model lighter if the code generator introduces too many placeholders or awkward translations, because that creates extra overhead (for example, our ResourceMap being an empty generated class instead of a real Java Map). In those situations, it can be more efficient to sketch a minimal design for communication and put more effort into writing, testing, and iterating on working code.

5 Using Generative AI

Note: Hyperparameters are not exposed in the ChatGPT web UI so default system was used. The following prompt used on ChatGPT (GPT-5.2 Thinking) in addition to attaching the UML diagram in section 3. "Generate Java code that matches this UML exactly (classes, fields, methods, enums, associations). Use package CatanAssignment1. If you need ResourceMap, implement it as Map<ResourceType, Integer>."

Question 1: What did your gen AI tool do well? What are its strengths?

The AI performed well in terms of translating the UML structure provided into consistent Java code. The AI generated all the classes/enums with proper names and the attributes and methods were mapped consistently. The code in general also looks like the diagram as for example, one of the more complicated classes, the Simulator class, has the maxRounds/currentRound attributes, an associated Board object, and a collection of players, just as described in the UML. The choices it made regarding how to represent the multiplicities in code were good as this is the type of code that is easy to get wrong by hand.

Additionally, the AI was better in filling in certain structure gaps that Papyrus had made when translating the UML. The ResourceMap that was attached in the image model provided was assigned to be a datatype as Papyrus did not have an option for Map types as parameters. The AI however, generated a proper ResourceMap class wrapping around Map<ResourceType, Integer> and gave it small helper methods (get/put/add/total) so other classes can operate on it. ChatGPT also provided some minimal placeholder logic that also compiles so it shows the flow through the program. This could be seen through the class Player, for example, which contains a simple implementation of rolling 2 6-sided dies.

Question 2: What mistakes did your gen AI tool make? What are its weaknesses?

The greatest weakness the AI tool had made was through guessing the behaviour. The logic that the AI created is reasonable but not necessarily correct in relation to the rulebook or the assignment specification. For example, BuildAction.cost() is hard-coding the typical costs for Catan, which could be incomplete, differently scoped, or not necessary at this stage of the game. In addition, functions such as Board.getIncidentEdges() / getNeighbourNodes() are probably just placeholders and will not be representative of the actual relationships between the nodes in the graph. Instead the methods would probably just return the first found element rather than the actual incident edges/neighbours. It also doesn't perfectly implement the UML multiplicities/constraints. Using List< > for things such as tiles, nodes, edges is sensible, but this does not guarantee exactly 19 tiles or 54 nodes unless extra checks are implemented. In summary, the AI is guessing the behaviour, which is not necessarily correct in relation to the rulebook or the assignment specification.

Question 3: How would you use gen AI in a large-scale software project? How would you balance its strengths and weaknesses?

In a large project, I would employ generative AI mostly for acceleration purposes. This would include using it for boilerplate code generation, scaffolding based on a model, generating consistent method stubs, implementing small helpers, and generating test skeletons or documentation drafts. The key idea is to view AI-generated code as a first draft that accelerates development, rather than

the ultimate final design. I would also employ it for suggesting alternative designs, but the design choice would be made deliberately by the team and not simply accepting the first draft. To balance both the strengths and weaknesses, I would employ some guardrails such as code reviews for AI-generated code, automated verification (formatting, static analysis, unit tests), and keep AI prompts very clear (such as "match the UML exactly and do not add extra fields/methods"). For behavior-heavy code, I would validate against specifications with tests and consider AI-generated rules as untrusted until proven correct. This would allow AI to accelerate development and consistency, while humans and tools enforce correctness, specifications, and maintainability.

Question 4: You run a company and you need to develop a software tool for your clients. Your software team suggests four potential strategies as shown in the table below. Which strategy would you use and why?

Given the fact that we start off with 1,000,000 users, lose 10 users/day until the tool is released (days to release = design days + programming days), get paid \$1000/user in the end and each strategy provides a churn rate for the fraction who leave, we can mathematically calculate the revenue generated for each strategy.

Cowboy Coding:

$0 + 20 = 20$ days

Users left:

$$1,000,000 - 10 \cdot 20 = 999,800$$

Paying fraction:

$$1 - 0.25 = 0.75$$

Revenue =

$$999,800 \cdot 0.75 \cdot 1000 = \$749,850,000$$

Vibe coding:

$0 + 10 = 10$ days

Users left:

$$1,000,000 - 10 \cdot 10 = 999,900$$

Paying fraction:

$$1 - 0.33 = 0.67$$

Revenue =

$$999,900 \cdot 0.67 \cdot 1000 = \$669,933,000$$

Conventional SE:

$20 + 20 = 40$ days

Users left:

$$1,000,000 - 10 \cdot 40 = 999,600$$

Paying fraction:

$$1 - 0.05 = 0.95$$

Revenue =

$$999,600 \cdot 0.95 \cdot 1000 = \$949,620,000$$

Model-driven SE:
$25 + 5 = 30$ days
Users left:

$$1,000,000 - 10 \text{å} 30 = 999,700$$

Paying fraction:

$$1 - 0.025 = 0.975$$

Revenue =

$$999,700 \cdot 0.975 \cdot 1000 = \$974,707,500$$

The best strategy in such a case would be model-driven software engineering, as it results in the highest expected revenue when taking into account both time to market and customer churn. In this case, only a few potential customers are lost each day (10/day), so finishing only 10-20 days earlier is not significant compared to the results of delivering a better quality product. This is also true because customer churn will remain at its lowest at 2.5%, meaning that many more customers will remain and generate revenue rather than demanding a refund, compared to losing a few customers by taking 30 days to complete the software. After calculating, you can also see that we will still have approximately 999,700 potential users left, resulting in a revenue of approximately \$974.7M, which is more than traditional SE (\$949.6M) but also significantly more than cowboy coding and vibe coding. Therefore, even though model-driven SE takes longer in terms of the design phase, it is a better balance between having a good structure, delivering a product quickly, and delivering a better quality product, which results in a higher revenue in general.

Question 5: Why did the Instructor ask you the questions in the previous point?

This question was asked because it makes you think like an engineer and not just a programmer. Programming generally has the mentality of whether you can build it, but an engineering way of thinking considers the best approach to do it under real-world constraints like quality, customer loss, risk, and business results. This question illustrates that different development methodologies are not just preferences, but result in quantifiable differences, and as a software engineer, one should be able to argue the approach on the basis of cost/benefit considerations like revenue impact and not just technical convenience.

6 Implementation

Table 1: Summary Table of Key Invariants Implemented (ED1-ED14)

ID	Decision	Why	Trade-off	Code Reference
ED1	Use a fixed beginners board topology with hardcoded node/edge/tile tables	Keeps map setup deterministic and avoids hex-coordinate generation complexity	Not reusable for randomized/custom maps without replacing tables	`src/main/Catan Assignment1/Board.java`
ED2	Store incidentEdges and neighbourNodes directly on each Node	Makes repeated rule checks (connectivity, distance) cheap and simple	More memory use and reliance on correct setup	`Node.java; Board.java`
ED3	Represent resources with EnumMap<ResourceType, Integer>	Type-safe and efficient for a fixed enum key set	Tied to predefined resource enum values	`ResourceMap.java`
ED4	Separate build flow into validateBuild(...) then executeBuild(...)	Prevents unsafe mutation and supports prechecking by AI	Some repeated branching logic across validate/execute paths	`Board.java`
ED5	Keep all build costs in one static source: BuildAction.costFor(...)	Prevents cost drift between affordability checks and execution	Global/static cost model (no player-specific modifiers)	`BuildAction.java`
ED6	Use BuildAction as the turn decision DTO (type + target node/edge)	Standardizes validation, execution, and logging interfaces	Runtime guards still needed for invalid type-target combinations	`BuildAction.java; Player.java`
ED7	AI chooses randomly from valid affordable actions, else PASS	Produces valid non-trivial behavior with low implementation complexity	Weak strategic quality	`Player.java`
ED8	Simplify >7-card behavior to "must attempt to build" (R5)	Matches assignment simplification while adding resource-spend pressure	Differs from official discard-on-7 rule behavior	`Player.java`
ED9	Simplify roll-7 handling: no production only; robber effects omitted	Keeps scope focused on build-rule invariants	Economy/game flow differs from full Catan	`Board.java; Simulator.java`
ED10	Update victory points immediately when state mutates	Avoids repeated board scans for score recomputation	Must avoid double-counting when extending features	`Board.java; Demonstrator.java`
ED11	Use deterministic starting placements + bootstrap resources	Guarantees legal and playable startup state for all players	Includes non-standard resource injection	`Demonstrator.java`
ED12	Add simulation stop guards: win condition or bounded rounds (max 8192)	Guarantees termination and protects against bad config values	Hard cap is practical but arbitrary	`Simulator.java; Config.java`
ED13	Parse turns config with default/fallback behavior	Keeps app runnable with missing or malformed config input	Invalid input is tolerated instead of hard-failing	`Config.java`
ED14	Clamp resource quantities to non-negative values in ResourceMap.put(...)	Prevents invalid negative inventory states	Can hide logic bugs that strict exceptions would expose	`ResourceMap.java`

Table 2: Summary Table of Key Invariants Implemented (R1-R6)

ID	Invariant Statement	Enforcement Point(s)	Description
R1	A new road must be placed on an empty edge connected to the same player's existing road or structure.	`Board.validateRoadBuild();Board.playerConnectedToNode()`	Illegal disconnected roads are rejected. Edge must exist and be empty. At least one endpoint must connect to the player's owned road or structure.
R2	A settlement can be built only on an empty node, connected to the player's network, with all adjacent nodes empty (distance rule).	`Board.validateSettlementBuild()`	No adjacent settlements/cities and no disconnected settlement placement. Node must be empty. Node must connect to player's network. Every immediate neighbor node must be empty.
R3	A city can be built only by upgrading that same player's settlement.	`Board.validateCityBuild();Board.executeBuild()`	City placement cannot create ownership conflicts or direct city-on-empty placement. Node must already contain player's settlement. Execution mutates only structureType to CITY and increments VP by 1.
R4	Dice roll 7 produces no resources (robber effects omitted by spec simplification).	`Board.produce(int);Simulator.runSimulation()`	No tile production happens when roll is 7. Early return in Board.produce(7). Simulator only calls production flow for non-7 rolls.
R5	If a player has more than 7 resource cards, the player must attempt a build action before passing.	`Player.takeTurn()`	Players with >7 cards do not immediately pass without first evaluating legal builds. PASS is returned only if no valid action exists.
R6	Simulation remains bounded: rounds are clamped and game terminates on win or configured round limit.	`Config.parseTurns();Simulator();Simulator.runSimulation();Simulator.checkWin()`	The simulation cannot run forever due to invalid config or absent winner. Config input clamped to [1, 8192] and defaults to 100 on parse issues. Main loop exits on win or max rounds.

Question 1: Did you get the domain models right at the first attempt, or did you iterate between modelling and implementation? If you did iterate, why and how? What does this say about your design and a human's modelling capability?

We did not change the UML after the implementation phase but went through a process of design iteration as we transitioned from modeling to executable code. The UML accurately captured the

structure and relationships of the domain — Board with its management of Tiles, Nodes, Edges; Players are Users of the Board; and Mutation of the Board's Original State (Player Mutating the Board) through the build action were the major abstractions through which the three parts of the software interact — remained constant throughout the course of the project. However, while we were designing for invariants and runtime behaviour, it became evident that conceptual clarity at the model level did not necessarily equate to behavioural correctness at the code level. For instance, one example of this would be that although the UML stated that the Nodes would have "Neighbours," to properly reflect the distance relationship (Rule-2), we needed to ensure that we accurately structured all of the true neighbouring relationships for all Nodes that physically exist on the topology of the Board. Thus, although the model was concerned with structural relationships, the actual implementation of the model reflected a requirement for an exact representation of the logical relationships between those structures as expressed through the design of a graph.

Table 3: Comparison Between UML Model and the Current Implementation

Class Aspect	UML Design	Implemented src/main..
ResourceMap	Defined as DataType	Coded as an enum in ResourceType as an integer
Board Topology	Abstract associations	Hardcoded arrays with 72 edges and 54 nodes
Nodes	Existance of associations	neighbour Nodes list provided
Player AI	taketurn() method only	Full game AI with validation loops
Build Validators	Generic validateBuild()	Three different build validation methods
Invariants	Not specified	Req 01-Req05 (R1-R5) stated

The distinction between levels of abstraction is significant. Informal, broad, high level conceptual descriptions define relationships and responsibilities in a human understandable way for modelling, whereas every relationship must have a definitive representation for implementation which allows for an unambiguous validation and execution of algorithmic steps associated with each relationship. The level of abstraction between the UML captures and what the implementation performs highlight the first reason why iteration was necessary. If the UML says that the nodes have incidicent edges but does not specificity how many edges there are per nodes (3) or how neighbours are computed (bidirectional setup) then, the implementation would not have sufficient information on how to do it. Another aspect to support why iteration was necessary is through comparing design-time vs runtime thinking. In the modelling phase, the thinking concepts ponder like, “A road connects two nodes”. However, during coding, more thoughts come about such as: Do edges store node references? Yes Do nodes store edge references? Yes (for O(1) lookup) What data structure? ArrayList<Edge>. It comes back to examining the strength of the UML. The current one does not specify the Beginners map topology and the number of total edges and nodes only appear in the implementation where the edge definitions (nodeA, nodeB pairs) are implementation artifacts that are also not disclosed. Finding the distinctions between the abstraction levels also solve any rule complexities early on. Since R2(the distance rule) was not fully specificied in the UML, the 3-neighbor check only emerged when coding validateSettlementBuild() and abstract "validateBuild" became concrete methods per build type.

Humans can construct very good representations of the structure of a system at a conceptual level; however, there are often behavioural edge cases (i.e., anomalous behaviour) that are not revealed until the system is executed. The iterative nature of modelling/implementation reflects the complementary nature of the modelling and implementation phases; modelling provides the structural guidance required to achieve behaviourally accurate execution while implementation provides feedback with respect to behavioural correctness needed to iteratively refine the model.

Question 2: What were some of the challenges in these iterations (technical or conceptual)? What tools or integrated tool chains would help you in those steps?

A few surfaced leveled technical challenges would be the board topology and defining the exact graph structure of a Catan board, ensuring settlements are 2+ intersections apart for the distance rule (r2), validating road builds connect to player's network for the road connectivity rule (R1) and generally tracking 6 resource types per player. Some solutions were less taxing to solve for example by adding an emun to track the type safety made resource management easier, or having bidirectional references going from edge to node was a good solution to the road connectivity problem. But issues like the board mapping needed more time and coding as that solution was to hardcoded edge definitions (72 pairs) and tile corners (19x6). However, the most challenging technical problem we faced about R2Enforcing the Distance Rule (Invariant R2), which specifies that all settlements must be located at least two intersections from each other, was arguably the most difficult aspect of the project. As a concept, enforcing R2 is easy; however, proving it on the board requires maintaining correct references between neighbouring nodes and ensuring that each neighbour is empty before allowing the placement of a new settlement. Even minor discrepancies in the neighbour reference list can lead to a silent enforcement failure of the distance rule; however, there will be no warning, and no indication at compile or run-time. This presents challenges when verifying compliance to R2 due to the aforementioned difficulties (i.e. 54 nodes with variable connectivity); therefore, the process of enforcing compliance to R2 requires very detailed evaluation of the surrounding node relationships, whereas enforcing compliance to R1 (i.e. road connectivity) merely requires confirming that the road connects to an existing player structure and is relatively easily understood.

Runtime debugging uncovered other conceptual areas which required additional thought. For example, the round Maximum Rounds configuration will sometimes not change based upon the configuration and the roads do not advance to the next round, which caused us to rethink the distribution of responsibility between Config, Simulator, Player and Board. It also emphasized the degree of coupling between State and Rule Enforcement in the Simulation System. Some useful tools for this iteration process are model-to-code synchronization tools, to detect drift between UML and implementation; graphic display tools, for analyzing board topology relationships; and property testing frameworks for the systematic validation of invariants across the nodes. There are many tools to consult for a smoother workflow in the future, in addition to Papyrus, using an Integrated Development Environment (IDE) or Continous Integration (CI) tools, and Kanban. Some more tools to note could be working by behaviour-driven development through tools like Cucumber, SpecFlow, or JBehave. It would help define games rules as executable specifications and provide pseudocode for concept analysis. For example if I wanted to builds roads that connect to my existing network as a player, the BBD workflow would provide me scenarios that encapsulate that feature such as a being connected roads (e.g. Given player P1 has a road at edge E1, When P1 builds a road at edge E2 and E2 connects to E1 via node N1 then the build is valid) or disconnecting roads with that same logic. That way, the rules become testable specifications. There are also tools like QuickCheck and jqwik for property-based testing. The benefit from this comes from verifying any invariants automatically, especially for situations like R1 and R2 since they are graph invariants.

Heres a sample of how its testing would look as code:

```
@Test
void settlementDistanceRuleAlwaysHolds() {
    forAll(boards -> {
        forAll(players -> {
            forAll(validSettlements(player, board) -> {
                assertThat(allNeighborsAreEmpty(node));
            });
        });
    });
}

@Test
void roadConnectivityAlwaysHolds() {
    forAll(validRoads(player, board) -> {
        assertThat(roadConnectsToPlayerNetwork(road, player));
    });
}
```

Question 3: Elaborate on the OO mechanisms and SOLID principles in your implementation. Are these any different from the ones used in the design? If they are: why?

The implementation uses various object-oriented methods including encapsulation through private fields that have bidirectional getters for data hiding and controlled mutation, enumerating Node types (ROAD, SETTLEMENT, CITY, PASS) to provide type-safe constants, creating data transfer objects where BuildAction has type, node and edge so that build decisions are encapsulated, and applying the strategy pattern to separate Player.takeTurn(Board) from GameState to allow for algorithm interoperability. The Object-Oriented Design principles, which include Encapsulation and separation of responsibilities, are evident in how the Simulator manages the flow of gameplay and Final Conditions of a game; the player's ability to make decisions and own resources; and the Board's management of the Game's topology, production rules, Validation of the Moves made by Players and execution of the Builds are all examples of how these principles can be applied. The BuildAction Value Object represents a single Build Attempt and thus separates the intentions behind the Build Attempt from the actual execution of the Build Attempt. This separation directly reflects the original UML Diagram design and reflects the same purpose-driven separation of responsibilities. Many component specifically adhere to a SOLID framework perspective. The Single Responsibility Principle prevails because each component of the game handles separate functionality; Board manages topology and validation while Player manages AI strategy and validation but does not rely on the game loop or Simulator to accomplish this; The Open-Closed Principle has added BuildType (extending the UML enums) without affecting the way that validation is done; The Liskov Substitution Principle , allows for the clean separation of BuildAction (data) with validation within the Player; and The Dependency Inversion Principle (DIP) means the Simulator relies on abstracted interfaces abstractions for both Board and Player (as opposed to concrete UML associations).
Furtherflow, the majority of classes conform to the Single Responsibility Principle from a SOLID framework standpoint. The Simulator manages which actions are taken within the game loop, the Player controls how strategies are executed and how resources are utilized, and the BuildAction class encapsulates the logic needed to define a BuildAction's cost and description. The Board class is the most notable example of a class that currently does not conform to this principle as it performs both static topology construction and dynamic rule checks on the board. It is acceptable to have

this level of fiunctionality in terms of the limitations of Task 4; however, a well-designed solution would separate the creation of the board's topology into a separate builder/map object.
The design intent has not been violated, but the implementation demonstrates that UML demonstrates static structure and that code must provide a definition of dynamic behavior over time. The initial UML design focused on static structure. Additionally, the initial UML design identified the datatype used for action separation as a strategy; however, the implementation of the design adds dynamic mechanisms, such as the board violating a principle of the Single Responsbility Principle and SOLID, to accommodate the behavior of the application at runtime within the limits of Task 4. Therefore, the differences between the UML design and implementation of the design are not contradictory; they provide different levels of abstraction. The UML model includes the relationships of the various objects in the system; whereas, the implementation will provide a mechanism for establishing those relationships once the system is functioning.

7 Reflection on the Engineering Process

Our team divided the workload by spitting the tasks according to the outline, while ensuring everyone contributed equally and has a part in each major component of the project. We break the assignment into modeling, implementation, testing, and documentation, and a group member was assigned ownership to one task. We all worked through our parts while helping each other on complicated tasks. We regularly reviewed progress together and redistributed work when necessary. This approach helped balance responsibilities and allowed every member to remain knowledgeable in each major part of the assignment.
Each team member contributed approximately 6-8 hours over the course of the assignment. Our time was distributed across modeling, coding, debugging, and report writing. While individual workloads may have varied depending on availability, we maintained transparency by discussing progress regularly and adjusting expectations when needed.
We primarily communicated through Instagram for discussions about meet ups or quick questions. We tried to check in with everyone during in person meetings, but due to members availability, this was done through the group chat as well. We had a separate doc that contained all written communication about the assignment which helped us maintain a record of design choices and reduced misunderstandings.
To handle time pressure, we prioritized core functionality early and established internal deadline to ensure we stay on task. We focused on incremental progress rather than attempting large last minute changes. When unexpected issues arose, we reassessed priorities and worked on delivering a stable implementation before refining optional improvements.
The most difficult organizational challenge was coordinating integration between independently developed components. Differences in assumptions about interfaces initially caused delays during merging. We learned that clear interfaces and integration plans earlier would make it easier on us closer to the deadline. In future projects, we would establish shared coding standard and interface contracts before implementation begins. This experience improved our understanding of collaborative software engineering and will influence how we plan and coordinate future team projects.

8 Roles and responsibilities

The team members contributed equally to the deliverable.

- TODO: **Shifa Zaman and Syaan Merchant** contributed to the conceptual design and the implementation of RQ1.1.
- TODO: **Shifa Zaman and Syaan Merchant** contributed to the conceptual design and the implementation of RQ1.2.
- TODO: **Shifa Zaman and Lakshmi Saranya Alamanda** contributed to the conceptual design and the implementation of RQ1.3.
- TODO: **Syaan Merchant and Sanika Surose** contributed to the conceptual design and the implementation of RQ1.4.
- TODO: **Syaan Merchant and Sanika Surose** contributed to the conceptual design and the implementation of RQ1.5.
- TODO: **Sanika Surose and Shifa Zaman** contributed to the conceptual design and the implementation of RQ1.6.
- TODO: **Sanika Surose and Lakshmi Saranya Alamanda** contributed to the conceptual design and the implementation of RQ1.7.
- TODO: **Lakshmi Saranya Alamanda and Syaan Merchant** contributed to the conceptual design and the implementation of RQ1.8.
- TODO: **Lakshmi Saranya Alamanda and Sanika Surose** contributed to the conceptual design and the implementation of RQ1.9.