Assignment Report

SFWRENG 2AA4 (2026W)

Assignment 1

Author(s)

Devan Sandhu	sandhd7@mcmaster.ca	sandhd7
Nadeem Mohamed	mohamn84@mcmaster.ca	mohamn84
Billy Wu	wu897@mcmaster.ca	wu897
Nikhil Ranjith	ranjin1@mcmaster.ca	ranjin1
Vivek Patel	patev124@mcmaster.ca	patev124

GitHub URL

https://github.com/sparksavior/2aa4-team1

January 28, 2026

1 Executive summary

Starting with broad conceptual sketches, Assignment 1 built a working simulation of Settlers of Catan by applying disciplined software methods. From those initial diagrams in UML, the approach moved stepwise into actual code, turning abstract ideas into functional parts. Object-oriented thinking shaped the structure, while SOLID guidelines helped keep components clear and independent. Game rules remained fixed throughout, serving as guardrails during development. Progress wasn't always linear - shifting back and forth between models and coding exposed mismatches that needed fixing. Despite these bumps, the outcome matched expectations, delivering a complete, running system aligned with requirements. Although the modelling phase helped sort out roles and connections among key components, putting things into practice revealed weaknesses in automatic code creation along with those in generative AI tools. What emerged was a working simulator that fulfills its intended purpose, showing how structured development methods outweigh unplanned programming efforts.

2 Requirements traceability

Req ID	Status	Implemented in	Design considerations
R1.1 Map setup and identification scheme	Implemented	Board.java Board.setup(), setupTiles(), setupIntersections(), setupPaths()	The board topology and tile configuration are hard-coded in Board.java. Tile identifiers follow the required structure (0 for center, 1–6 for inner ring, 7–18 for outer ring). However, intersection identifiers are assigned sequentially and do not follow the specified ring-based identification structure.
R1.2 Four randomly acting agents	Implemented	Simulator.java, Player.java Simulator.createPlayers(), Player.makeMove()	Simulator.java creates exactly four players. However, agent behaviour in Player.java is deterministic rather than random. Actions are selected using a fixed priority order rather than randomly choosing among all valid executable actions.
R1.3 Simulation follows rulebook (excluding specified features)	Implemented	Simulator.java, Board.java, Player.java Simulator.playRound(), Board.produceResources(), Player.handleDiceRoll7()	Resource production and core building mechanics are implemented. However, when a seven is rolled, players discard cards, whereas the assignment specification instructs that rolling a seven should simply skip resource production.
R1.4 Simulation runs for	Implemented	GameConfig.java, Simulator.java	The maximum number of rounds is read from a configuration file and constrained to the

configured rounds or until 10 victory points		GameConfig.fromFile(), Simulator.isFinished()	allowed bounds. The simulator terminates either when the configured round limit is reached or when a player achieves 10 victory points.
R1.5 Simulator halts upon reaching termination conditions	Implemented	Simulator.java Simulator.run()	The main simulation loop checks termination conditions at each iteration and halts execution immediately once either condition is satisfied.
R1.6 Key invariants respected	Implemented	Board.java, Player.java Board.canPlaceRoad(), Board.canPlaceSettlement(), Player.upgradeCity()	Road placement requires connection to existing infrastructure owned by the same player. Settlement placement enforces the distance rule by preventing adjacent occupied intersections. City upgrades replace existing settlements owned by the player. However, settlement placement does not explicitly require connection to the player's road network.
R1.7 Console output format and victory point summary	Implemented	Simulator.java Simulator.takeTurn(), Simulator.playRound()	Each action is printed in the required format and victory points are displayed at the end of each round.
R1.8 Players with more than seven cards attempt to build	Implemented	Player.java Player.makeMove()	When a player holds more than seven cards, Player.java attempts to build or upgrade. However, actions are selected deterministically rather than randomly from all possible executable actions as required.
R1.9 Demonstrator class with main method	Implemented	Demonstrator.java Demonstrator.main()	Demonstrator.java contains a public static main method that initializes the configuration and executes a simulation run.

3 Design and domain modeling

Useful came the modelling phase, turning loose guidelines into a clear framework for the system. Though gameplay appears in story form within the spec, entities along with their links and control appear directly in the UML version. Ambiguity dropped once roles gained definition ahead of coding. Where the PDF leaves edges blurry, the domain view pulls out classes, how they connect, parent-child patterns, and quantity rules plainly.

Though written in plain language, the rulebook pushes toward structured models by allowing different readings of its content. Because connections, possession, or swapping parts need clear boundaries, they become fixed design constraints. That shift helped pinpoint who handles what, revealed gaps - like lacking Path or Intersection elements - and made future changes easier by splitting data (Board) from logic (Simulator). Only when ambiguity fades do hidden pieces emerge.

Despite its scope, the model left out two elements. Not included were the shifting phases of turns and how dice function - these relate to actions, which fit more naturally within state or sequence illustrations. Structure, not timing, guides the class diagram’s design. Runtime movement plays no part here.

Using objects helped shape the structure via containment, type relationships, one thing built from another. Ownership patterns emerged through how pieces relate, whereas the playing surface brings together fixed parts. Rules like single duty per class guided separation of tasks, allowing growth without altering existing code. Thinking in models made translating ideas into working form more straightforward.

4 Translating engineering models to program code

One way to see it - structure shapes translation here. Classes in UML turn into Java classes, just like that. Fields emerge from attributes, shaped by context. Method signatures form where operations once stood. Where inheritance appears, extends follows, line by line. Enums move across untouched, fitting neatly into Java's enum framework. What you designed stays visible after generation. The core shape of the model holds, undistorted.

Partial translation of associations occurs. Arrays emerge from multiplicities like Tile[] or Intersection[], even though structure stays accurate. Despite correctness, Java benefits more from flexible collections - List, for example - offering stronger encapsulation. The preserved shape remains useful, yet demands adjustments before real-world application fits naturally.

Errors appeared throughout the process of creating output. Instead of forming correct Java constructors, the system produced them as void methods, violating basic rules of the language and needing human fixes afterward. Inside certain classes, repeated internal class definitions showed up - like more than one Board within a single container - even though these duplicates did not exist in the original design diagram and served no purpose. Primitive types from the modeling layer, including EInt and EString, made their way directly into the resulting source files, forcing developers to substitute standard equivalents like int or String by hand. What happened suggests translation from models to working code does not always preserve accurate programming structure, demanding careful inspection after conversion.

Though method signatures formed properly, their bodies stayed blank - this makes sense because behavior cannot come only from structural UML. Accurate inheritance translation occurred, keeping superclass–subclass links exactly as set in the original model.

One main advantage? Keeping the structure aligned across model and real-world build. When design matches code closely, through enforced sync, gaps in architecture become less likely. Still, evidence shows output from generators often needs reworking before deployment. For big systems - or those where failure risks are high - this method makes sense, due to its stability. On smaller efforts, though, skipping the extra step of modeling might save time overall.

5 Using Generative AI

Starting off, the GenAI tool produced Java classes that were neat and clearly laid out, matching key parts of the UML design. Instead of merging concepts, it kept each major component - Board, Simulator, Player, Tile - distinct and properly defined. What stood out was how inheritance appeared in logical form: Piece emerged as an abstract parent class, while Building and Road branched from it, along with Settlement following correct hierarchy rules. Unlike what often happens with Papyrus, which tends to nest classes unnecessarily, this version stayed streamlined. Clean structure came through without extra layers muddying the result.

Still, GenAI struggled to maintain structural meaning. Where UML showed two-way links, outputs often turned them into one-directional or left them out entirely. Take Tile and Intersection: their connection appeared incomplete, weakening the overall framework. Certain functions got altered too - how resources spread was made simpler than it should be, while the Simulator's sequence of operations missed key parts of the original plan. That suggests GenAI may copy surface layout but fails to consistently uphold underlying relationships.

When building big systems, GenAI can help draft initial code structures - yet design thinking still belongs to people. Because models make assumptions, every generated piece needs checking alongside specifications and intent. Without careful review, errors might slip into foundational layers. Oversight by developers, paired with tests and checks, keeps the system aligned and reliable.

One way to look at income outcomes shows that using models in software development works best. Even with greater initial expenses, the sharp drop in customer loss boosts earnings over time. What separates coding from engineering becomes clear here - coding aims for fast results, whereas engineering values accuracy, high standards, and lasting performance.

6 Implementation

It took several rounds of testing before the domain model fit what we needed, shifting back and forth between design tweaks and actual code. Moving from String to int for identifiers made a difference - handling positions by number streamlined how data was accessed. At first, connections like those between Tile and Intersection worked in just one direction, even though UML diagrams suggested they were mutual. While building features, it became clear that each intersection had to track nearby tiles so resource calculations ran faster. The handling of R1.8 rules started out incomplete; discards were defined

somewhere, yet never fully linked when dice results triggered them. During testing, this issue came to light, then addressed through restructuring to improve internal consistency. What stands out is how far apart the design ideas sit from actual code execution - showing clearly that refinement happens step by step, not all at once. Engineering moves forward because revisiting earlier work isn't an exception - it's built into the process.

Starting mid-sentence, clarity faded when syncing two-way connections without repeating elements. Shifting focus, speed gains - like tagging tiles with serial IDs - only surfaced after models were built. From a design angle, rigid adherence to SOLID principles clashed with diagram readability. Ending here: splitting every duty into separate types meant too many parts, pushing the project past reasonable limits.

Core object-oriented features stay intact from plan to product: inheritance structures the chain from Piece up through Building to Settlement or City, behavior varies via Building-typed pointers, while parts form wholes inside Board. Though clean separation of duties and reliance on abstractions slip here and there, those gaps come from conscious choices - balance matters more than textbook ideals. Work during build showed something clear early on: progress lives in back-and-forth motion, shaping ideas then grounding them, again and again, never just copying diagrams into software.

7 Reflection on the engineering process

Five people made up our group. We put in about seven hours' time. Work got split so everyone took one main job or was building the final report. Although we all had one major task we all worked together and significantly contributed to each other's parts. One tackled setting up the repo, managing tasks with Kanban boards, tagging releases, plus handling docs structure. Someone else worked on UML diagrams, refining them step by step. Another dove into Papyrus - generating code and studying how it translated. A fourth explored GenAI outputs, noting patterns found along the way. The fifth built Task 4's pieces: maps, config reading, rules that stay fixed, agent behaviors, logs, and added notes to the Demonstrator class. Sections came from a common outline that everyone owned their part with proof tracked as we went.

Most of our talks happened face to face, though we also used Microsoft Teams when needed. Work flowed through GitHub, where we tracked changes, managed tasks, and followed versions closely. Progress took shape on a Kanban board, making it easier to see who did what and how far things had moved. What slowed things down most was underestimating implementation complexity. Though the first steps seemed clear and under control, halfway through, new layers showed up - connections and checks that no one expected when drawing UML diagrams. Starting off with a solid plan helped us stay ahead of deadlines. Right from the beginning, having set dates kept things moving smoothly instead of piling up later. Tasks spread across the team fairly meant no one got overloaded along the way. Tracking each step carefully made it easier to adjust when needed. What stood out most was how small updates throughout improved results. Clear roles gave everyone direction without confusion. Planning like this turned complex work into manageable pieces. The whole experience showed that steady organization beats last-minute rushes every time.

8 Roles and responsibilities

The team members contributed equally to the deliverable.