Assignment Report

SFWRENG 2AA4 (2026W)

Bonus Assignment 2

Author(s)

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

GitHub URL

https://github.com/Nadeem-Mohamed/2AA4-code-gen-bonus-2.git

February 13, 2026

1 Executive Summary

We developed a code generator that is able to translate visual domain UML models into executable object oriented code. The goal of this assignment was to bring the abstract software design and its concrete implementation together by building a program that can bridge this gap.

We were able to develop a python based code generator that parses XML files from a UML tooling website called draw.io. The built solution successfully identifies concrete and abstract syntax and maps shapes such as rectangles and directed arrows to programming concepts such as classes, attributes, and inheritance. The developed solution generates code in Java.

The solution was verified against 2 scenarios, including the default baseline model that models the Car and Driver example, as well as a new university management system model that contains composition relation types to test the robustness and accuracy of the solution.

Overall, this assignment provides good hands-on experience in how a compiler thinks and illuminates the practical challenges of parsing XML files. Moreover, we were able to comprehend the advantages of deterministic code generators compared to the probabilistic nature of Large Language Models.

2 Syntax of the Model

Table 1 Below defines the formal language used for our code generator. It is able to map visual notation to the programming concepts, which will then be represented in Java.

Table 1: The syntax definitions

Concrete Syntax	Abstract Syntax	Well-formedness rule
Rectangle	Entity (Class)	Must have a name written inside it.
Simple Arrow	Association Relationship	Must be drawn between 2 entities. Can be circular to reference itself.
Arrow with empty triangle	Inheritance	Must be drawn between 2 entities. Must not be circular (no cyclic inheritance)
Label on Arrow Association	Role Name & Multiplicity	Must contain field name and (N) for list multiplicity and (1) for multiplicity of 1.

We chose simple rectangles to represent core domain concepts. The text inside the rectangle will define the name of the generated class. A directed arrow indicates that the course class is known about or contains the target class. Just like the normal definition, inheritance was defined as an empty triangle arrowhead. This then would use the extends keyword rather than creating a private field variable.

3 The Algorithm to Translate Models

The algorithm uses graph traversal concepts from our discrete math course (SFWRENG 2DM3) and file data processing concepts that were introduced in a first year programming course (ENGINEER 1P13). The core algorithm was developed as a group. The diagram.xml is basically a directed graph where the rectangles are vertices and the arrows are the edges. The relations between classes are the edges of the graph which stores tuples containing the source, target, and label. The algorithm does 2 passes, one to identify all the boxes (classes) and store it to the 'entities' dictionary. The second round will identify the arrows and safely look up the boxes. The algorithm has been described as pseudocode below.

```
GenerateCode(xml_input_file)

  INITIALIZE entities_map AS Empty Map <ID, Name>
  INITIALIZE relations_list AS Empty List <Relationship>
  INITIALIZE inheritance_map AS Empty Map <ChildID, ParentID>

  FOR EACH cell IN xml_input_file:
    IF cell IS "vertex" AND has "value":
      SANITIZE cell.value (remove HTML, special chars)
      ADD (cell.id -> sanitized_name) TO entities_map

  FOR EACH cell IN xml_input_file:
    IF cell IS "edge":
      GET source_id, target_id FROM cell

      IF source_id OR target_id NOT IN entities_map:
        CONTINUE (Skip invalid edge)

      IF cell.style HAS "endArrow=block" AND "endFill=0":
        MAP source_id -> target_id IN inheritance_map

      ELSE:
        DETERMINE cardinality:
          IF label CONTAINS "(N)" -> is_list = TRUE
          ELSE -> is_list = FALSE

        DETERMINE field_name:
          IF label EXISTS -> use sanitized label
          ELSE -> use name of target entity
```

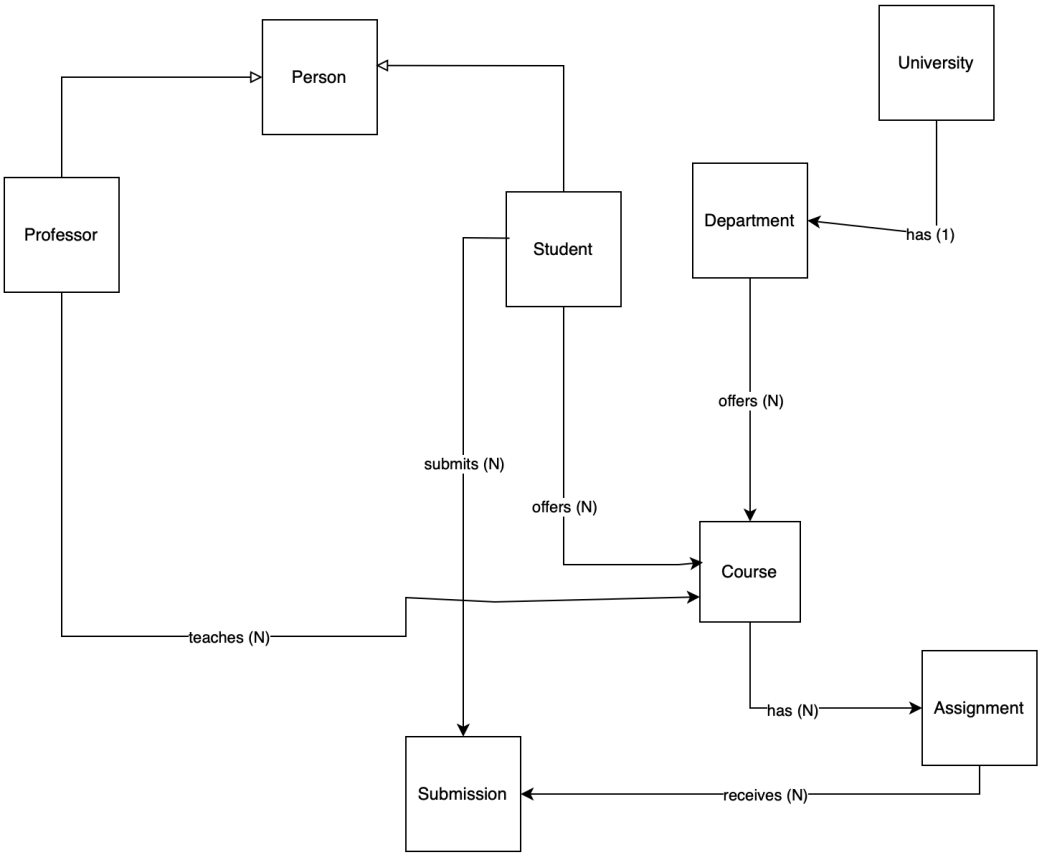
```
        ADD {source, target, field_name, is_list} TO relations_list

FOR EACH entity IN entities_map:
  CREATE FILE (entity.name + ".java")

  IF entity.id IN inheritance_map:
    parent_name = entities_map[inheritance_map[entity.id]]
    WRITE "public class " + entity.name + " extends " + parent_name
  ELSE:
    WRITE "public class " + entity.name

  FOR EACH relation IN relations_list WHERE relation.source == entity.id:
    target_type = entities_map[relation.target]

    IF relation.is_list IS TRUE:
      WRITE "    private List<" + target_type + "> " + relation.field_name
    ELSE:
      WRITE "    private " + target_type + " " + relation.field_name

  CLOSE FILE
```

4 Proof of Concept Implementation

The proof of concept was successfully implemented in python. The generator reads the XML file and parses through the structures and produces the correct functional code with appropriate visibility modifiers as required. The implementation was validated using 2 model code pairs stored in our GitHub repository.

Example 1: University System

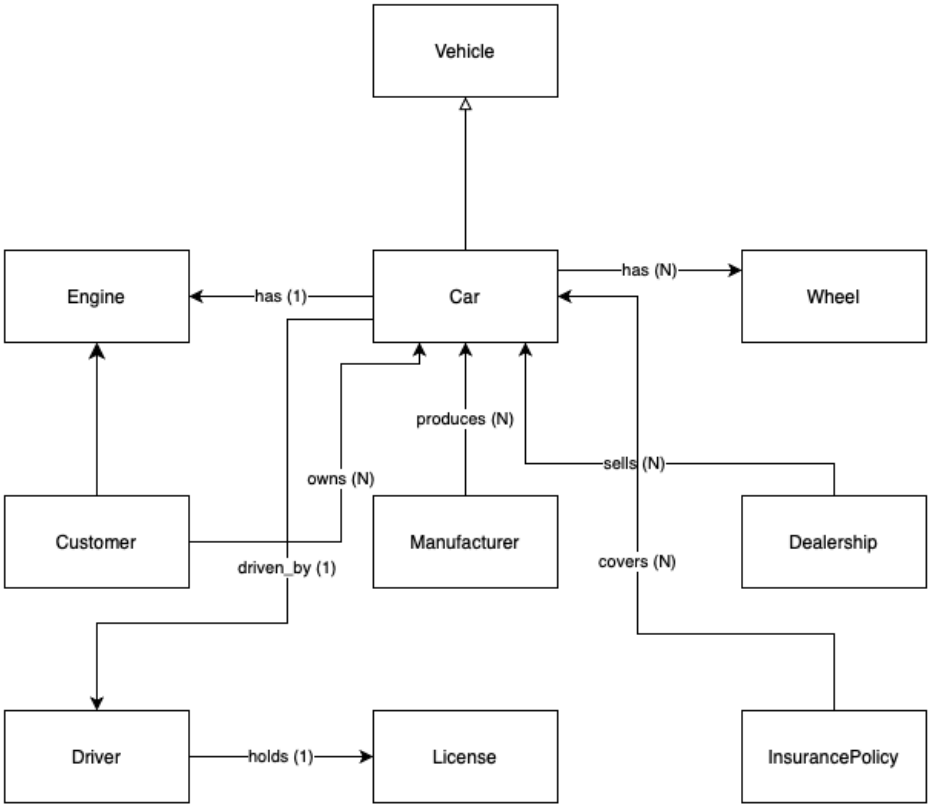
- Input: XML model (below is the visual version)



- Output: Corresponding Java files can be viewed here.

Example 2: Car System

- Input: XML model (below is the visual version)



- Output: Corresponding Java files can be viewed here.

5 Engineering Process

To extend our code generator to support another object oriented programming language is simple. We implemented Java code generation. If we were to write the algorithm for Python generation, the translation semantics remain the same, where rectangles are representing classes, and arrows map to classes or fields. We learned that extending the code generator to output C code would be more complex as it is a procedural language and not object oriented. To implement in C, we would need to redesign translation rules by using structs and pointer arrays. C would also require header .h files and the actual implementation .c files, while lacking garbage collection that many high level languages like Java feature. We would also have to write memory management functions for all the entities and this would make it a new architectural design rather than a simple syntax extension.

Our code and an LLM's generated code have different approaches. Our generator is a strict rule based system where it maps visual elements like rectangles and arrows to structural programming constructs such as classes, attributes and lists. The benefit of our implementation is the reliability for the given input .XML files. The limitation of our solution being highly rigid, meaning it cannot write Java code for more complex relationships. Another limitation is not easy to adapt to new requirements, because it would require more time to program the new requirements manually.

When using GPT LLM it is essentially predicting the output based on statistical patterns that are based on large datasets. The LLM's solution relies on context rather than hardcoded rules. The benefits of LLM's are flexibility and can write algorithms in a variety of programming languages and adapt to completely new paradigms without needing to be explicitly programmed. Moreover LLM's are very good at prototyping in a matter of seconds which makes planning and idea generation more quicker. The limitations of LLM's include unpredictability and being prone to hallucinations. It can also be prone to bugs, or inconsistent adherence to project structure.

We believe our solution is best for formal design workflows. Our algorithm can scaffold the correct class structure and associations for a large system with no errors. LLM's can maybe be beneficial when we need to go to the next step which would be implementation of the generated classes and methods. We can also use LLM's to clean up our code for better readability, which is a simple task that most LLM's will successfully complete without hallucinating.

Upon researching the top 2 workbenches, Eclipse features 2 primary language workbenches for developing custom design formalisms.

1. **Visual Workbench (using Sirius)**
   This tool allows developers to define graphical modeling workbenches. Essentially, instead of writing code to draw a shape, Sirius will define viewpoints to map domain concepts. For example, a Player visual representation. Sirius can handle the rendering, palette creation and property views all automatically.

2. **Textual Workbench (using Xtext)**
   This tool allows textual domain specific languages, allowing to define custom grammar, and automatically generate full compiler infrastructure. This includes parser, linker, and rich text editor with syntax highlighting and code completion.

To apply these workbenches to the Catan assignment, we can use Sirius to visually design custom Catan maps. We can use Hex and Port elements and drag them over to a canvas to visually configure the board layout and generate Java code to initialize the board configuration. The application of Xtext in Catan assignment would be to design a game replay to record game logs. Instead of verbose Java code, we can write a .txt file with commands to log actions of players in the game. Xtext would then parse this .txt file and generate the JUnit test cases to replay the exact game sequence for debugging.

6 Roles and responsibilities

The team members contributed equally to the deliverable.