Class Diagram Tool

A tool for displaying C++ class structures

Davis Bolt, Hunter Harris, Patience Lamb

CS 4488: Capstone Project

Spring 2022

Table of Contents

Abstract

This manual provides instructions of the usage and design of the Class Diagram Tool for future development purposes.

1. Introduction

The Class Diagram Tool was developed as a senior design project with the aims to accomplish four deliverables:

1. Determine the design of a class including: class name, methods, and fields.
2. Determine the inheritance of the class structure.
3. Determine the multiplicity of a class structure.
4. Display all relevant class information in a user-friendly GUI similar to Visual Studio's Class Diagram tool.

Through intelligent object-oriented design techniques as well as modern C++ and tool usage, the team was able to accomplish the four tasks within a reasonable amount of time in a stable and reliable manner.

The core of Class Diagram Tool is templated class design and regular expression parsing to extract information from C++ header files using a header prototyping and source file implementation structure consistent with the Google C++ Style Guide and common industry practices. The GUI is designed using the Dear ImGUI library coupled with a DirectX11 backend developed in Visual Studio 2019.

The current iteration includes drop-down menus on classes showing fields (variables) and methods and uses UML design to show multiplicities and inheritance.

2. Usage

The bin directory contains an executable of the Class Diagram Tool for runtime purposes. On open, the executable pulls up a blank canvas and allows the user to input a single or recursive directory into the filepath to generate a UML style diagram.

2.1 Front End Interface

The front end interface is composed of four parts: the filepath textbox, “Generate” button, “Search Subdirectories” checkbox, and the UML canvas. The interface also includes horizontal and vertical scroll bars to fit large amounts of nodes and edges within the diagram. As seen in Figure 1, the Car project included in the Examples directory fits a large number of classes on the canvas-- which can expand to accommodate many class nodes.

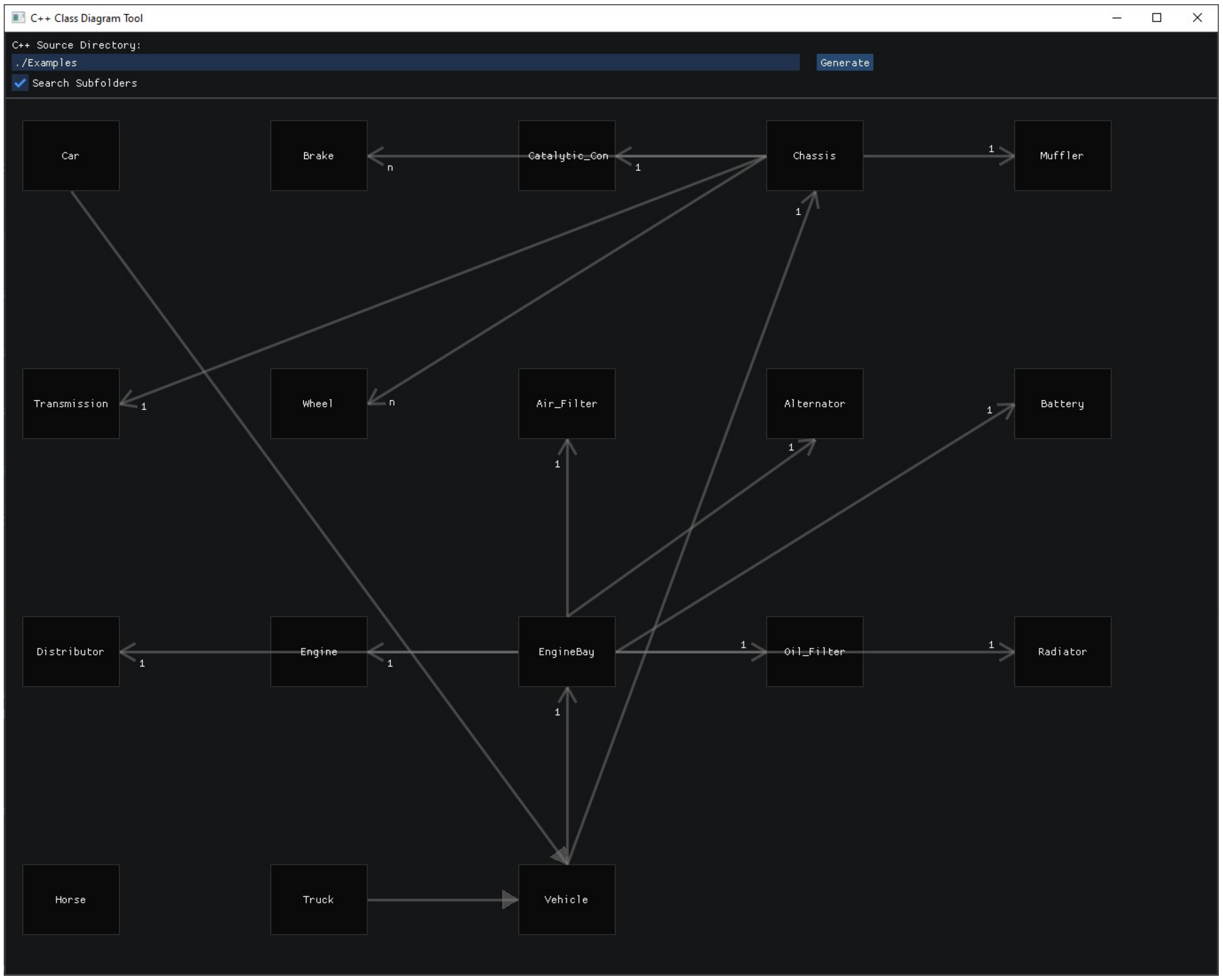


Figure 1: Car project example project structure.

The filepath to the project is typed in by the user in the filepath textbox and then generated via the "Generate" button. Unchecking the "Search Subdirectories" checkbox and then clicking the "Generate" button again loads a blank canvas into the Class Diagram Tool as the Car project is nested within multiple subdirectories.

By each class is defined as a square node with UML lines known as edges showing relationships between nodes. Inheritance is defined by a filled in edge to the superclass, as seen in Figure 2 where the Truck subclass inherits from the Vehicle superclass.

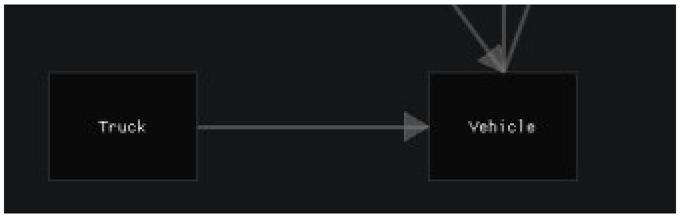


Figure 2: Truck inherits from Vehicle.

Directed association is indicated through a non-filled in arrows, where the multiplicity is indicated on the side using the associated class. For example, a vehicle's chassis can have n number of wheels and brakes for each chassis (1 to n multiplicity) as seen in Figure 3.

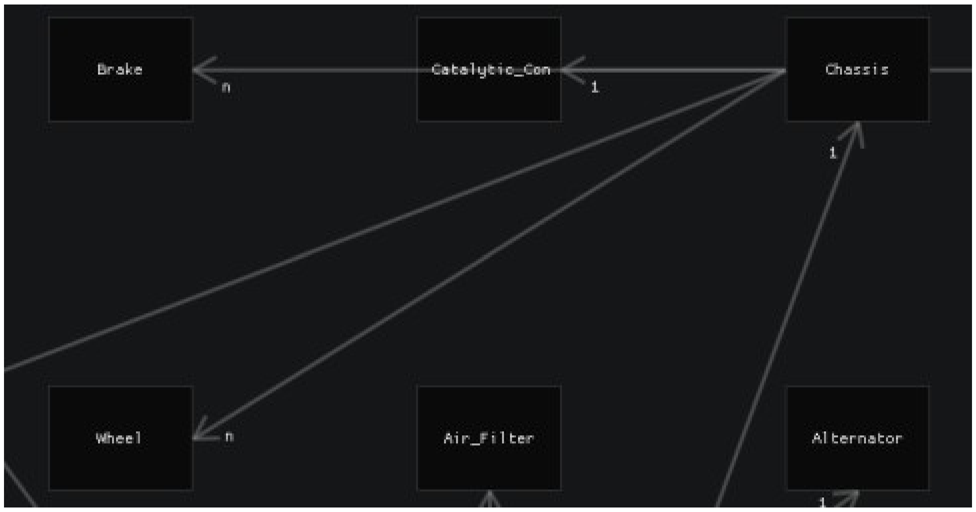


Figure 3: A 1 to n multiplicity for wheels/brakes to chassis

However, an engine bay only needs one air filter, making it a 1 to 1 multiplicity, as seen in Figure 4.

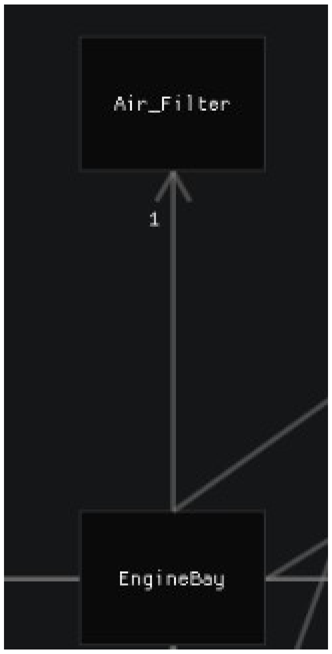
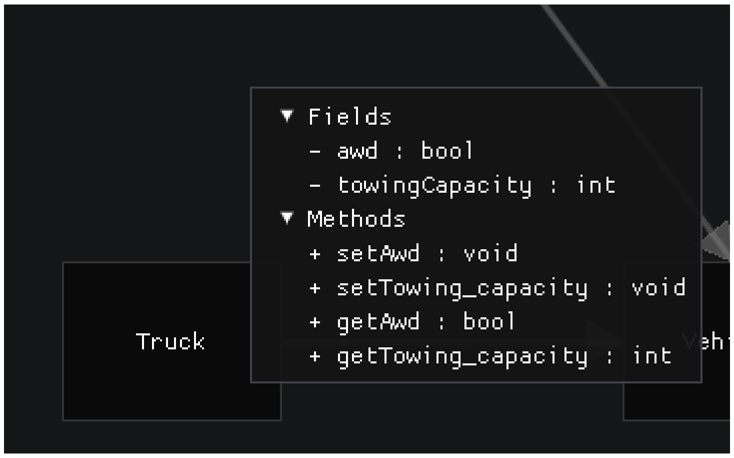


Figure 4: A 1 to 1 multiplicity between air filter and engine bay.

Current designs assumes any included header files indicate directed association and parse out the multiplicity.

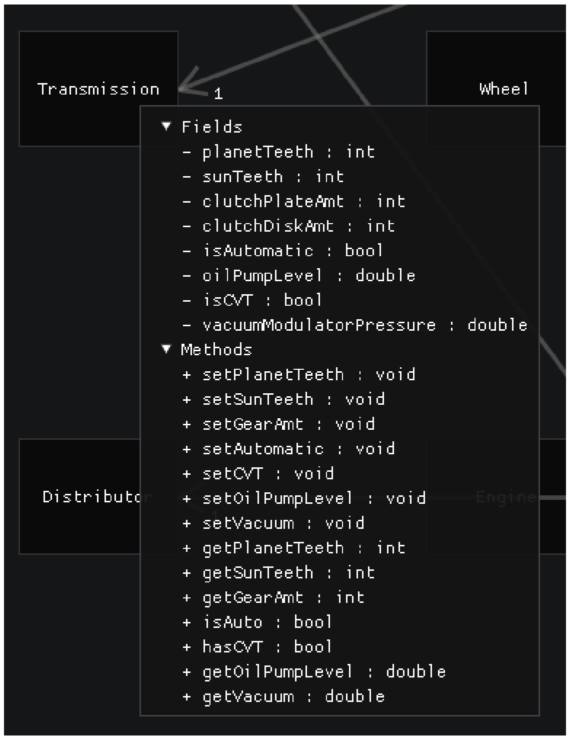
Clicking on a node shows the details about the class including fields and methods, for example, the Truck class has private variables of awd and towingCapacity and public methods of setAwd, setTowing_capacity, getAwd, and getTowing_capacity as seen in Figure 5. The keyword following the colon in the fields category shows the variable's type, whereas the keyword following the colon in the methods category shows the method's return type. Currently, the tool only recognizes private (-) and public (+) scopes.



```
▼ Fields
  - awd : bool
  - towingCapacity : int
▼ Methods
  + setAwd : void
  + setTowing_capacity : void
  + getAwd : bool
  + getTowing_capacity : int
```

Figure 5: Truck class objects.

The Class Diagram Tool can display large amounts of information associated with classes which dynamically expands to fit fields and methods. The Transmission class is a great example of a large amount of fields and methods, as seen in Figure 6.



```
▼ Fields
  - planetTeeth : int
  - sunTeeth : int
  - clutchPlateAmt : int
  - clutchDiskAmt : int
  - isAutomatic : bool
  - oilPumpLevel : double
  - isCVT : bool
  - vacuumModulatorPressure : double
▼ Methods
  + setPlanetTeeth : void
  + setSunTeeth : void
  + setGearAmt : void
  + setAutomatic : void
  + setCVT : void
  + setOilPumpLevel : void
  + setVacuum : void
  + getPlanetTeeth : int
  + getSunTeeth : int
  + getGearAmt : int
  + isAuto : bool
  + hasCVT : bool
  + getOilPumpLevel : double
  + getVacuum : double
```

Figure 6: Transmission class objects.

2.2 Input Description

The Class Diagram Tool uses header files to parse out UML design from protypes developed in the header. The file must include a .h, .hh, or .hxx extension. The class name is derived from the class definition within the file and the tool can parse through multiple definitions per file. The structure is seen below.

```
class class_name
{
  // body of class
};
```

Inheritance is derived from the colon operator, as seen in the subclass example below.

```
class subclass_name : scope base_class_name
{
  // body of subclass
};
```

Multiplicity is determined by instances of included classes where a singular instance is defined as below.

```
#include “otherclass.h”

class class_name
{
  scope:
      otherclass object();
};
```

A 1 to n relationship can be defined in terms of these types of collections: vector, deque, forward_list, list, stack, queue, priority_queue, set, multiset, map, multimap, unordered_set, unordered_multiset, unordered_map, or unordered_multimap. A collection is defined as below.

```
#include “otherclass.h”

class class_name
{
  scope:
      collection_type<otherclass> object;
};
```

Fields and methods are distinguished according to if they have a curly bracket (method) or not (field).

```
class class_name
{
  scope:
      keyword field;
  scope:
      keyword method(){};

};
```

3. Implementation and Development

Development was done using C++ 20 standards with the Dear ImGUI library in Visual Studio 2019 using the default MSVC compiler in Visual Studio. The Class Diagram Tool currently only compiles the GUI using DirectX11 within Windows Vista and newer. Efforts have been made to use OpenGL3 within Dear ImGUI and CMake for cross-platform compatibility, which have since been abandoned due to time constraints.

3.1 Class Design

The UML diagram for the class structure is seen in Figure 7.

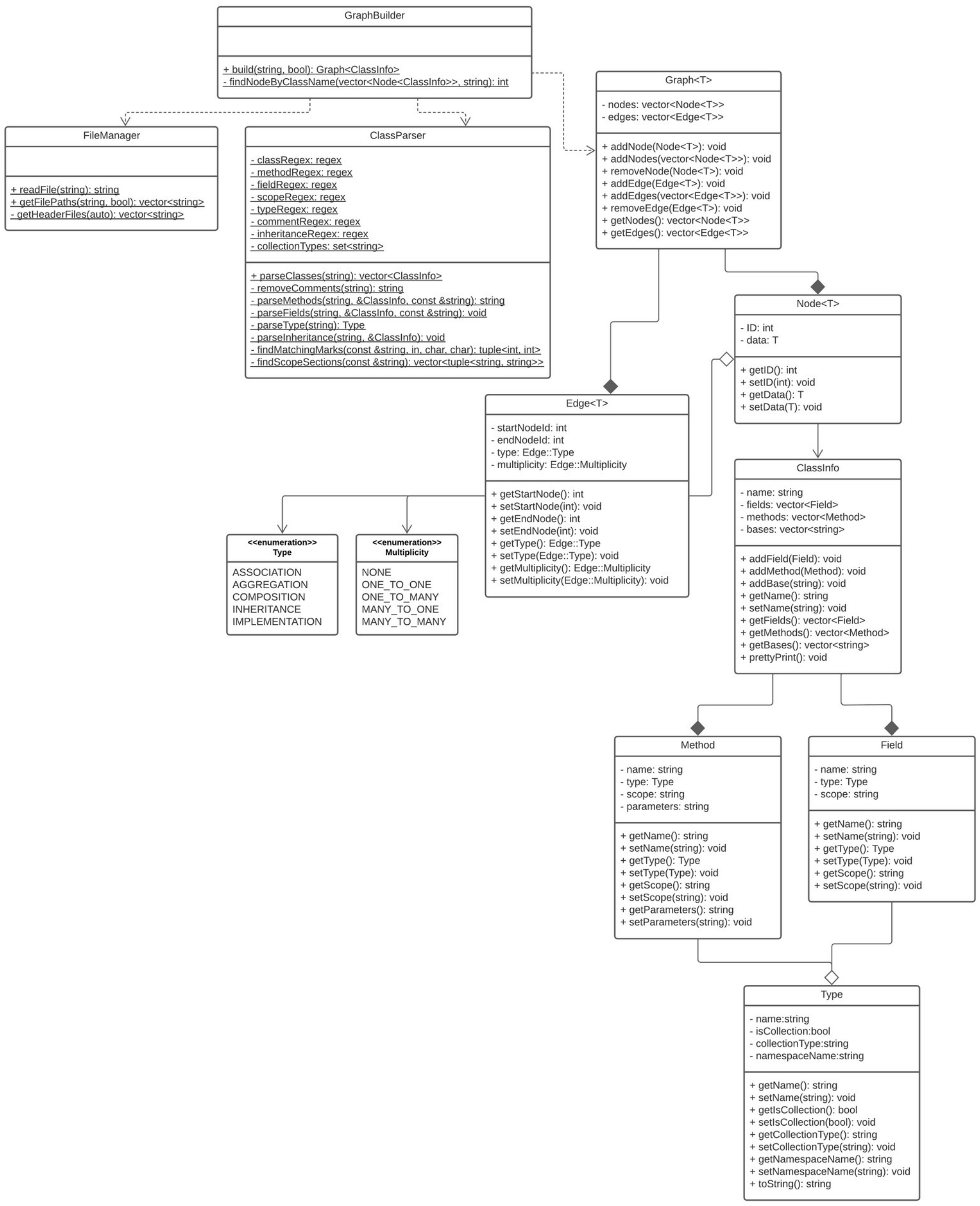

GraphBuilder
+ build(string, bool): Graph<ClassInfo>
- findNodeByClassName(vector<Node<ClassInfo>>, string): int
FileManager
+ readFile(string): string
+ getFilePaths(string, bool): vector<string>
- getHeaderFiles(auto): vector<string>
ClassParser
- classRegex: regex
- methodRegex: regex
- fieldRegex: regex
- scopeRegex: regex
- typeRegex: regex
- commentRegex: regex
- inheritanceRegex: regex
- collectionTypes: set<string>
+ parseClasses(string): vector<ClassInfo>
- removeComments(string): string
- parseMethods(string, &ClassInfo, const &string): string
- parseFields(string, &ClassInfo, const &string): void
- parseType(string): Type
- parseInheritance(string, &ClassInfo): void
- findMatchingMarks(const &string, in, char, char): tuple<int, int>
- findScopeSections(const &string): vector<tuple<string, string>>
Graph<T>
- nodes: vector<Node<T>>
- edges: vector<Edge<T>>
+ addNode(Node<T>): void
+ addNodes(vector<Node<T>>): void
+ removeNode(Node<T>): void
+ addEdge(Edge<T>): void
+ addEdges(vector<Edge<T>>): void
+ removeEdge(Edge<T>): void
+ getNodes(): vector<Node<T>>
+ getEdges(): vector<Edge<T>>
Node<T>
- ID: int
- data: T
+ getID(): int
+ setID(int): void
+ getData(): T
+ setData(T): void
Edge<T>
- startNodeId: int
- endNodeId: int
- type: Edge::Type
- multiplicity: Edge::Multiplicity
+ getStartNode(): int
+ setStartNode(int): void
+ getEndNode(): int
+ setEndNode(int): void
+ getType(): Edge::Type
+ setType(Edge::Type): void
+ getMultiplicity(): Edge::Multiplicity
+ setMultiplicity(Edge::Multiplicity): void
<<enumeration>> Type
ASSOCIATION
AGGREGATION
COMPOSITION
INHERITANCE
IMPLEMENTATION
<<enumeration>> Multiplicity
NONE
ONE_TO_ONE
ONE_TO_MANY
MANY_TO_ONE
MANY_TO_MANY
ClassInfo
- name: string
- fields: vector<Field>
- methods: vector<Method>
- bases: vector<string>
+ addField(Field): void
+ addMethod(Method): void
+ addBase(string): void
+ getName(): string
+ setName(string): void
+ getFields(): vector<Field>
+ getMethods(): vector<Method>
+ getBases(): vector<string>
+ prettyPrint(): void
Method
- name: string
- type: Type
- scope: string
- parameters: string
+ getName(): string
+ setName(string): void
+ getType(): Type
+ setType(Type): void
+ getScope(): string
+ setScope(string): void
+ getParameters(): string
+ setParameters(string): void
Field
- name: string
- type: Type
- scope: string
+ getName(): string
+ setName(string): void
+ getType(): Type
+ setType(Type): void
+ getScope(): string
+ setScope(string): void
Type
- name:string
- isCollection:bool
- collectionType:string
- namespaceName:string
+ getName(): string
+ setName(string): void
+ getIsCollection(): bool
+ setIsCollection(bool): void
+ getCollectionType(): string
+ setCollectionType(string): void
+ getNamespaceName(): string
+ setNamespaceName(string): void
+ toString(): string

Figure 7: UML diagram of Class Diagram Tool's structure.

The project was developed according to the Google C++ Style Guide where variable definitions, method prototypes and static methods are all defined within the header class. Implementation is done in the source file. Templating was done for the Graph, Node, and Edge classes to allow for definition flexibility in the next team's development efforts. The type and multiplicity of each of the edges are defined as an enumeration within the Edge class and the Node methods and fields as well as types are defined in separate classes.

GraphBuilder builds the nodes and edges into a graph and passes the information along to the GUI, allowing for interaction to the underlying components for the user. The FileManager reads in the directory and associated files within the program and feeds it to the GraphBuilder. The ClassParser parses the class name, methods, fields, inheritance, collections, scopes, and comments and feeds it into GraphBuilder. Graph adds nodes and edges into vectors to determine the relation and information about the nodes and edges and feeds it into GraphBuilder.

3.2 Tool Usage

The Dear ImGUI library is a self-contained immediate GUI library where the user only has to drag and drop the imgui.cpp and imgui.hh into their library or import the library from the GitHub repository. Using the DirectX11 example found within ImGUI's examples/example_win32_directx11 main.cpp, the Class Diagram Tool was built upon the DirectX11 example in Visual Studio 2019. Although not an event-driven library, the Dear ImGUI library allows for limited user interaction with minimal effort. Due to the relative newness of the Dear ImGUI library, documentation and examples are lacking, but with the amount of attention the library is getting, it will only be a matter of time before it becomes a standard practice to use it.

3.3 GUI Design

The GUI was loosely based on other class diagramming tools like Doxygen and Visual Studio's Class Designer. The idea was to make the project standalone to allow for ease of use unlike Doxygen but still have a relatively simple and descriptive display. The choice of darker colors is consistent with many user's desires for dark mode displays for working as well as the movement towards dark mode being a default state. The choice to show relationships through UML style arrows and numbering was to make it more intuitive for the user to use without having to learn an entirely new toolset or layout. Ultimately, the GUI was designed to be familiar and easy to use for all skill levels to get a comprehensive layout of their classes for debugging and presentation purposes.

4. Conclusions

The Class Diagram Tool acts as a minimum viable product for future use and development. The relatively simple design and prototype enhances readability for future developers to elaborate on.

4.1 Summary

In its current state, the Class Diagram Tool parses header files into classes, fields, and methods and determines multiplicity and inheritance in relation to other classes. The parsed data is broken down into nodes and edges and graphed on a GUI. The goal of the project was to achieve a product similar to Visual Studio's Class Designer tool but allow for a standalone project to be imported.

The Spring 2022 team was able to create a reliable tool which statically generates a UML style diagram given an input directory.

4.2 Future Work

Future efforts to the Class Diagram Tool project include the following:

1. A navigation method in 2D space (zoomed out window in corner).
2. Zooming capabilities.
3. Organization of superclasses and subclasses.
4. Ensure edges do not cross over nodes or other edges.
5. Determine and graph composition, aggregation, dependencies, and generalization.