MASS JAVA Benchmarking

Autumn 2022 Term Report

Anirudh Potturi

In Partial Fulfillment of the Requirements

For the Degree of

Masters of Science

Under Guidance of Dr. Munehiro Fukuda

University of Washington

https://orcid.org/0000-0002-9270-9628

Contents

LIST OF FIGURES

Chapter 1

Introduction

The purpose of this paper is to provide a summary of the work performed in the Autumn Quarter of 2022. This quarter's goal was to develop applications using an Agent-Based Modelling toolkit. A previous student, Vishnu Mohan, developed Eight agent navigational patterns he identified that were commonly used and automated their execution [1]. This work is intended to simplify the programmability and enhance the migration capabilities of agents. Four applications were selected to apply these navigational patterns: Breadth First Search, Triangle Counting, Closest Pair of Points, and Range Search. However, sufficient work was needed to prove that these improvements improve the programmability and offer additional performance improvements. Vishnu compared what is termed as Old MASS and New MASS. To further extend this work, the same applications had to be developed using a competitor Agent-Based Modelling Toolkit.

1.1 History of Agent-Based Modeling

The essential phase of this quarter was to dedicate time to understanding and performing a literature review of Agent-Based Computations and Modelling. I thoroughly researched and gathered resources to help me realize the importance of the work done at The DSLab. This phase was beneficial because I was introduced to ideas like Complex Adaptive systems and Agent-Based Simulations. I also learned the technical methodologies of computational models of multi-agent interactions that were first introduced in the 1940s. Understanding the rich yet limited history of Multi-agent simulations

and modeling was vital to understand the underlying agent migration techniques of the Computational Geometry applications. The agent migration patterns used were inspired by classic concepts from areas like cellular automata theory (The Von Neumann and Moore Neighborhood methods) [2], etcetera. Vishnu Mohan automated the migration of agents to these neighboring places of agents.

1.2 Agent-Based Modelling Toolkit Selection

The first and foremost step to be done was to select an ABM Toolkit environment. The two best and most suitable options available to us were NetLogo and Repast-3. NetLogo uses the concept of Turtles as agents and Patches as Places. It is based on the Logo Dialect and thus requires developers to adapt to the language. Repast-3, short for Recursive Porous Agent Simulation Toolkit, is a collection of Three agent-based modeling libraries: Java-based Repast J, C based Repast .NET, and Python-based Repast Py. Repast J was a better option mainly because of the language. On further analysis, we found that Repast Simphony is the latest development built on top of Repast 3 with a completely new code base [3]. Repast-J was also significantly improved with modular plug-in and use components. Thus, we proceeded to move forward and develop the four applications using Repast Simphony.

The following sections explain the work I carried out throughout this quarter.

Chapter 2

Repast Simphony

As part of this quarter, my work with Repast Simphony was to ensure we had four applications, as mentioned. A former student, Max Wenger, worked on two applications: Triangle Counting and Closest Pair of Points using Repast Simphony. My role with both applications was to ensure they were in working order and touch up on anything if necessary. I worked on refactoring this code and renaming the components for consistency. The reasoning behind this will be explained later in this section. Breadth First Search and Range Search had to be developed from scratch. We wanted to develop these applications while trying to mimic MASS Java. For this, I first developed the base code that took core concepts of MASS Java. In particular, the base code consisted of Agent, Place, and Agent Manager components. The Agent Manager is essential to the Repast Simphony applications we developed. The base code was implemented from scratch since it was not readily available. As the name suggests, the agent manager is responsible for managing operations like agent creation, agent termination, and controlling agent migration. The Agent component creates objects that will act as the agents in applications. The Place component is used to develop the space and acts as a point of invocation of method calls to the AgentManager. Implementing features of MASS Java in Repast Simphony for each application requires slight modifications to this base code. The motivation behind developing this code is to streamline future work in the group with this toolkit.

2.1 Breadth First search

This was the first piece of development work for this quarter. The idea of developing the MASS Base code was a result of this application. Breadth First Search, as we know, is a widely used Tree traversal technique with real-world applications like Peer-to-peer networks.

All Repast Simphony classes require a class dedicated to building the application's context. In this application, The BFSBuilder is the component that does it (see Appendix A: BFSApp). It adds objects of the graph into the context, which will later be displayed on the GUI. GraphGen is the component that generates a graph and neighbors of each node of the graph. In this case, Each Place acts as the node of the graph, and each Place has neighboring places. Each Place is initially assigned an Agent that belongs there. At the beginning of the Simulation, only the Agent at Place 0 is active. All other agents are inactive, meaning they cannot migrate or perform any activity. First, Agent 0 looks for its neighboring places and picks one to which it will migrate and return. On completion of this, the Agent triggers the Agent at the Destination place to be activated. This activation, of course, is only done by AgentManager. If more places have yet to be visited, a scheduled method in each Place triggers the Agent residing in that Place to migrate to a neighboring Place. Finally, with no more places to be explored, the Simulation is halted.

2.2 Range Search

Range Search is an application with real-world uses in Geographical Information Systems. This Computational Geometry problem uses a space over which points are distributed. Users can query for all the points in the range of their choice's maximum and minimum - x and y coordinates.

The context builder for this application resides in the RangeSearch component (see Appendix B: RangeSearch). This component is responsible for Reading the input file,

Constructing the KDTree, and, lastly, building the Context. The AgentManager in this application initiates the search for nodes in the tree within the query range. Vishnu Mohan's Range Search had to be reverse-engineered for this application to be as close as possible to that of the MASS Java version. This meant that the underlying code of MASS had to be understood and reimplemented for Repast Simphony. This further extended the MASS Base Code for this toolkit.

This application mimics the working of the MASS Version. An agent first starts at the root node. If the coordinates' bounds are on the node's left child, the agent continues traversing, i.e., migrating. Every time a point lies on the right child node, AgentManager triggers a new agent to take this path and traverse. To better understand the number of agents used for the traversal, a method of counting the number of agents required in the simulation was added. A critical difference between the Repast and MASS versions is that while MASS spawns a new agent, Repast Simphony activates an agent. Thus, every time an agent was activated in the application, the counter was incremented by one.

2.3 MASS Base Code

The MASS Base Code extracts all the structural elements developed to mimic MASS in Repast Simphony. This piece of work is derived from the previous two applications developed. It contains components allowing future users to reuse existing code and focus on algorithm implementations only. The ApplicationBuilder component (see Appendix C: MASS Base Code) is a template component that provides a basic structure of context building in Repast. While users may choose to develop this component from scratch independently, it is still available for those who wish to simplify their work. The FileInputReader component is re-engineered from existing work done by former students in The DSLab. This component is modified for Repast to read input data files and populate Places and Vertices. A Place provides an environment for an Agent and part of a graph or a problem to reside in. The Vertex component is used to construct

graphs. Every vertex of a graph or application resides in a Place. Each place is also updated each time an agent visits the place with the agent's footprint. This footprint can be beneficial when building applications where an agent may want to know the most recent agent that visited the place. This can also help in enhancing migrations where an agent does not visit a place it just visited. To be precise, an agent may be terminated if it realizes it has entered a cycle of node/nodes.

The MASS Base Code is currently being updated with more features that have been identified as necessary.

Chapter 3

Code Analysis Tools

3.1 Java Static Code Analysis Tool

The Java Static Code Analysis Tool builds upon the work done by a former High School Intern, Kent Fukuda. Kent developed a tool called LoC, which stands for Line of Code. The tool can read an input file, identify whether it is a Java or C++ file and read each line. The tool identified the number of lines in the code, the number of comments, and the number of blank lines in each file without using a Parser, which was impressive. However, to further analyze the contents of each file and identify variables and methods, we needed a Parser that could accomplish the goal. For this tool, JavaParser was used to parse classes written in Java. JavaParser is an open-source library that provides a multitude of methods that can be used to parse and extract fragments of code. This library was used to expand the application of the Code Analysis Tool to identify the number of variables and methods in a Class. Furthermore, the tool can identify and count the number of conditionals and iteration statements used in a class. The purpose of counting the logical paths in a class is that we wanted to perform the Programmability Analysis of MASS vs. Repast. A good metric for this was to compute the Average Cyclomatic Complexity. Cyclomatic Complexity counts the number of logical paths in a method. Each method is initialized with a value of One, and for every Logical Statement encountered, we increment the Cyclomatic Complexity of the method by One. The metrics of each method are displayed at the end. We also compute the average Complexity of a method in a class and display the results.

The InitiateVisit Component initiates the MethodVisitor Component to visit sections of the code and increment the counter (see Appendix D: Java Static Code Analysis Tool). A list is used to track the count of each logical statement encountered in the code. Additionally, the tool can be run from any directory by the user. For example, the tool can be run from within a MASS Application's directory. The tool will identify the current working directory, explore all files and folders in the directory and analyze relevant code files.

3.2 C++ Static Code Analysis Tool

C++ code is hard to parse. A previous attempt was made by a student, Kevin Wang, to develop a similar tool for C++ Applications but remained unsuccessful. Selecting the appropriate library for this tool was more challenging than we thought it would be. This was because much of what is available needs more documentation and usage examples to understand the usage of methods available. We explored options like CPPParser and ANTLR and eventually decided on using ANTLR. ANTLR is a parser available for many languages. This tool was developed using ANTLR's C++ Grammar files for Java. The grammar files contain rules which define how C++ code is parsed. This tool is implemented similarly to the Java Code Analysis Tool. The only key difference and quite challenging was the parsing. During development, we realized that ANTLR surprisingly had no specific rules to parse an else-if statement as intended. On the contrary, the 'if' and 'else' statements were parsed correctly. The 'else-if' statement was being parsed separately as an 'if' and an 'else' statement. This introduced an inconsistency in our results because the counts of statements produced were incorrect. To fix this, we defined a rule in ANTLR's grammar file to parse the 'else-if' statement successfully.

Chapter 4

The International Conference on Agents and Artificial Intelligence 2023

The ICAART 2023 paper demonstrates the improvements in parallel performance with the Automated Migration and presents a programmability comparison between New MASS and Repast Simphony [4]. The paper was based on Vishnu Mohan's contributions toward generalizing agent navigational patterns and automated agent migrations [1]. My work for this quarter helped us perform programmability analysis between New MASS and Repast Simphony. The Static Code Analysis Tool for Java was used to analyze existing applications developed with New MASS and their Repast versions. From the metrics we produced, we could conclude that the overall Lines of Code in Repast were higher than in MASS [4]. From Table 4.1, we proved that the LoC in Repast was higher because of the need to develop the structural elements of applications in from scratch.

Table 4.1: Quantitative Programmability Comparison between MASS and Repast Simphony [4]

Measures	Libraries	BFS	Tri Count	Range Search	CPP
LoC	MASS	79	175	400	362
	Repast	432	260	539	314
Cyclomatic Complexity	MASS	2.25	3.875	3.944	3.1
	Repast	1.785	2.45	2.6	2.31
Agent LoC (A)	MASS	17	40	122	95
	Repast	229	76	139	109
Space LoC (S)	MASS	19	37	120	10
	Repast	111	94	130	43
Model Mgmt LoC - (A + S)	MASS	43	98	158	257
	Repast	92	90	270	162

Additionally, we compared both versions of each application to justify our findings further. Before performing a detailed code analysis, we assumed that the Cyclomatic

Complexity of MASS would be higher than that of Repast. Our results proved that our assumption was correct. We were expecting the cyclomatic complexity of MASS, in general, to be high because of the nature of developing applications. MASS saw reduced LoC, but the Cyclomatic Complexity increased because of the frequent use of conditional statements to call appropriate base methods. A detailed analysis of the findings was explained in the ICAART paper.

Chapter 5

Results and Analysis

We know that Repast is a GUI-based ABM toolkit. For this reason, all applications run on it depend on the interface heavily. Each application and the structure of the problem space are rendered on the display. The application seen in Fig. 5.1 is Breadth-First Search with 50 Vertices connected by edges. Buttons on this interface allow users to start and end simulations, besides offering many other options. We started facing issues during the testing process when we tried benchmarking the applications with the largest data sets. The GUI struggled to render so many objects onto the display, which is not surprising considering that everything was being done on One system. We decided to leave the problem space off the display to continue testing for the largest data sizes. This beats the purpose of having an interface.

5.1 Breadth-First Search

The BFS Application results were promising at the beginning of the tests. In Table 5.1, the time to finish a simulation with 10 and 100 vertices was surprisingly close. We expected to see a decline in performance at some point because, with hundreds and thousands of agents traversing the Tree, the application demands more compute power. Another reason we believe causes an increase in time is the checks performed by every active agent at a given time. Since agents are responsible for identifying places around them, they need to check neighbors of the current place. Next, agents request AgentManager to see whether or not the neighboring places have been visited. Based on AgentManager's response, agents continue their exploration.

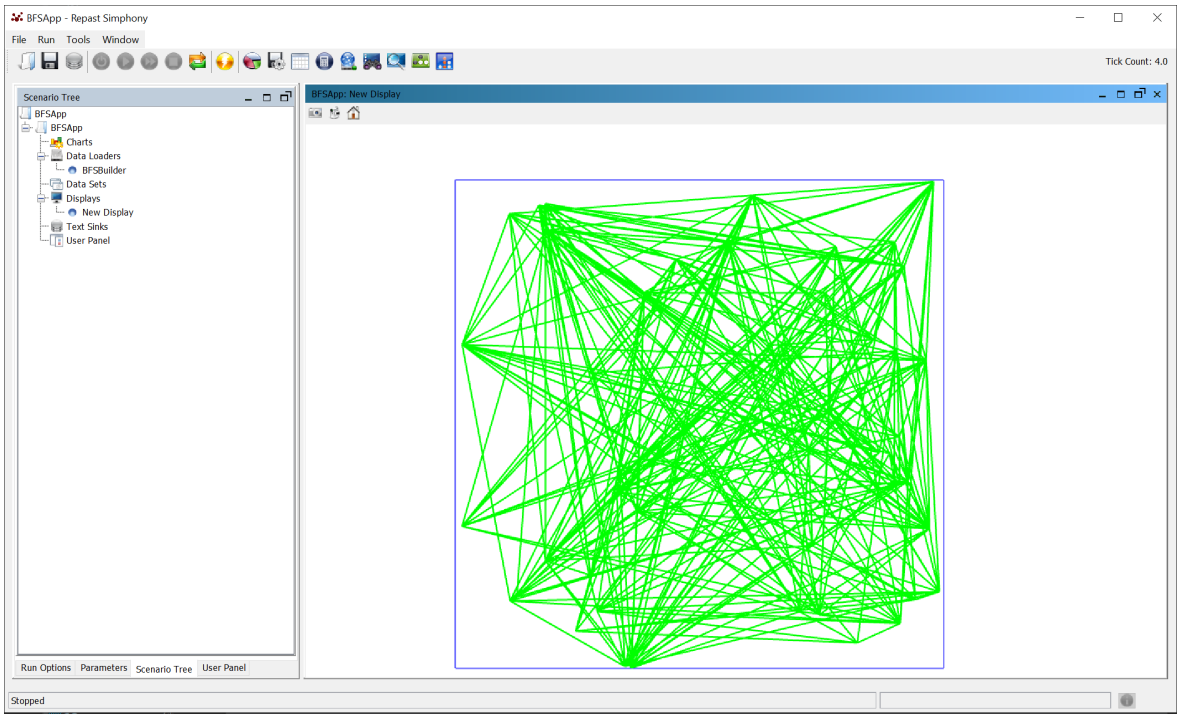


Figure 5.1: JavaCodeAnalysisTool Sample Output Part 2: Cyclomatic Complexity of Methods

Table 5.1: Breadth First Search Benchmarking Results

Number of Vertices	Ticks	Elapsed Time (in seconds)
10	4	1.6
100	3	1.5876297
1000	3	4.7715473
10000	3	3837.312504

5.2 Range Search

The results of Range Search were surprising when compared to Breadth First Search. From Table 5.2, the time is taken to construct the KDTree very fast, regardless of the data size. This is because of the approach used to construct the Tree. The Tree is constructed recursively by halving the data and creating Left and Right branches. Overall, the Range Search process was also very efficient because agents do not traverse the entire Tree. Instead, agents check the bounds/coordinates passed as input starting at the Root node. If the points are present on the left branch, the agents traverse to the left. This process is vice versa if the bounds are on the right-side branch. At every stage, agents perform this check. As a result, there is a decrease total number of

agents used up in the entire search process. An additional piece of work added helped us calculate the total number of agents required in the application.

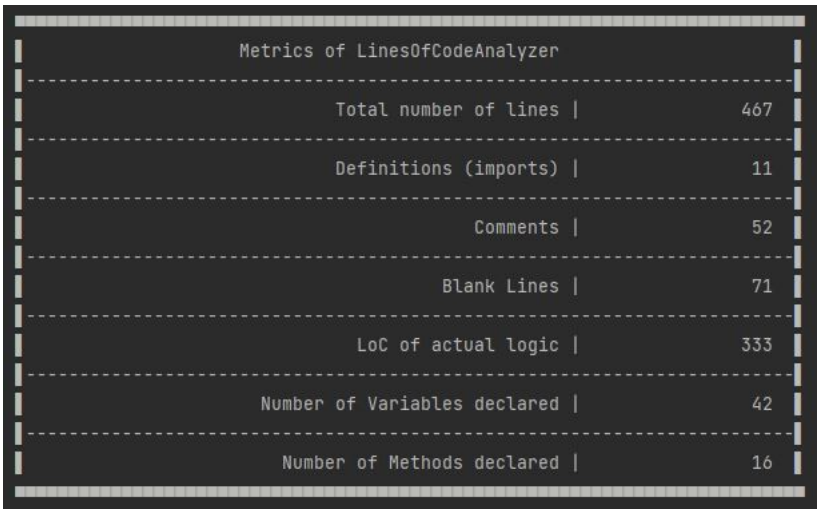
Table 5.2: Range Search Benchmarking Results

Number of Vertices	Ticks	KDTree construction Time (sec.)	Range Search Time (sec.)	Number of Agents
100	4	0.0008537	1.5650227	13
500	6	0.0051113	1.5223931	39
5000	8	0.1351563	1.5709776	202
10000	10	0.3765646	3.2501259	376

5.3 Java Static Code analysis Tool

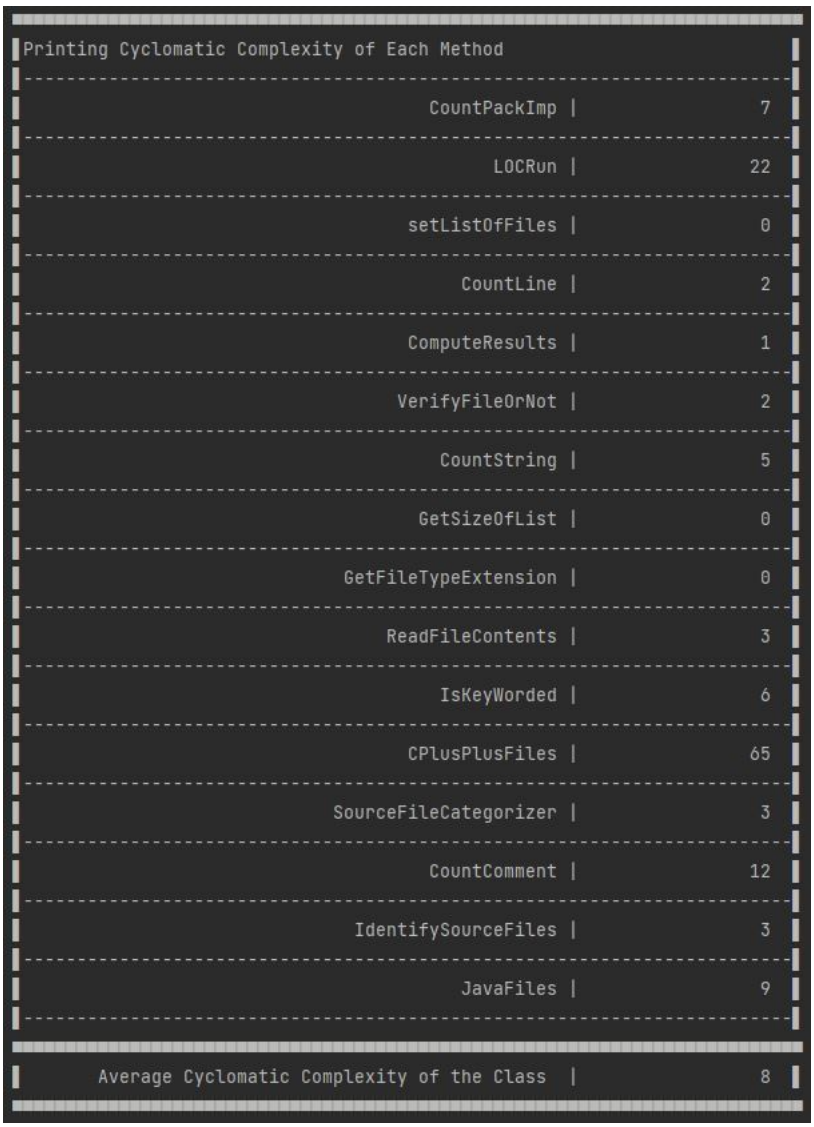
The Code Analysis Tool was beneficial in presenting results to the ICAART submission. The detailed metrics helped draw insights and conclusions. This section will describe the results from the very first test of the application. The code analyzed was the source code of this tool. The following metrics are the result of One class only. The tool displays all the metrics explained below for every class present. To simplify the readability of the output, it is divided into three segments.

Firstly, the tool displays simple data about the class (Fig. 5.2). This segment gives a brief overview of the code at a higher level. Since the main method of a class is also a method in Java, the tool does not necessarily distinguish the main method in a class in a unique way. In the second part, the tool displays the metrics of each method of a class (Fig. 5.3). It displays the name of the method defined as is and the Cyclomatic Complexity of that method. Toward the end of this segment, the tool displays the average Cyclomatic Complexity of a method in the class. In the final segment, the tool displays a count of all the class's Conditionals, Loops, and Error Handling statements (Fig. 5.4). This segment is beneficial in realizing what increases/add to the Complexity of the application.



Metrics of LinesOfCodeAnalyzer	
Total number of lines	467
Definitions (imports)	11
Comments	52
Blank Lines	71
LoC of actual logic	333
Number of Variables declared	42
Number of Methods declared	16

Figure 5.2: JavaCodeAnalysisTool Sample Output Part 1: Basic Metrics



Printing Cyclomatic Complexity of Each Method	
CountPackImp	7
LOCRun	22
setListOfFiles	0
CountLine	2
ComputeResults	1
VerifyFileOrNot	2
CountString	5
GetSizeOfList	0
GetFileTypeExtension	0
ReadFileContents	3
IsKeyWorded	6
CPlusPlusFiles	65
SourceFileCategorizer	3
CountComment	12
IdentifySourceFiles	3
JavaFiles	9
Average Cyclomatic Complexity of the Class	8

Figure 5.3: JavaCodeAnalysisTool Sample Output Part 2: Cyclomatic Complexity of Methods

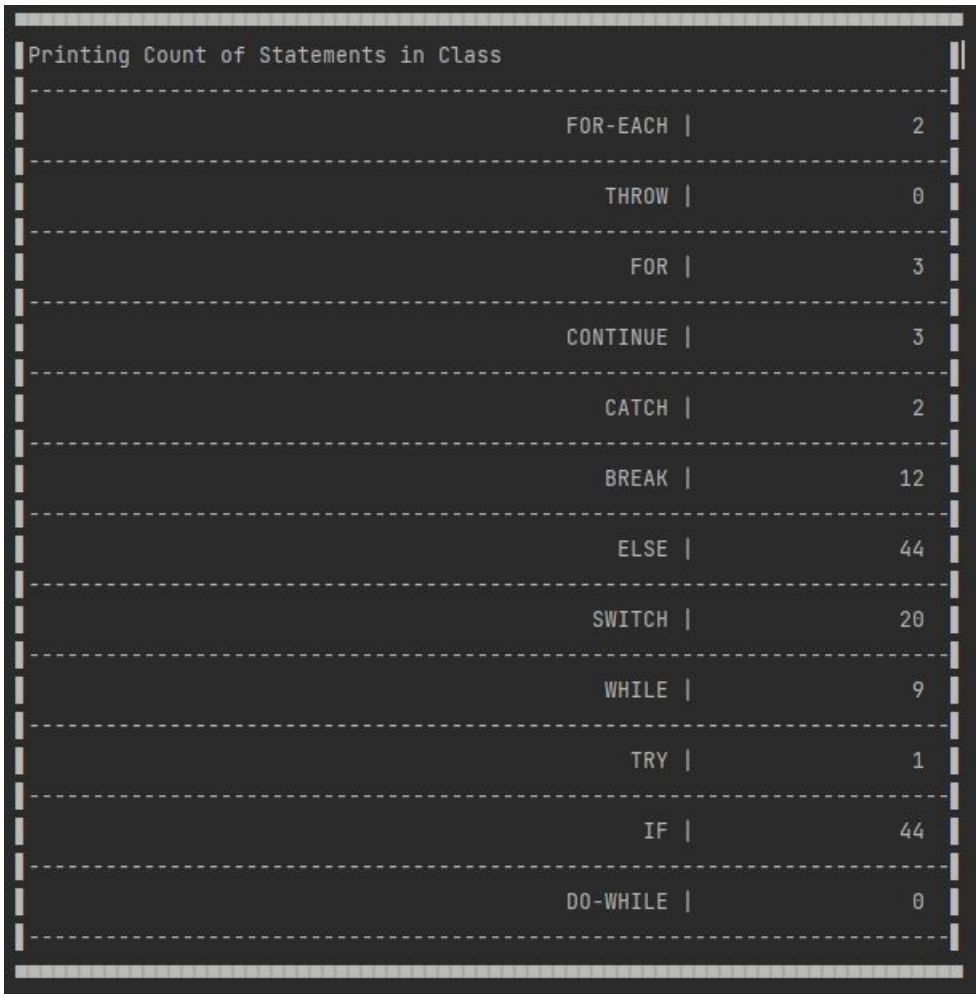


Figure 5.4: JavaCodeAnalysisTool Sample Output Part 1: Count of Statements

ACRONYMS

Symbol	**Meaning**
ABM	Agent-Based Modelling
ANTLR	ANother Tool for Language Recognition
LoC	Lines of Code
MASS	Multi-Agent Spatial Simulation
GUI	Graphical User Interface
BFS	Breadth-First Search
CPP	Closest Pair of Points
CPPParser	C++ Parser

REFERENCES

[1] V. Mohan, "Automated agent migration over structured data." UW Master's White Paper, 2022.

[2] C. Macal and M. North, "Tutorial on agent-based modelling and simulation," in *Journal of Simulation*, September 2010. https://doi.org/10.1057/jos.2010.3.

[3] M. J. North, N. T. Collier, J. Ozik, E. R. Tatara, C. M. Macal, M. Bragen, and P. Sydelko, "Complex adaptive systems modeling with repast simphony," March 2013. https://doi.org/10.1186/2194-3206-1-3.

[4] V. Mohan, A. Potturi, and M. Fukuda, "Automated agent migration over distributed data structures," Accepted on 12/22/2022.

APPENDICES

A BFSApp

Figure A.1: BFSApp: Structure of classes

B RangeSearch

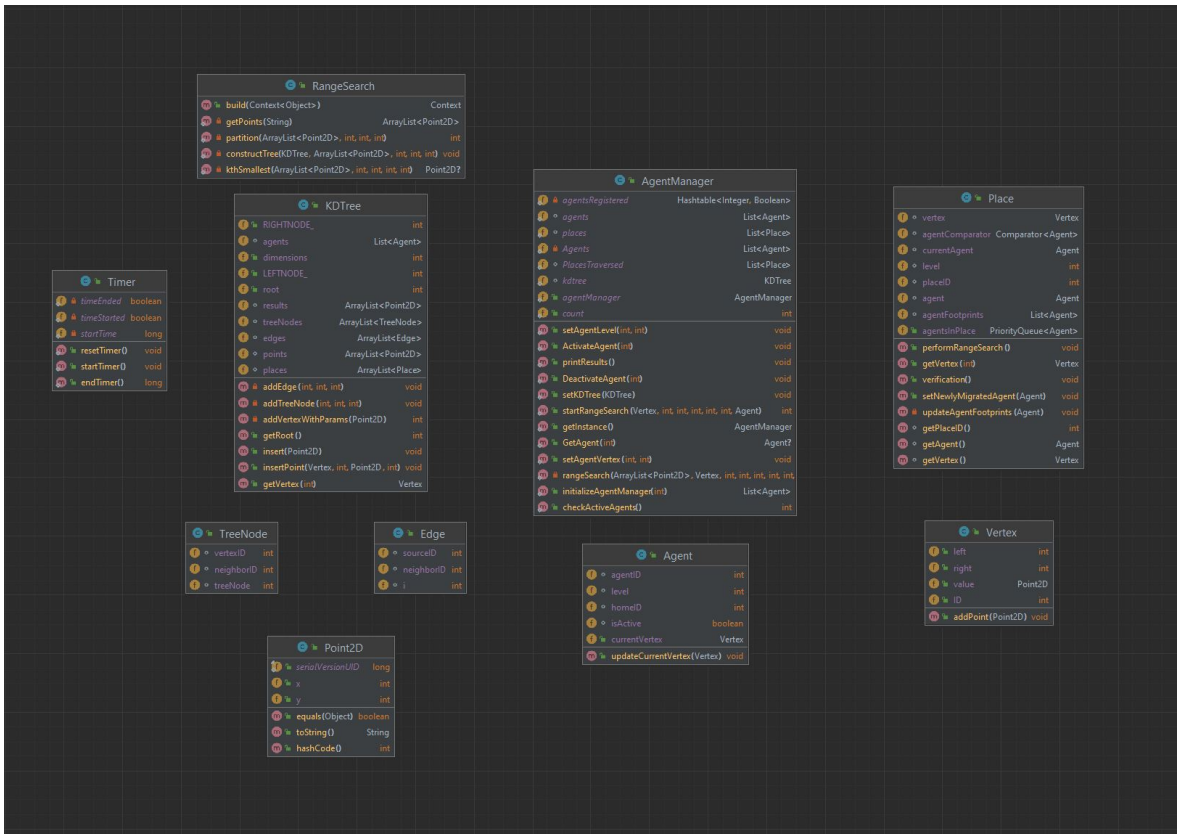


Figure B.1: RangeSearchApp: Structure of classes

C MASS Base Code

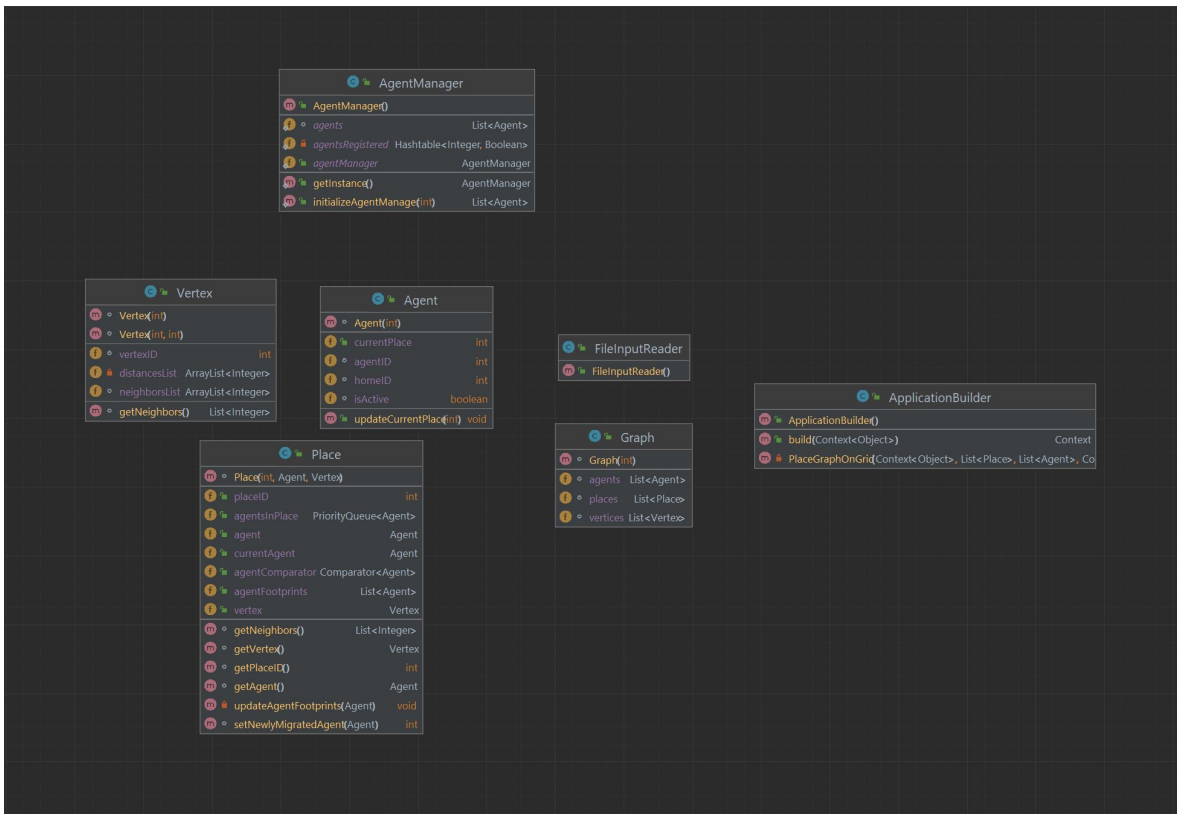


Figure C.1: MASS Base Code: Structure of classes

D Java Static Code Analysis Tool

Figure D.1: JavaCodeAnalysisTool: Structure of classes

E C++ Code Analysis Tool

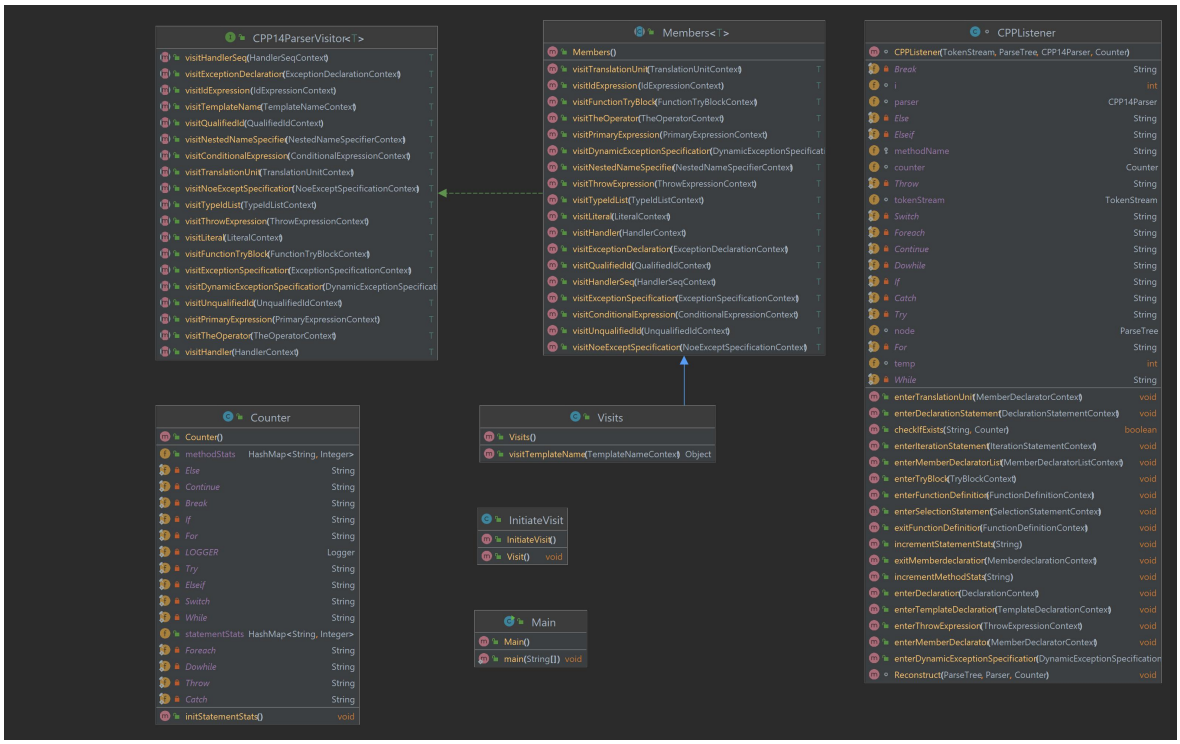


Figure E.1: C++CodeAnalysisTool: Structure of classes