Sprint 4 Group Report

By

David Molina 40111257

Karim Rhoualem 26603157

Tristan Gosselin-Hane 40093465

Mahad Janjua 40102893

Rhys Rosenberg 40071895

Alessandro Morsella 40096192

Nizar El Jurdi 40053913

A report submitted in partial fulfillment of the requirements of SOEN 341

Concordia University

December 2020

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1.0 DESCRIPTION

C_m Assembler is a cross-assembler written in java. Its purpose is to process C_m assembly language files and assemble them into C_m binaries according to the C_m instruction set. It is also capable of producing a listing file detailing the structure of the program (its labels, instructions, operands and address offsets) if desired. It is fully compliant with the Cm assembly language specification and produces reproducible results.

Internally, C_m Assembler reads an assembly source file and parses it into an in-memory intermediate representation (IR). It then parses the intermediate representation in order to translate it into the appropriate binary form according to the C_m assembly specification. C_m Assembler contains a robust command-line interface with multiple options allowing users to generate a listing file, print verbose output and print help messages. C_m Assembler also includes a helpful error-reporting mechanism allowing users to find out exactly where in the assembly file errors have occurred during the assembly process in the event of a parsing failure. Finally, C_m assembler ships with a full test suite allowing for developers to catch possible regressions early in the development process if any are introduced.

2.0 DOMAIN DICTIONARY

Concept	Definition	Modeled as	System name
Assembly file	A file (.asm) that contains Cm assembly instructions.	File (.asm)	[fileName].asm
Listing file	A file (.lst) that contains the addresses, the line numbers, and the machine code for each assembly instruction along with the original instructions.	File (.lst)	[fileName].lst
Tokens	Objects recuperated from the lexer that categorizes each word (mnemonics, operands, labels, etc.) found in the assembly file.	Package of Java classes	Tokens
Lexer	A Java class that uses a reader to read the assembly file to create different types of tokens.	Class	Lexer.java
Parser	A Java class whose purpose is to parse a series of tokens into LineStatements and Instruction objects.	Class	Parser.java

Table 1: Important Concepts

Action	Definition	Modeled as	System name
Assemble	Translates Instruction objects into binary representation, and hexadecimal representation. Creates the listing file if specified.	Methods	void assemble (boolean createListing)
Error report	Reports and records all types of errors with the position of the error and its type.	Classes	ErrorReporter.java, Error.java
Resolve	Assigning a resolved value to Labels and inserting them into a Label symbol table.	Method	boolean resolve(int currentAddr, SymbolTable<Integer> labelTable)
Read	Reads the assembly file character by character and returns a character until EOF is reached. Returns a null character if EOF reached.	Method	char read()
Parse	Parses lineStatements objects into Instructions objects and adds the former into the assemblyUnit object.	Method	AssemblyUnit parse()

Table 2: Important Actions

3.0 KEY CONCEPT MODEL

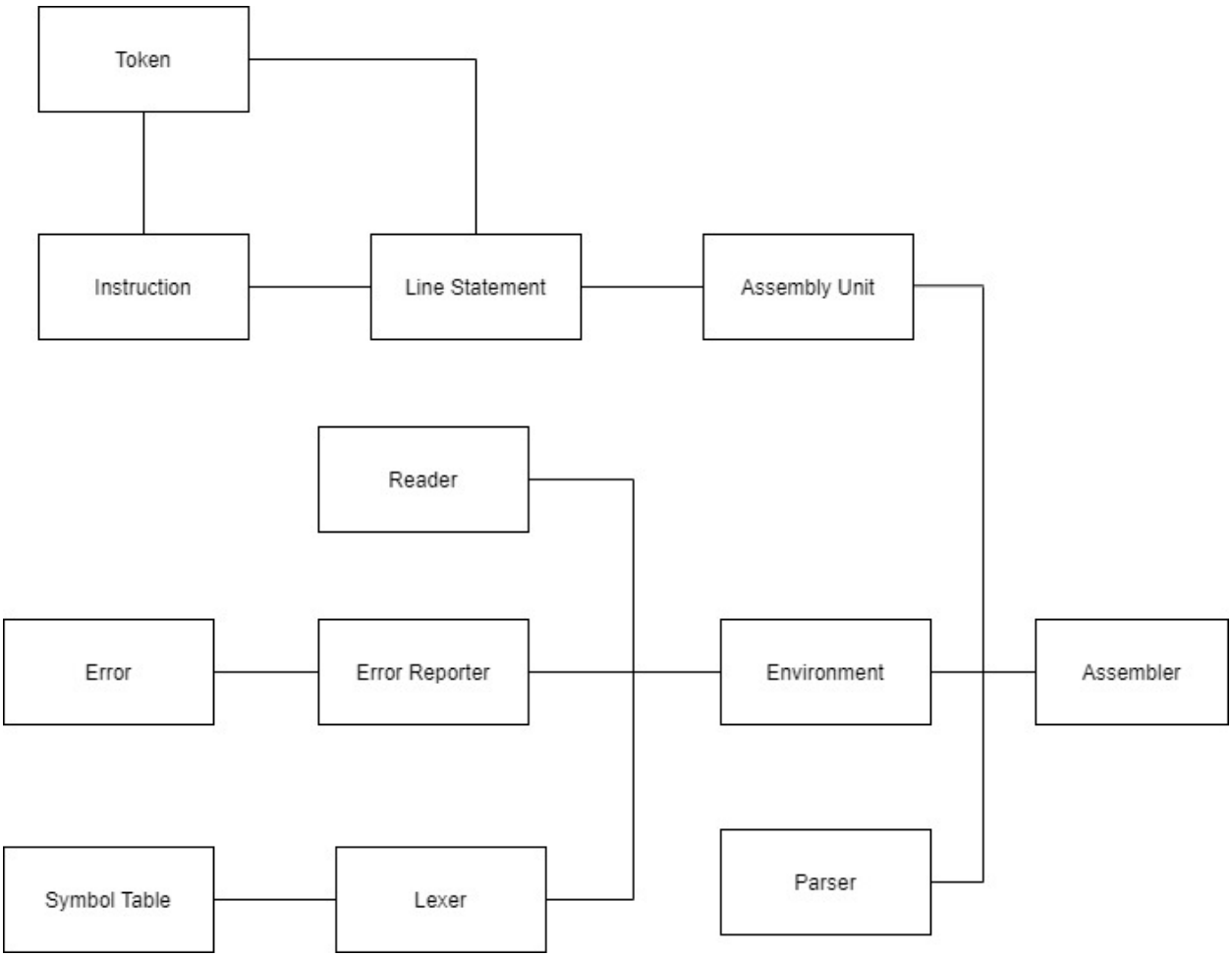


Figure 1: Key Concept Model of the Problem Domain

4.0 USE CASE DIAGRAM

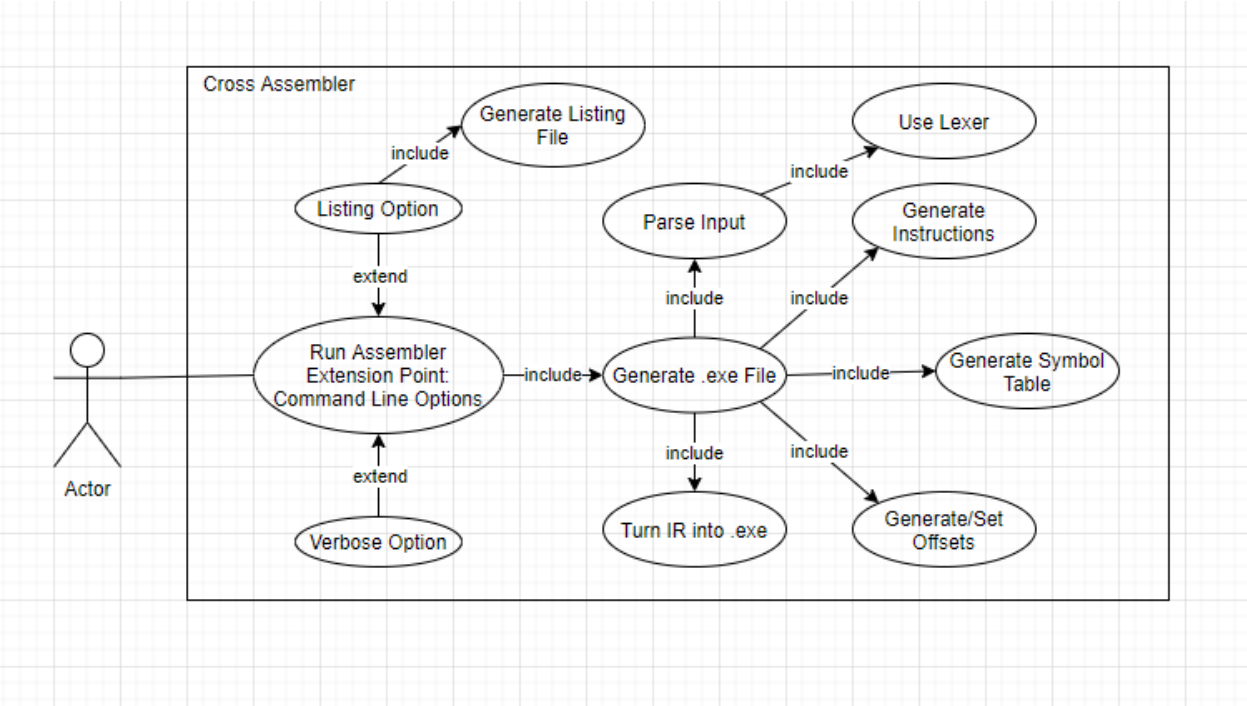


Figure 2: Use Case Diagram of the Cross Assembler.

5.0 CLASS DIAGRAM

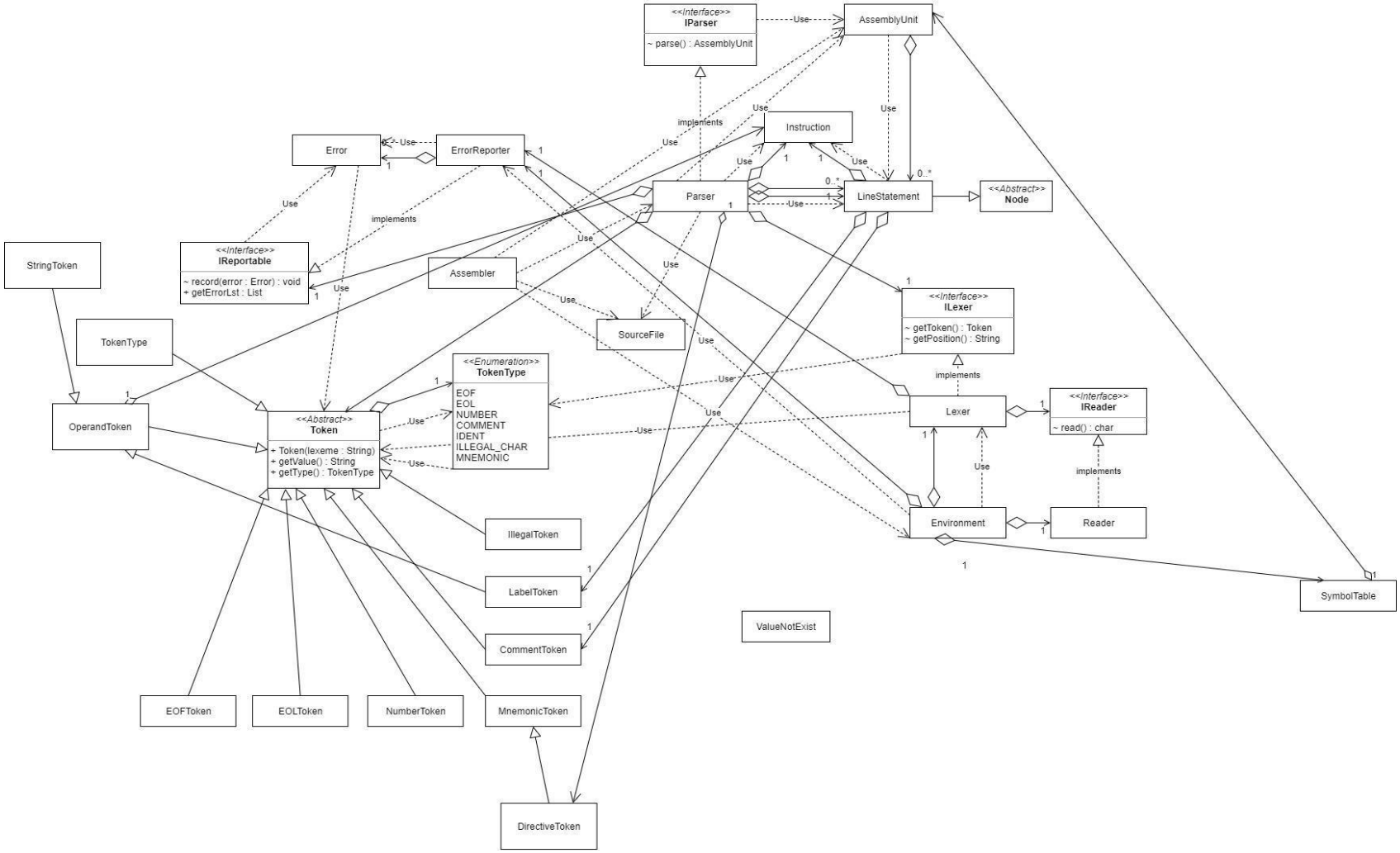
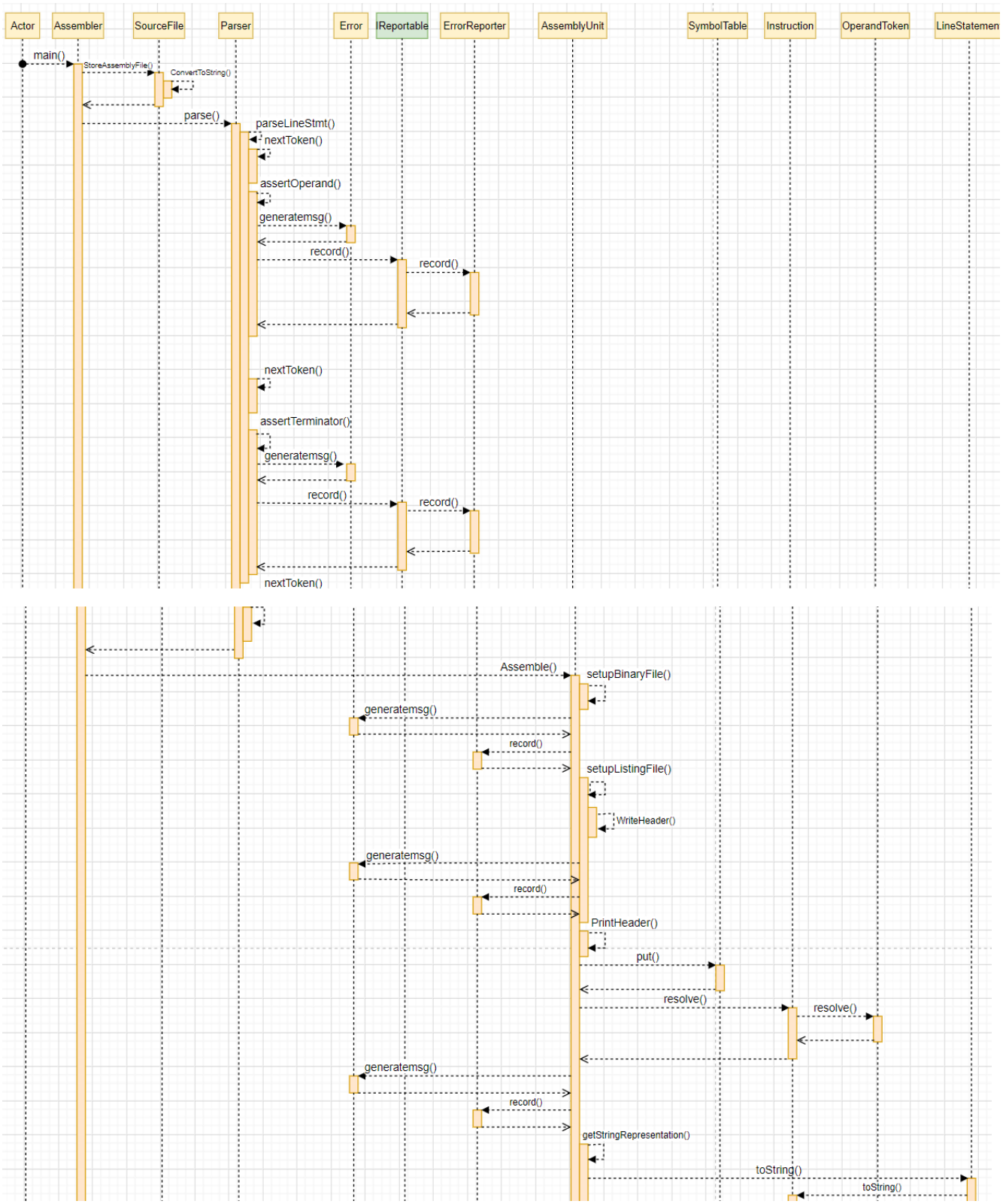


Figure 3: UML Class Diagram

6.0 SEQUENCE DIAGRAM



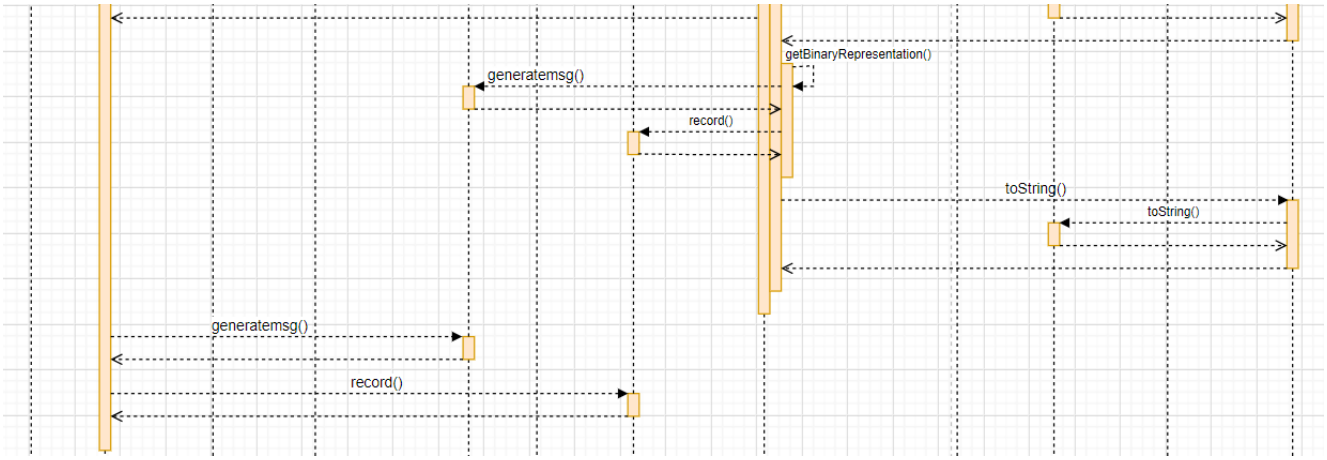


Figure 4: Sequence Diagram

7.0 EXPLANATION OF DESIGN PATTERNS

7.1 Strategy Design Pattern

The strategy design pattern is used to resolve operands through the first pass completed by the assembler. An abstract OperandToken class has an abstract resolve() method that must be defined by the subset of possible operands that may exist. Operands can either be labels (strings) or numeric values (integers). As such, there exist two subclasses, StringToken and NumberToken that implement the OperandToken abstract class. They both define their own logic to resolve their operands. Given that both of these subclasses invoke the resolve() method under the umbrella of the parent OperandToken class, this approach can be categorized as an implementation of the strategy design pattern.

7.2 Factory Design Pattern

The factory design pattern is used to create tokens for the components of line statements that have been analyzed by the lexer. For each token in the assembly file, the Parser class calls the getToken() method from the Lexer class. This parses a sequence of characters and creates a token based on what is read from the assembly file. The created token is then inserted into the AssemblyUnit class, which is the intermediate representation of the assembly file that is used to generate the listing and executable files. This automatic token creation can therefore be categorized as an implementation of the factory design pattern.

7.3 Composite Design Pattern

The composite design pattern is implemented in the parser. A java List of line statements is created and stores the various types of line statements that may possibly arise within an assembly file. The aggregation of these line statements make up the assembly unit, otherwise known as the intermediate representation of the assembly file. The criteria needed for the composite pattern are satisfied because of the various types of line statements that are held by the assembly unit. Line statements are composed of any combination of the following : labels, mnemonics, operands (string or numeric), and comments. Given that they can all be accessed by iterating through the assembly unit list, the assembly unit can be categorized as the implementation of the composite design pattern.

8.0 SPRINT PLANNING

The following are three tables showing the story points allocated to the various user stories provided in the backlogs for each sprint.

8.1 Sprint 1

Spring Backlog Item	Story Points / Velocity
Generate and print an assembly listing	2
parse an AssemblyUnit	8
parse a LineStatement	4
parse an Instruction	2
parse an InherentInstruction	2
create a Node	1
extract tokens with a Lexer	4
enter and lookup nodes in SymbolTable	1
record and report errors in ErrorReporter	2
keep line number and column number in Position	1
save error messages and position in ErrorMessage	1
keep filename and extension in SourceFile	1

Table 3: Sprint 1 Backlog

8.2 Sprint 2

Spring Backlog Item	Story Points / Velocity
Generate a binary executable file	4
Scan a number	1
Scan a comment	1
Parse an ImmediateInstruction	2

Table 4: Sprint 2 backlog

8.3 Sprint 3

Spring Backlog Item	Story Points / Velocity
Scan a directive	1
Scan a string	2
Parse a relative instruction	4
Parse a directive	2

Table 5: Sprint 3 backlog

9.0 CONCLUSION

The project produced a cross-assembler in the java programming language. The team of seven individuals implemented a scrum, sprint-based methodology to complete the project. There were four sprints and a final report to be submitted, and each sprint required that a fully-functioning deliverable be demonstrated with the included product backlog items that were specified during sprint planning.

The first sprint was where the greatest bulk of the work occurred, and a solid base for the assembler was created. A working model of the assembler was generated that was capable of handling an assembly file composed solely of mnemonics (referred to as inherent instructions). A lexer reads the mnemonics from the assembly file character-by-character, checks them against a symbol table of valid mnemonics, stores them as tokens, and then inserts them into the line statement objects that they belong to. Then, a parser creates an in-memory representation (IR) of the assembly file, which is referred to as the assembly unit. The assembly unit is then read line-by-line and token-by-token and converted into a listing file. Additionally, an error reporting mechanism was put into place to handle parsing errors.

The second sprint expands the type of tokens that the lexer is able to handle. The assembly file now includes labels, immediate operands, and comments that could now be stored as tokens as well, and inserted into their respective line statement objects. The immediate operands are numeric values of size 3 or 5 bits that are stored within the 8 bit mnemonic. These tokens are added to the output of the listing file, and an executable file with the equivalent information is also generated for the first time.

The third sprint includes functionality that allows the assembler to handle relative instructions in addition to the inherent and immediate ones. This means that operand tokens must now be lexed, and could potentially be labels or numerical values. Additionally, directives are also handled, meaning that string tokens have been added to the list of possible outputs generated by the lexer. Of course, the corresponding listing and executable files also correctly reflect the new types of information that could be found in the assembly file.

The first three sprints implemented all of the product backlog items that were requested by the product owner. As such, there was no new functionality to implement in sprint 4. Instead, more emphasis was placed on adding javadoc comments, reformatting the program files, and ensuring that the functionality was made as clear and legible as possible. Additionally, a good deal of refactoring was done to optimize the code and ensure that all possible error cases were handled.

Overall, the project was a success and our team managed to create a fully-functioning cross-assembler that implemented all of the product backlog items. The demos were well-received and they always met the criteria specified by the product owner for each given sprint.

10.0 APPENDIX: LISTING OF JAVA SOURCE FILES

```
├──main
│   ├──java
│   │   └──co
│   │       └──tdude
│   │           └──soen341
│   │               └──projectb
│   │                   │  Dump.class
│   │                   │
│   │                   ├──Assembler
│   │                   │   │  Assembler.java
│   │                   │   │  AssemblyUnit.java
│   │                   │   │  SourceFile.java
│   │                   │   │
│   │                   │   └──AssemblyParser
│   │                   │          IParser.java
│   │                   │          Parser.java
│   │                   │
│   │                   ├──Environment
│   │                   │      Environment.java
│   │                   │
│   │                   ├──ErrorReporter
│   │                   │      Error.java
│   │                   │      ErrorReporter.java
│   │                   │      IReportable.java
│   │                   │      Position.java
│   │                   │
│   │                   ├──Lexer
│   │                   │   │  ILexer.java
│   │                   │   │  Lexer.java
│   │                   │   │
│   │                   │   └──Tokens
│   │                   │          CommentToken.java
│   │                   │          DirectiveToken.java
│   │                   │          EOFToken.java
│   │                   │          EOLToken.java
│   │                   │          IllegalToken.java
│   │                   │          LabelToken.java
│   │                   │          MnemonicToken.java
│   │                   │          NumberToken.java
│   │                   │          OperandToken.java
│   │                   │          StringToken.java
```

```
│   │               │       Token.java
│   │               │       TokenType.java
│   │               │
│   │               ├──Node
│   │               │      Instruction.java
│   │               │      LineStatement.java
│   │               │      Node.java
│   │               │
│   │               ├──Reader
│   │               │      IReader.java
│   │               │      Reader.java
│   │               │
│   │               └──SymbolTable
│   │                      SymbolTable.java
│   │                      ValueNotExist.java
│   │
│   └──resources
│          logging.properties
│
└──test
    ├──java
    │   └──co
    │       └──tdude
    │           └──soen341
    │               └──projectb
    │                   └──test
    │                          AssemblerTest.java
    │
    └──resources
           asm_binary
           asm_input.asm
           asm_output.lst
```