

# CSCI 468 Portfolio Report

*Steven Vought, Wesley Smith*

## **Section 1: Program**

The source is included in the file source.zip in this directory. A link to the source folder, `/capstone/portfolio/source.zip`.

## **Section 2: Teamwork**

At the beginning of this project we were instructed to form pairs of team members as we would be working on this capstone project together with a teammate. I partnered up with Wesley Smith, a good friend of mine also in the CS program whom I have worked together with for other class projects during my time here at Montana State University. Since we have worked together in the past, we already have a solid understanding of each other's strengths and weaknesses when it comes to coding projects.

Originally we had planned to setup a single repository for both of us to work on together, but our professor decided early on in the semester that we would instead have private repositories for each student, and collaborate with shared code through Github rather than a single code base to work with. We constructed our separate project repositories and would share

code with each other as we completed portions of the project for each checkpoint. We each created documentation and tests for the others project.

For us, it was easier to meet up on video conferences set up through Discord or meet up physically to work together on our code base. As professor Carson was the one who wrote the backbone of our coding project and helped us through a few sections of the compiler coding process, I will also be listing him as one of our partners in our contributions calculations. Below is a general breakdown of our team member contribution for this project:

**Total estimated hours:** 150

**Team Member 1:**

**Contributions:** Built own working Catscript program and assisted team member 2 in building his their version of the program. Created tests and documentation for team member 2.

**Estimated Work Hours:** 85 hours, approx. 57% time contribution

**Team Member 2:**

**Contributions:** Built own working Catscript program and assisted team member 1 in building their own version of the program. Created tests and documentation for team member 1.

**Estimated Work Hours:** 65 hours, approx. 43% time contribution

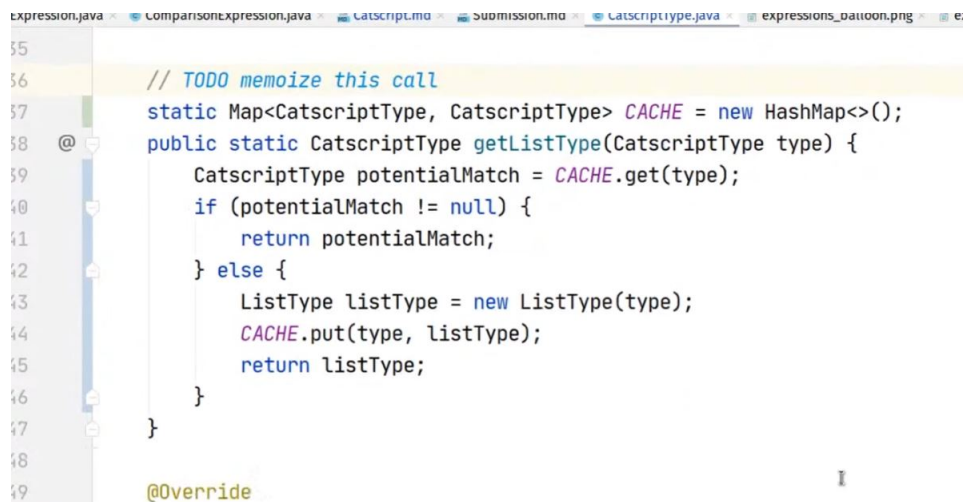
**Team Member 3:** Carson Gross

**Contributions:** Help sessions and explanation videos from class

**Estimated Work Hours:** 30 hours of helping our projects, approx. 20% time contribution

## Section 3: Design Pattern

For our project the design pattern the class had decided to go with was the memorization pattern. This design pattern allows for the conservation of algorithm speed by saving “time-consuming” processes performed in our code and referencing the results of those calculations later if the same inputs are put in again. We accomplished this by using a HashMap to store the Catscript Types as they are decided in the code and would reference the type assigned to the input if it matched a previously used input.



```
expression.java  ComparisonExpression.java  Catscript.md  Submission.md  CatscriptType.java  expressions_oatlooh.png  e:
55
56 // TODO memoize this call
57 static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
58 @ public static CatscriptType getListType(CatscriptType type) {
59     CatscriptType potentialMatch = CACHE.get(type);
60     if (potentialMatch != null) {
61         return potentialMatch;
62     } else {
63         ListType listType = new ListType(type);
64         CACHE.put(type, listType);
65         return listType;
66     }
67 }
68
69 @Override
```

## **Section 4: Technical Writing**

### ***1 Introduction***

This report will cover the creation of a Catscript compiler that was created in class. The purpose of creating this compiler is to combine all the skills from previous computer science classes as well as introduce us to the inner workings of a compiler. This project also gave us exposure to a team project in the computer science field. The sections of this paper are as follows:

- 1) Introduction*
- 2) Background*
- 3) Methods and Discussion*
- 4) Conclusion and Future Work*

### ***2 Background***

A compiler translates code from a high-level language into machine-code in order for it to be executed on a computer. Compilers allow programmers to write something that is readable and understandable to the human eye while still creating something that is understandable to a computer. The communication between the readable code and machine code is important as it is what decides how efficiently the program can be debugged or further extended.

A standard compiler starts with a Tokenizer, which runs through every character of the program and converts them into a list of tokens. Next step in creating a compiler is the Parser. This is where the tokens created from the tokenizer come in handy as the compiler has no way of interpreting the program at this point. The parser takes a list of tokens as input, then parses them based on a series of context-free grammars. The CFG breaks down the given list of tokens into valid program pieces, creating a tree filled with different node defined by rules of the CFG. If the program is invalid, it would fail in this step and errors would be given. If the program is valid, it will create a parse tree filled with different tokens from the tokenier.

### ***3 Methods and Discussion***

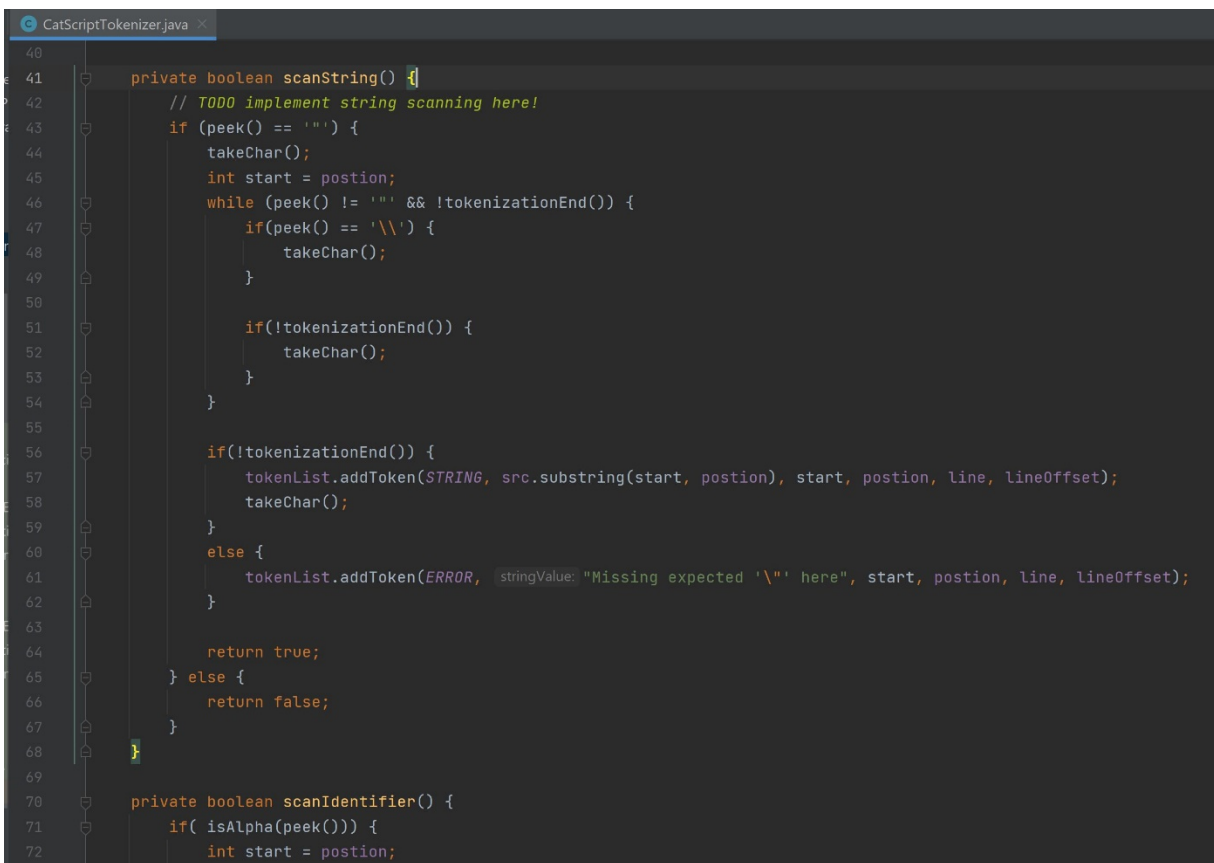
#### **3.1 Tools and Setup**

We used a GitHub repository to receive updates on the project from the professor and to push our current version as a submission for checkpoints throughout the semester. My team used IntelliJ for the development process which interacts directly with GitHub which made pushing and pulling an easier process. This was the first I had ever used GitHub in any capacity and was an extra step of learning.

For team management and group meetings, we used Discord. This was due to the Covid 19 pandemic causing us to be in distanced learning, Discord allowed us to meet up at any time that was convenient for everyone and discuss what we where working on and what we were having issues with.

## 3.2 Tokenizer

The tokenizer is a simple and powerful tool that is the first step in program compilation. The tokenizer runs through the entire input and breaks it down into tokens, a list of tokens may look like this, [“”], “(“, “,” “A”], then outputs the list of tokens. Even though this is a simple step in creating a compiler, there are complications in the process, including output tokens not being legal expression in the given language. Below is snippet of tokenizer where we scanned a string.



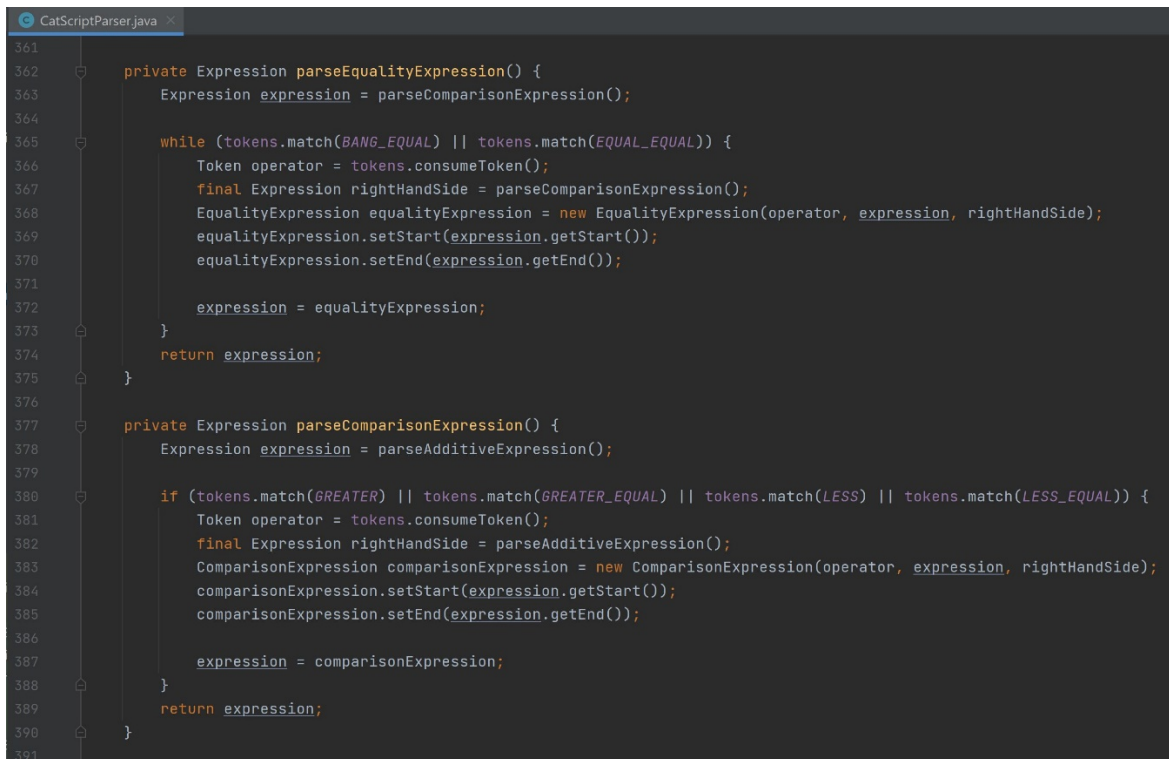
```
40
41 private boolean scanString() {
42     // TODO implement string scanning here!
43     if (peek() == '"') {
44         takeChar();
45         int start = postion;
46         while (peek() != '"' && !tokenizationEnd()) {
47             if(peek() == '\\') {
48                 takeChar();
49             }
50
51             if(!tokenizationEnd()) {
52                 takeChar();
53             }
54         }
55
56         if(!tokenizationEnd()) {
57             tokenList.addToken(STRING, src.substring(start, postion), start, postion, line, lineOffset);
58             takeChar();
59         }
60         else {
61             tokenList.addToken(ERROR, stringValue: "Missing expected '\"' here", start, postion, line, lineOffset);
62         }
63
64         return true;
65     } else {
66         return false;
67     }
68 }
69
70 private boolean scanIdentifier() {
71     if( isAlpha(peek())) {
72         int start = postion;
```

### 3.3 Parser

Second step in creating the compiler was the parser, this is where the recursive nature of our algorithm made things a little easier to understand. Our program originally just had an Expression and we had to implement, based on the given grammar, the different types of possible expressions and statements that would be acceptable. These included unary expressions, additive expressions, comparison expressions, for statement, if statement, and many more.

When trying to parse an expression, the compiler we wrote would call `parseEqualityExpression()` which then works its way through `comparisonExpression`, `AdditiveExpression`, and so on. In the case of “1+1” we would end up in `parseAdditiveExpression()`, which then would call `parseEqualityExpression`, creating the recursive nature. This will continue to happen until everything has been parsed successfully or an error has been thrown. Below is a snippet of

the `parseEqualityExpression` and `parseComparisonExpression` methods.



```
361
362 private Expression parseEqualityExpression() {
363     Expression expression = parseComparisonExpression();
364
365     while (tokens.match(BANG_EQUAL) || tokens.match(EQUAL_EQUAL)) {
366         Token operator = tokens.consumeToken();
367         final Expression rightHandSide = parseComparisonExpression();
368         EqualityExpression equalityExpression = new EqualityExpression(operator, expression, rightHandSide);
369         equalityExpression.setStart(expression.getStart());
370         equalityExpression.setEnd(expression.getEnd());
371
372         expression = equalityExpression;
373     }
374     return expression;
375 }
376
377 private Expression parseComparisonExpression() {
378     Expression expression = parseAdditiveExpression();
379
380     if (tokens.match(GREATER) || tokens.match(GREATER_EQUAL) || tokens.match(LESS) || tokens.match(LESS_EQUAL)) {
381         Token operator = tokens.consumeToken();
382         final Expression rightHandSide = parseAdditiveExpression();
383         ComparisonExpression comparisonExpression = new ComparisonExpression(operator, expression, rightHandSide);
384         comparisonExpression.setStart(expression.getStart());
385         comparisonExpression.setEnd(expression.getEnd());
386
387         expression = comparisonExpression;
388     }
389     return expression;
390 }
391
```

The parser creates a parse-tree using top-down, where the parser start at the root and building downward, and uses Left-most-Derivation. This means the parser works from the left of the input to the right looking one token ahead to determine which production to use.

Creating the parser was a time-consuming effort with many learning experiences. The opportunity to get to use recursion in a project showed how useful an efficient recursive code can be. As this was a Test Driven Development (TDD) project, most of our difficulties were in trying to understand what the error messages were and how to go about fixing them. For this, my team used the debugging tool in IntelliJ frequently, which allowed us to see what the program was doing/storing at certain points, known as break points, and with that information we could typically determine where and why an error was occurring.

### **3.4 Eval**

The next step was evaluation, where the compiler evaluates a given expression, example:

```
var x=5+5  
print(x)
```

in this example the value 10 would be printed to the screen. Three pieces have to evaluate correctly for this to happen, print statements, variable statements, and additive statements. The additive statement evaluates the “5+5” portion and gives 10, the variable statements creates a new identifier “x” in the symbol table and assigns the value (10 in this case) to x. The print statement was a little tricky to get working at first as you don’t have to evaluate the print statement itself but rather the expression within it. For each expression and statement in our grammar we had to create an eval so that it was evaluated correctly during runtime. Below is a



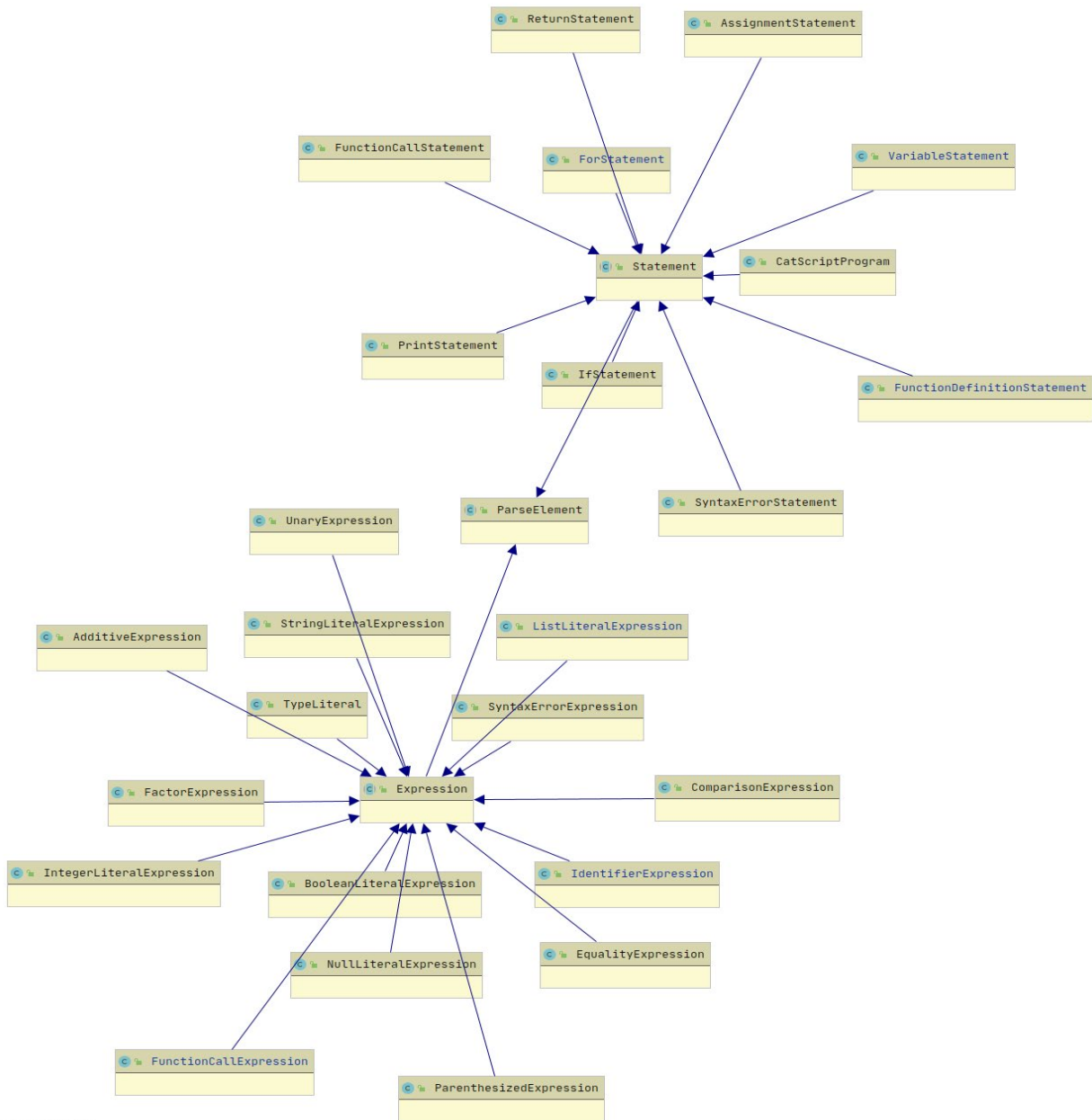
snippet of the evaluation code for an additive expression, this was also a challenge due to having to take into account integers as well as strings.

```
AdditiveExpression.java x
65 //=====
66
67 @Override
68 public Object evaluate(CatscriptRuntime runtime) {
69     if(getType().equals(CatscriptType.STRING) && isAdd()) {
70         String lhsValue;
71         String rhsValue;
72         if (leftHandSide.getType().equals(CatscriptType.NULL)) {
73             lhsValue = leftHandSide.getType().toString();
74             rhsValue = rightHandSide.evaluate(runtime).toString();
75         } else if (rightHandSide.getType().equals(CatscriptType.NULL)) {
76             lhsValue = leftHandSide.evaluate(runtime).toString();
77             rhsValue = rightHandSide.getType().toString();
78         }
79         else {
80             lhsValue = leftHandSide.evaluate(runtime).toString();
81             rhsValue = rightHandSide.evaluate(runtime).toString();
82         }
83         return lhsValue + rhsValue;
84     }
85     else {
86         Integer lhsIntValue = (Integer) leftHandSide.evaluate(runtime);
87         Integer rhsIntValue = (Integer) rightHandSide.evaluate(runtime);
88         if (isAdd()) {
89             return lhsIntValue + rhsIntValue;
90         } else {
91             return lhsIntValue - rhsIntValue;
92         }
93     }
94 }
95 }
```

## ***Conclusion and Future Work***

This implementation is not the most efficient possible implementation as we did not make any attempts in optimization. Future work would include optimization once the rest of the given tests in the TDD project were complete. This project gave us a great insight in how compilers work, an introduction to Test Driven Development, and a large codebase team project. This project gave us a better appreciation for the amount of time and work it must have taken to create languages like Java, C, and Python.

## Section 5: UML



## Section 6: Design Trade-offs

We learned and implemented the recursive descent algorithm for creating parsers. This algorithm had us create a function for each production in the grammar of the language, these functions are then recursively called as needed throughout the program to parse the given input. For example, in the following figure, a parenthesized expression, (2+5), would be created by starting in the “expression” production and working down to the “primary\_expression” production, this gives the necessary parenthesis, then the “expression” production is called again. This shows the recursive nature of the algorithm when trying to parse an input of something like “(((2+4)+3)-5)”.

```
parameter_list = [ parameter, {',', ' parameter' } ];  
parameter = IDENTIFIER [ , ':', type_expression ];  
return_statement = 'return' [ , expression ];  
expression = equality_expression;  
equality_expression = comparison_expression { ("!=" | "==") comparison_expression };  
comparison_expression = additive_expression { (">" | ">=" | "<" | "<=" ) additive_expression };  
additive_expression = factor_expression { ("+" | "-" ) factor_expression };  
factor_expression = unary_expression { ("/" | "*" ) unary_expression };  
unary_expression = ( "not" | "-" ) unary_expression | primary_expression;  
primary_expression = IDENTIFIER | STRING | INTEGER | "true" | "false" | "null" |  
    list_literal | function_call | "(" , expression, ")"  
list_literal = '[', expression, { ',', expression } ']';  
function_call = IDENTIFIER, '(', argument_list , ')'  
argument_list = [ expression , { ',', expression } ]
```

The benefits to using the recursive descent algorithm include how clearly it demonstrates the recursive nature of parsers, how practical it is to use, and how widely its used in the industry. There are some trade-offs with this design choice as well however when compared to using a parser generator instead. These downsides include having to create more infrastructure for the program and having to write more code to accomplish same tasks.

## **Section 7: Software development life cycle mode**

This project was built using the Test-Driven Development (TDD) model, where the professor built the bulk of project and all the classes we would need, then created a number of tests that we had to get working. These tests drove us to complete parts of the program, which in turn required us to create methods and fields in certain areas to get the program working as intended.

This was the first time I had been introduced to the TDD method and I enjoyed working in it. There were issues however where I feel this method kept my team back, mainly when a given error occurred when running tests we would not know what function or class was causing the error, this issue got better as time went on and we got more familiar with the project and what was entailed. I feel this method was great in teaching us how to work in a large code base project that you didn't necessarily start on, what I think working on a legacy code project may be like. The tests also gave my team a great indication of what we should be working on, which kept us focused on the project and writing code rather than trying to figure out what was needed to be don't next.