CSCI 468 Portfolio

Spring 2021

Lin Shi, Bowen Kruse

Section 1: Program

See source.zip file in the same directory (/capstone/portfolio).

https://github.com/LinShi11/csci-468-spring2021-private/blob/main/capstone/portfolio/source.zip

Section 2: Teamwork

Overview:

The project was completed in a group of two and it has been a semester long task. Throughout the semester, team member one was responsible to complete all the code and design the parser, while team member two was responsible to write the tests for the program as well as write a documentation for the program. The tests were implemented using patch and was passed to the other individual by email. The other communicates between the partners were completed using discord. Additionally, the partners had the access to the each other's GitHub repository for comparison and discussion; however, each one is responsible to complete their own program.

Member Contribution:

Team Member one:

- Primary programmer
- Provide tests for Team Member two for their code
- Provide documentation for Team Member two
- Estimated hours of contribution: about 200 hours

Team Member Two:

- Provide tests for Team Member one
- Provide documentation for Team Member One
- Estimated hours of contribution: about 10 hours

Section 3: Design pattern

The design pattern implemented in the code was memorization. When there is some expensive algorithm implemented and the result could be reused, it is much efficient to memorize the result and store it somewhere. This is known as caching. It is commonly used to avoid the unnecessary code when the data has could be reused.

Within our code, memorization was used in CatScriptType. More specifically, we used memorization in the getListType function. The function's task is to find the type of the list that we are dealing with. Therefore, the CatScriptType system will return a type of based on listType, which extends from CatScriptType. Just like the example below:

```
public static CatscriptType getListType(CatscriptType type) {
    ListType listType = new ListType(type);
    return listType;
}
```

However, instead of just finding the new listType every time the function is called, we will use the memorization pattern. First, a static hash-map is created to hold the information. Within the function, the previous type is found. If the instance is null, that means this is the first time the function has been ran, the listType will then be calculated, added to the CACHE and returned If the instance of CACHE is not null, the previous match will be returned. Example below:

```
static Map<CatscriptType, CatscriptType> CACHE = new HashMap<>();
public static CatscriptType getListType(CatscriptType type) {
    CatscriptType previousMatch = CACHE.get(type);
    if(previousMatch == null){
        ListType listType = new ListType(type);
        CACHE.put(type, listType);
        return listType;
    } else{
        return previousMatch;
    }
}
```

By accomplishing the memorization design pattern, computation time will be saved if the method is called many times through the execution process. Although in this case, the algorithm is not extremely expensive, it is a good skill to have and it is a great practice of the memorization design pattern.

Section 4: Technical writing

Overview:

The CatScript programming language was created as a learning tool for CSCI 468. Throughout the course, we discussed ideas such as tokenizer, parser, and code generator. In the beginning of the course, we were provided with the essential skeleton code that had the tools used for debug as well as the essential code to run the program, both as a local server and when we did not have every aspect of the program completed. Additionally, Java was chosen to be used as the language since Java had a good String class built into it.

CatScript Types:

The CatScript language has a small type system, which contains the following:

```
int - a 32 bit integer
string - a java-style string
bool - a boolean value
list - a list of value with the type 'x'
null - the null type
object - any type of value
```

As previously mentioned, one of the main reasons to choose Java was the Java-styled string that have been implemented in our program. The other types are integers, boolean, list, null, and object.

CatScript Grammar:

The CatScript language is a context-free grammar, which is a formal grammar with a structure that is set by its recursive rules. The grammar is a set of production rules that will describe all possible string for the language. The production contains the head and the body. The head contains all the non-terminals and the body could have both the non-terminals and the terminals. Additionally, the language has been written in Backus-Narur form and the non-terminals are in lowercase while the terminals are either in quotes or capitalized. Below is the CatScript Grammar. It is split into the statement section and expression section:

Expression Grammar:

```
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 } ]
```

Expression overview:

The expression grammar follows a recursion where equality expression is called first, followed by comparison expression, additive expression, factor expression, unary expression, and primary expression in that order. This has been done because of the priorities. For example, multiplication and division has a higher priority than addition and subtraction; therefore, it needs to be further down the parse tree to accomplish that. Thus, down in the bottom of the recursion tree, the primary expressions, such as string and integer, will be called first. Then the unary expression, the negative and “not” will be looked at. Next, it is the factor expression. Followed by the additive expression and comparison expression. Lastly, it is the equality expression. Additionally, the list literal and function call are part of the primary expressions and will be identified at the lowest level.

Equality Expression:

The equality expression is used to compare two expression to determine whether they are equal or not. CatScript use “==” operator for equal and “!=” for not equal. In the end, the equality expression will return a true or false based on its evaluation. The equality expression consist an expression, operator ("==" | "!="), and an expression. For example:

```
1 == 1 // evaluates to true
1 != 1 // evaluates to false
1 != 2 // evaluates to true
1 == 2 // evaluates to false
```

Comparison Expression:

Comparison expression is used to determine the whether the numeric relation (<=, <, >, >=) between the two sides is true or not. Again, the comparison expression will have a left-hand side and a right-hand side dividing among the symbol. Both the left-hand side and the right-hand side will recursively iterate down the parsing tree. In the end, the expression will be evaluated and a boolean will be returned. The comparison expression will contain expression, operator ("<=" | "<" | ">" | ">="), and an expression. Below is a quick example:

```
1 > 10 // evaluates to false
1 >= 10 // evaluates to false
1 < 10 // evaluates to true
1 <= 10 // evaluates to true
```

Additive Expression:

The additive expression will have a left-hand side and a right-hand side, both will follow the recursive parsing tree and call factor expression. The subtraction (-) will take the two integers and find its value. On the other hand, addition (+) is a little different. If there are two integers, it will add them up and find its value. However, if at least one side is a string, the plus operator will be a string concatenation and return a string. Additive expression will consist expression, operator ("+" | "-"), and expression. Such as:

```
1 - 1 // evaluates to 0
1 + 1 // evaluates to 2
1 + "Hello" // evaluates to "1HELLO"
"Hello" + "World" // evaluates to "HelloWorld"
```

Factor Expression:

In a factor expression, it will contain the left-hand side and right-hand side. Both sides will continue to call unary expression. The factor expression will take both sides, complete the operator, "/" for division and "*" for multiplication, and return the result. Once again, factor expression will have expression, operator ("/" | "*"), and expression.

```
4 * 2 // evaluates to 8
4 / 2 // evaluates to 2
```

Unary Expression:

The unary expression is different than all the expression above, it contains at least one unary operator ("not" | "-") and some primary expression. Since there is not a left-hand side, if the unary operator finds the operator, it will continue to iterate through it until it finds a primary expression, which will be discussed in the next section. If the primary expression is an integer, the unary expression will flip the sign of the integer. If primary expression is a boolean, it will find the opposite of the current value. Unary expression will have at least one ("not" | "-") and followed by the primary expression.

```
    not true // evaluates to false
    - 1 // evaluates to -1
```

Primary Expression:

Primary expression is the last one on the recursive list. It could have identifier, a string, an integer, a boolean, a list literal, a function call, or parentheses with an expression in it. The identifier will be some variable such as x. String will be a string of, for example, "x". Integer will be some numeric number. Boolean consist of true, false, or null. These three and the parentheses are the only terminals in the primary expression and the others are non-terminals. List literal and function call will be discussed next. Expression is just calling the parseExpression function above, where it will iterate through and the parse tree again.

```
    x // evaluates to identifier
    "x" // evaluates to string
    1 // evaluates to integer
    (<some kind of expression here>) // evaluates to parenthese and call
expression
```

List Literal:

The list literals are identified in the primary expression with a left bracket. Once a left bracket is identified in the primary expression, it will continue to iterate through for different expressions. Additionally, there will be commas within the list literals to separate all the different expressions.

```
    [1, 1+1, 2, 3] //evaluates to 1, 1+1, 2, 3
```

Function Call:

Similar to List Literals, function call is determined within primary expressions. Function call contains an identifier, followed by the parenthesis and argument lists. Therefore, within the primary expressions, once the identifier has been found, the next token will be looked at to determine whether it is just an identifier or a function call.

```
    foo(1, 2) // evaluates to function call, function foo
```

Argument List:

Argument list is called in function call expression where it is almost the same as list literals without the brackets. It contains the expressions with comma separating them.

```
    foo(1, 2) //evaluates to function call, argument list is 1 and 2
```

Expression Conclusion:

Based on the grammar, the expression will be running as recursion. The bottom of the parse tree is the primary expression, which consist of the different terminals such as integer value, string value, boolean value, parenthesis, etc.

Statement Grammar:

```
catscript_program = { program_statement };

program_statement = statement |
                    function_declaration;

statement = for_statement |
            if_statement |
            print_statement |
            variable_statement |
            assignment_statement |
            function_call_statement;

for_statement = 'for', '(', IDENTIFIER, 'in', expression ')',
                '{', { statement }, '}';

if_statement = 'if', '(', expression, ')', '{',
                    { statement },
               '}' [ 'else', ( if_statement | '{', { statement }, '}' ) ];

print_statement = 'print', '(', expression, ')'

variable_statement = 'var', IDENTIFIER,
     [':', type_expression, ] '=', expression;
```

```
function_call_statement = function_call;

assignment_statement = IDENTIFIER, '=', expression;

function_declaration = 'function', IDENTIFIER, '(', parameter_list, ')' +
                       [ ':' + type_expression ], '{',
{ function_body_statement },  '}';

function_body_statement = statement |
                          return_statement;

parameter_list = [ parameter, {',' parameter } ];

parameter = IDENTIFIER [ , ':', type_expression ];

return_statement = 'return' [, expression];

type_expression = 'int' | 'string' | 'bool' | 'object' | 'list' [, '<' ,
type_expression, '>']
```

Statement Overview:

The grammar for program statement consists of two different sections, the statement and function declaration. The statement consists of for, if, print, variable, assignment, and function call statement. Based on the grammar, the statement is not called recursively. Instead, the statements are called based on the first token. Additionally, if none of the previous statements satisfies, the function declaration will be used. In terms of priorities, all the statements will take the same priorities after the expression has been evaluated. If none of the statements has been true, the function declaration will then be tested.

For Statement:

For statement will be choose after the first token has been analyzed. Since the first keyword is "for", the for statement will be looked at. Then it consists of the parenthesis, identifier (the variable used in the for loop), keyword "in", the expression, and end parenthesis. Next, there is the body of the for statement, starting with the left curly brace, then the body will be parsed as statements, followed by the closing curly brace.

```
    for (x in [1,2,3]){
        print(x)
    }
    // this will evaluates to printing 1 2 3 each time
```

If Statement:

If statement will be identified by the "if" keyword. Next, the expression is within the parenthesis, which mean the expression will be parsed. This will be a boolean result that will determine whether to continue within the if statement. However, there could be the else and the else if statements. Therefore, the "else" keyword will be analyzed and the token after that will be looked at to see if the next token is the "if" token. If there is the else if token, another if statement will be parsed. Which means the if statement will parse recursively if there is an else if present. If there is only an else token, the statement will be parsed and that is the end of the if statement.

```
var x = 2
if(x == 1){
    print(x)
} else if(x == 2){
    print("else if" + x)
} else{
    print(x)
}
// this will evaluates to "else if2"
```

Print Statement:

The print statement will be identified by the "print" keyword. Then, it will be an expression within the parenthesis. Which means it will parse the expression as body.

```
print("x" + 1)
// this will evaluates to x1
```

Variable Statement:

The variable statement starts with the key word "var". Then the variable name, identifier, will be parsed. Next, there could be the type expression, which will be talked about next. Within the variable statement, it is the "=" and the expression. This will mean the variable name is set equal to the expression.

```
var x = 1+2
// x will be evaluated to 3
```

Type Expression:

The type expressions are used in multiple spots. It consists of a ":" followed by the type expression that was talked about above. These are used to determine the type of the variable or return type of a function.

```
var x : string = "1"
// x will be a string with the value of 1
```

Function Call Statement:

The function call statement only consists of the function call expression. However, both the function call statement and the assignment statement, which will be looked at next, starts with identifier token. Once function call statement is decided, it will follow the function call expression.

```
foo()
// evaluates to function call statement for function foo
```

Assignment Statement:

The assignment statement starts with an identifier, just as function call. However, there is a "="; therefore, once identifier is found, the next token is analyzed to determine whether it is an assignment statement or a function call statement. Once the assignment statement is determined, it is going to be followed by the expression.

```
x = 1 + 2
// x evaluates to 3
```

Function Declaration Statement:

Once all the statement has been analyzed, the function declaration will be looked at next. Function declaration starts with keyword "function" and identifier, the function name. Next there is a parameter list in parenthesis, parameter list will be talked about next. Then, the function declaration could have a type expression that will determine the return type of the function. Additionally, the default return type will be void to demonstrate nothing to return. Function declaration will have function body statement within curly brace, which is the end of the function declaration.

```
function foo() : string {
}
// evaluates to function foo with a return type of string
```

Parameter List:

The parameter list consists of identifiers possibly with type expression.

```
    function foo(x, y){} // evaluates to x and y as parameter list
    function foo(x: int, y : int){} // evaluates to x and y both as integer as
parameter list
```

Function Body Statement:

The function body statement will have either the statement or the return statement. Which means the statement will be parsed as the body of the function declaration. The return statement will be talked about next.

```
    function foo(){
        print(1)
        print(2)
    } // evaluates to 1 2 as the result of print statement. It is running the
statements within the body.
```

Return Statement:

Although the return statement is not written within the statement of the grammar. Based on the function body statement, the return statement is on the same level as the other statement. Return statement will start with the key word of "return" followed by some expression. The return will only exist in function declarations.

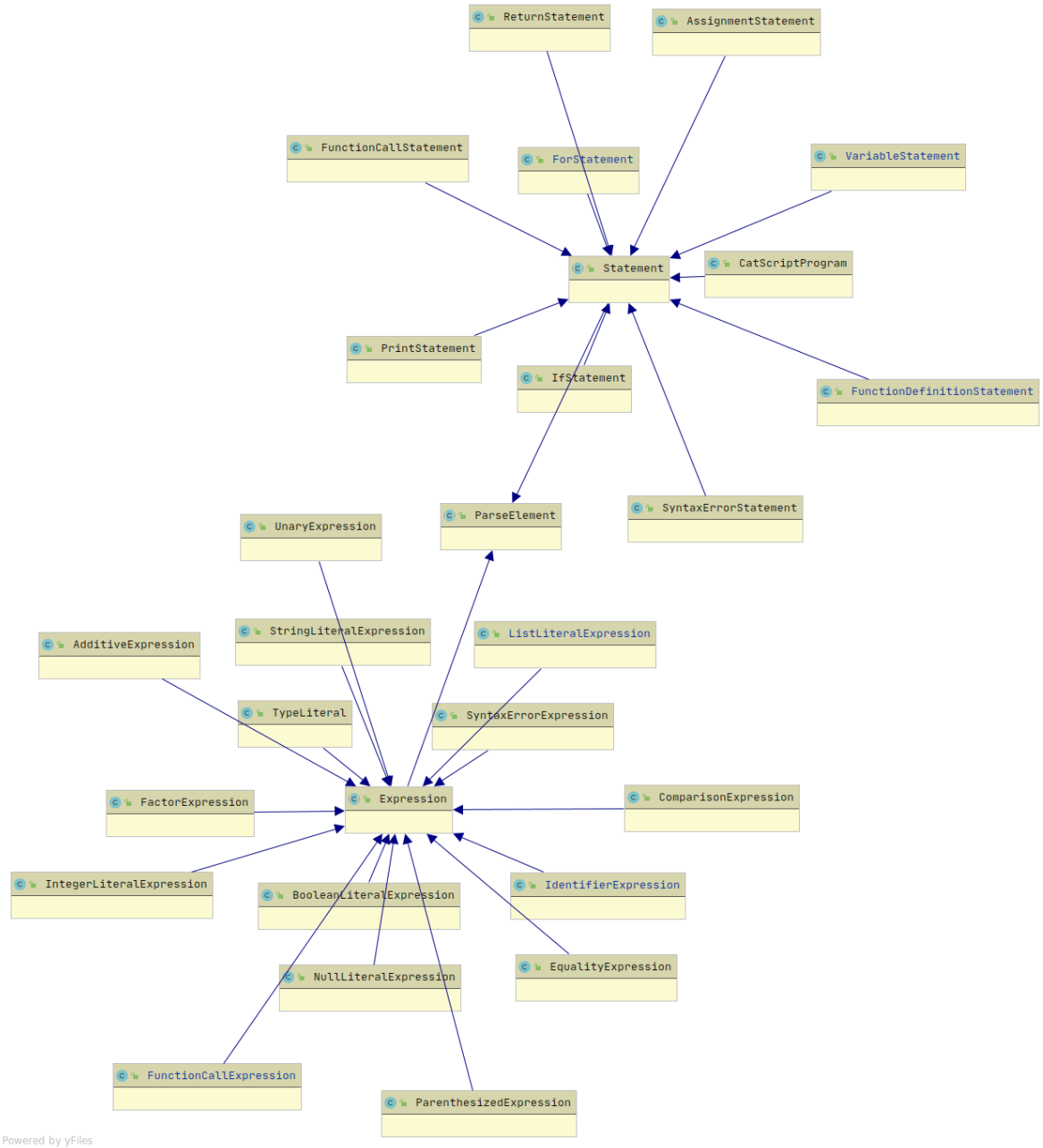
```
    function foo(): int{
        return 1
    } // evaluates to return integer of 1
```

Statement Conclusion:

Different than the expression, the statement does not run recursion. However, it is split into two different sections of statement and function declaration statement. All of the statement will be ran before function declaration is ran. Additionally, although return statement is not written within the statement in the grammar rules, based on the grammar itself, the return statement can only occur within the function declaration statement, but it is the same priorities as the other statements.

Section 5: UML.

One of the important aspects of the CatScript language is the parser itself. How the parser is designed will determine how the language will be interpreted. In the above section, the detail of each feature and how each feature will be parsed was talked about. In this section, the boarder view of the parser will be talked about. The following UML describes the parser at a top level:



At the very top level, Statement and Expression extends from parseElements. Additionally, the parenthesizedExpression, syntaxErrorExpression, listLiteralExpression, comparisonExpression, nullLiteralExpression, identifierExpression, functionCallExpression, equalityExpression, stringLiteralExpression, additiveExpression, booleanLiteralExpression, factorExpression, integerLiteralExpression, and unaryExpression are all extension of the expression class. Some of the extension are talked about previously as the grammar has been looked at. As seen, there are more function such as the syntaxErrorExpression that has not been talked about. SyntaxErrorExpression contains the possible errors that could occur because of the expression parser.

On the other hand, the UML also shows the different statements that extends from the statement. They are the ifStatement, functionDefinitionStatement (functionDeclaration), forStatement, functionCallStatement, printStatement, syntaxErrorStatement, returnStatement, assignmentStatement, catScriptProgram, and variableStatement. Within these unique statements, it is clear how some of it has been talked about previously, such as the if, for, function declaration, function call, print, return, assignment, and variable statement. Additionally, there are the syntaxError and catScriptProgram. SyntaxError are the possible errors associated with parsing statement. The CatScriptProgram is used since in the end, all the statements are going to be part of a larger program that will be ran.

Section 6: Design trade-offs

Introduction:

During this course, one of the main designs to choose was whether to use hand-written recursive descent or a parser generator when completing the parser. There are many parser generators available, such as ANLR. It will take the language specifications and outputs a parse tree. On the other hand, a handwritten parser is used for this course. The recursive descent algorithm or top-down parser is created for this course. In this section, the pros and the cons of the two methods will be talked about. Additionally, the choice of hand-written recursive descent parser will also be discussed.

Parser Generator:

There exist many parser generators; however, most of them have the same pros and cons. In theory, parser generators are better when one wish to complete the parser as fast as they can. The parser generator has the APIs and built-ins that will make writing the code for the parser much quicker and easier. Additionally, a parser generator will require less infrastructure to get going. Which could be very beneficial to someone new and need to complete the parser one step at the time. Furthermore, a parser generator could be much faster than many hand-written parsers, especially a hand-written parser that was created by someone doing a compiler for the first time.

On the other hand, there are some down sides to a parser generator. For example, although it does not require a lot of code to get working, if one wish to add some extra, specific rules to the grammar, it might require extra work than a hand-written parser. Next, if there were flaws in the parser generator used, it would be very time-consuming, since one would need to fix the parser code, if it is open-sourced, and ask the author of the parser generator to accept the changes. Therefore, it is close to impossible if one fix an error in the parser. Additionally, one would spend much more time reading the APIs of the parser generator rather than thinking about the actual parser. This takes away the educational purposes of writing a compiler. Lastly, even learning the parser generator does not mean that such skills is widely used in industry today.

Hand-written Recursive Descent:

The parser written for this course is a recursive descent algorithm, also known as top-down. Since one must write this from scratch, it would be easier to debug. One could include a break point and use the tools available to step through each line of code

and determine where the errors is. Additionally, as a parser, it is much easier to include an appropriate error message, which will make the end-product user-friendly. Lastly, by completing a compiler from scratch, it is a much better learning tool.

Although writing a parser from scratch might sounds like to ideal way to complete a compiler. It has a few down-sides as well. For example, the parser is difficult to write since one must understand the how the parser works before writing it. Additionally, it might be hard to get started since the compiler needs a lot of infrastructures to even run.

Conclusion:

Overall, to have a better learning experience, the hand-written parser was chosen for this course. Although it might be slower than the parser generator, speed of the parser is not one of the objectives for this course. Additionally, the professor has given the students a lot of the infrastructure code needed to run the compiler. Furthermore, the professor gave us a few example and hints about how to complete the parser. Thus, all the negative sides of a hand-written parser have been compromised. On the other hand, writing a parser from scratch will allow one to learn the fundamental of a parser. Additionally, one can learn something that could be used within the industry. In conclusion, there are just more benefits associated with a hand-written parser and that has been chosen for this course.

Section 7: Software development life cycle model

Introduction:

In this project, a test-driven development cycle is used. The cycle consists of adding a test, running all the test, write some code to pass the test, and repeat the process. The TDD tries to make everything as simple as they can be while maintaining all the features. Regarding this project, it is a little different, most of the tests were provided to us by the professor. Given each checkpoint, some code is written to make sure these tests pass. However, the full test-driven development was not used since the process never repeats.

Pros:

Within the course, TDD is one of the easier models that could be implemented in the course. To communicate effectively, a development model that has clear checkpoints should be used. Additionally, the model should be simple, easy to test and implement. Therefore, all the above criteria met the TDD model. Furthermore, the TDD model will make the grading portion easier. For the students, the TDD model made things clear regarding to what need to be implemented and what needs to be completed to make everything work. Following the checkpoints and grammar will ensure, on the big scale, that the compiler works.

Cons:

On the other hand, there were a few down-sides to the TDD. For example, the project has been a semester-long project, which means what has been completed might affect the later checkpoints. In one occasion, the way the program parsed the expressions was passing the test; however, it had more problems during the next phase when parsing the statement. This means the tests did not cover all possibilities. Additionally, the whole process of TDD was not implemented since the students were not asked to include more tests. The tests were set and even if additional problems were discovered, no other tests will be included on the individual basis.

Conclusion:

Although there were a few cons associated with using the TDD in this course, the pros are much beneficial. As time progresses, the professor will include some tests as he sees fit. For the first time using the model in this course, the flaws are small. Even working through these small errors and try to determine the bug was another learning process of its own. It is a great experience to learn about a simple development model like TDD.