****** **Faculty of Arts, Computing, Engineering & Sciences**

Department of Computing

Final Year Individual Project (SEGM)

[55-6727]

2015/16

|  |  |
| --- | --- |
| **Author:** | **Jordan Cain** |
| **Date Submitted:** |  |
| **Supervisor:** | **Christopher Bates** |
| **Degree Course:** | **BSc Computer Science** |
| **Title of Project:** |  |

|  |
| --- |
| **Confidentially Required?**  **NO 🞎**  **YES 🞎** |
|
|

Title page

Contents

[1 Abstract 5](#_Toc445837392)

[2 Introduction 6](#_Toc445837393)

[2.1 Project Overview 6](#_Toc445837394)

[2.2 Aims and Objectives 6](#_Toc445837395)

[2.3 Limitations Remove this and detail specifics in the aims and objectives 6](#_Toc445837396)

[3 Requirements 7](#_Toc445837397)

[3.1 Initial requirements 7](#_Toc445837398)

[4 Research 8](#_Toc445837399)

[4.1 Optimisations 8](#_Toc445837400)

[Chosen Optimisations 8](#_Toc445837401)

[4.2 Implementation Language choice 9](#_Toc445837402)

[4.3 Data Structure Abstract Syntax Tree 10](#_Toc445837403)

[4.4 AST Traversal 10](#_Toc445837404)

[4.5 Implementation Platform 10](#_Toc445837405)

[5 Design 12](#_Toc445837406)

[5.1 Abstract Syntax tree 12](#_Toc445837407)

[5.2 Parser 13](#_Toc445837408)

[5.3 Interface 13](#_Toc445837409)

[5.4 Testing 13](#_Toc445837410)

[6 Implementation 14](#_Toc445837411)

[6.1 Development approach 14](#_Toc445837412)

[6.2 Source Control 14](#_Toc445837413)

[6.3 Parser 14](#_Toc445837414)

[6.4 Optimisation detection 15](#_Toc445837415)

[6.5 Atom Plugin 15](#_Toc445837416)

[7 Testing 15](#_Toc445837417)

[8 Evaluation 15](#_Toc445837418)

[9 Reflection 17](#_Toc445837419)

[9.1 Emerging requirements 17](#_Toc445837420)

[9.2 Personal Development 17](#_Toc445837421)

[10 Bibliography 18](#_Toc445837422)

[11 Appendix 20](#_Toc445837423)

[11.1 Research - Optimisations 20](#_Toc445837424)

[11.2 Test Results 22](#_Toc445837425)

# Abstract

# Introduction

## Project Overview

Software development is an ever growing field with a diverse set of language choices, the TIOBE index shows Java as the most popular language of 2015 (TIOBE 2016) and Oracle states that Java has over 9 million developers worldwide (Oracle 2016). Java is a general purpose programming language designed to have as few dependencies as possible making it widely portable and understandable vastly popular. With an increasing number of self-taught developers emerging due to programmes such as Code Academy (codacademy 2016) and Hour of Code (code.org 2016) it stands to reason that the quality of Java code may be deteriorating. Whilst numerous IDEs (Interactive Development Environment) exist that provide syntax checking for Java developers no IDE offers suggestions based on the code execution or quality of code (NetBeans 2016) (Eclipse Foundation 2016).

This means potentially less well educated developers and shortfall from IDEs, an opportunity exists to create an application that will provide suggestions. The suggestions could be based on execution efficiency or code readability as either would improve the code quality.

The project application will be made up of multiple modules, a module to read, parse and store Java files, another to scan the stored files and discover areas for optimisations and produce suggestions, and finally a front end user interface to present the suggestions in a suitable environment.

## Aims and Objectives

**Aim**

The overall aim is to construct an application that can successfully parse a variety of Java files and produce a variety of helpful suggestions to a developer. The deliverable application will demonstrate the ability to provide a selection of optimisations upon which the application can be expanded to include more.

**Objectives:**

1. Research a variety of “optimisations” (areas of improvement) that can be applied to given Java code
2. Successfully read and store Java files of varying complexity
3. Explore the stored Java code whilst being aware of the context of each element
4. Identify helpful areas of optimisation within the Java code
5. Present the optimisations in a suitable and user friendly interface

## Limitations Remove this and detail specifics in the aims and objectives

The project and deliverable is intended to prove that Java optimisations can be spotted and suggestions provided to improve code quality, only a number of optimisations will be possible in the time frame provided. As a result of the time restriction the application will be written with expandability in mind thus allowing new optimisations to be added at a later date.

# Requirements

A list of requirements is essential to ensure that the application is fit for developers and expandable for future additional optimisations. The requirements will also serve as tests to determine the success of the project and deliverable. Whilst this list of initial requirements is produced it is likely that further requirements will present themselves throughout the development process and will be documented under emerging requirements.

## **Initial requirements**

**Research possible optimisations to Java code the application can provide**

Research will take place to determine a comprehensive list of optimisations that apply to Java code, the list of optimisations will allow the application to focus on specific areas. The areas of the optimisations to be researched will not be restricted as this will allow the application to be specialised at a later date.

**The application can parse a Java file and store the contents in a suitable data structure.**

The application must ensure that it can store a Java file in a suitable data structure, the most suitable data structure will be a Tree. Each node of the tree will have to represent a different “block” of the Java code, for example a package, class, method etc. will each be a node. The application will only be required to operate on one Java file at a time which can be assumed to only have one package but could have multiple classes. The different “blocks” of Java the application must handle are: Packages, Classes, Methods, Loops, Conditions, Try/Catch blocks and simple statements.

**The application can traverse the stored data structure and identify areas for optimisations.**

In order for the application to succeed it is essential that areas of the Java code can be inspected to determine if optimisations can be applied. The application should have the ability to inspect each line of Java and identify in order to determine if it’s an area for optimisation.

**Optimisation suggestions should be presented to the developer through a User Interface (UI).**

The application should have an interface with developers in order to provide suggestions, the interface should be simple to use and not require multiple applications to be running, i.e. integrate the suggestion interface into the developer’s current Java editor.

**Execution time of the application should be kept to a minimum.**

It is unacceptable for the application to slow a developers workflow, thus the application should prioritise time efficiency, this requirement is only likely to become an issue when the application is to be applied to large Java files with thousands of lines of code.

# Research

Before tackling the design and development of the application is important to first understand the parameters of the application, the parameters being the "parsability", storage and traversal of Java code along with the final output of the application.

## Optimisations

The first step in producing an application that provides Java optimisation suggestions is to produce a set of possible optimisations. All areas of optimisations will be considered during this stage in order to get a broad picture of current optimisation methods, from this broad list a small selection will be chosen to implement in the application.

A variety of different approaches to optimisation exist covering varies areas: Speed, Memory, Readability, portability, etc.

*Full list of the listed optimisations along with a more detailed explanation is available in the appendix of this report.*

## Chosen Optimisations

The chosen optimisations for the application will focus on the execution speed of Java, the two optimisations that will be implemented are for loop unrolling and Recursion prevention.

For loop unrolling is the process of taking a for loop that has a known number of iterations and replacing it with the same code repeated for the number of iterations. Unrolling for loops removes the need for a sizeable amount of Java bytecode, bytecode being the instruction set the JVM uses, without a for loop no extra variable, condition and goto statements are required (Troy Downing 1997). The table below shows the original Java and the compiled bytecode, the for loop code produces 11 line of bytecode, and a total of 35 instructions will be called during the execution (4 \* 6 + 11, 4\*6 being the number of instructions executed as a result of the goto). In comparison the unrolled version produces only 8 line of code and a total of 8 instructions will be executed to produce the same result.

|  |  |  |
| --- | --- | --- |
| Java for loop Code | Java bytecode | |
| int j = 0;  for(int i=0; i<5; i++){  j++;  } | 0: iconst\_0  1: istore\_1  2: iconst\_0  3: istore\_2  4: iload\_2  5: iconst\_5  6: if\_icmpge 18  9: iinc 1, 1  12: iinc 2, 1  15: goto 4  18: return | load int value 0 onto stack  pop stack, load int value into var 1  load int value 0 onto stack  pop stack, load int value into var 2  load int var from var 2  push int value 5 onto stack  pop two ints of stack, if value 2 is greater or equal to value 1 goto 18:  increment var with constant  increment var with constant  goto 4:  return void |
| Java unrolled loop | **Java bytecode** |  |
| int j = 0;  j++;  j++;  j++;  j++;  j++; | 0: iconst\_0  1: istore\_1  2: iinc 1, 1  5: iinc 1, 1  8: iinc 1, 1  11: iinc 1, 1  14: iinc 1, 1  17: return | load int value 0 onto stack  pop stack, load int value into var 1  increment var with constant  increment var with constant  increment var with constant  increment var with constant  increment var with constant  return void |

(Meyer and Downing 2016)

(Lindholm et al. 2015)

Recursive calls often require a large amount of stack space and can often lead to stack overflows, a stack overflow occurs when the call stack pointer exceeds the bounds of the allocated space I.E. when an application recurses too deeply (Oracle 2016). The JVM will throw a StackOverflowError when this occurs which often causes applications to crash. Another issue with recursive calls in Java is that they’re often slower than their iterative alternatives, this is a result of the heavy cost of function calls along with overhead associated with managing the stack.

The use of recursion is often to aid in the readability of code, recursive methods are often shorter and more elegant (fewer loops). Every recursive function can be transformed into an equally effective iterative function by replacing recursive calls with iterative alternatives (Drozdek 2012) and by doing so reduce the risk of crashing along with potential increases in time and memory efficiency (Mäyrä and Rönkkö 2007).

## Implementation Language choice

Whilst the application will be working with Java code the application does not have to be implemented using it, although consideration will be made for Java alternatives will be considered in order to choose the most appropriate language. A suitable implementation language will provide object orientation, appropriate data structures and rapid development.

**Python**

Python is a general purpose programming language with libraries for everything from networking and file I/O to Threading, it uses mutable data types (No type casting required) and requires little “Boiler plate” code so allows rapid development (McConnell 2009). Python supports multiple programming paradigms including object orientated, Functional and Procedural and is growing in popularity across the industry from rank 13 in 2003 to rank 4 in 2015 according to the TIOBE index (TIOBE 2016). The philosophy of the python language is described in the "Zen of Python" which is a collection of 20 software principles, amongst the principles are "Readability counts" and "Beautiful is better than ugly" (Peters 2004), with these principles Python code is easy to understand and will allow for the application to be easily open sourced and expanded by future developers.

Python is shipped as standard with OS X, El Captain (Apple operating system) and Ubuntu (The most popular Linux distribution) and is straight forward to install for windows. Along with the cross platform support for Python there are also a variety of editors available to develop in, for example Atom has syntax highlighting for Python as standard along with plugins available for Visual Studio, Eclipse and many more (Python Software Foundation 2016). Python applications can be run at the command line and thus would allow for simple integration into an atom plugin.

**Java**

Java is another general purpose programming language with a huge array of libraries, it relies on a large amount of “boiler plate” code but does run on the JVM so would allow for a very portable application assuming it only needed to run on the Command Line. Java would allow for relatively easy implementation of a plugin for an IDE such as Eclipse or NetBeans, and with my existing Java knowledge would not require a large amount of learning.

**C++**

C++ is another general purpose language that is closely linked to the hardware allowing for more powerful and efficient applications (Memory management, parallelism). Whilst performance is a priority for my application (i.e. a quick return time of suggestions) is necessary I will be able to achieve the required level with both Java and Python. C++ compiles to an executable file so would be widely portable but would only work in a console or using a GUI I produce.

**Final choice**

The application will be developed using Python as it allows for rapid development and prototyping due to its lack of boiler plate code. With its ability to run on the command line it should be straight forward to integrate into an IDE or editor. Python is heavily documented and should be simple to learn in a short time frame allowing application development to start with minimum delays.

## Data Structure Abstract Syntax Tree

Choosing a suitable data structure to hold the java files is quite straight forward, Java is written like a hierarchy, a package has many classes, which has many variables and methods etc. Therefor a tree structure fits well, the type of tree to be used is known as an Abstract Syntax Tree (AST). An AST is a tree representation of the abstract syntax structure of code, each node of the tree is responsible for managing a construct in the code though not every detail will need to be stored, thus the abstract nature (Object Management Group (OMG) 2011). ASTs are the industry standard for representing programming languages, for example both Netbeans and Eclipse, the most widely used Java editors utilise them (The Eclipse Foundation 2016) (NetBeans 2016).

An alternative approach to designing a custom Abstract Syntax tree would be to use an existing tool such as Antlr, Antlr is a language recognition tool that takes a set of rules (Grammar) and generates a parser that can build and walk parse trees (Parr 2016). Whilst an Application program interface (API) exists for Python to utilise Antlr it is limited to version 3.1.3 whereas the current version is 4.5.2, support is also no longer provided for the Python API.

## AST Traversal

To discover possible optimisations the AST will be traversed using a “Walker”, a custom Walker will be designed as opposed to using an integrated one in a tool such as Antlr. The Walker will perform both breadth and depth first searches of the AST, for example finding all the Class objects in the tree will require a breadth first search as the walker does not need to progress deeply into the tree. When searching for recursion each Statement object needs to be visited in order to detect all possible instances of recursion.

## Implementation Platform

**Eclipse/NetBeans Plugin**

Eclipse and Netbeans are the most common IDEs for Java developers, they provide comprehensive syntax guidance for developers as well as project builders (Ant) and GUI builders. Developing plugins for these IDEs is complex and would limit the application implementation to a language that will run on the JVM, whilst it is possible for python to run on the JVM using Jyphon (Jyphon 2016) it would limit the python version to 2.7.

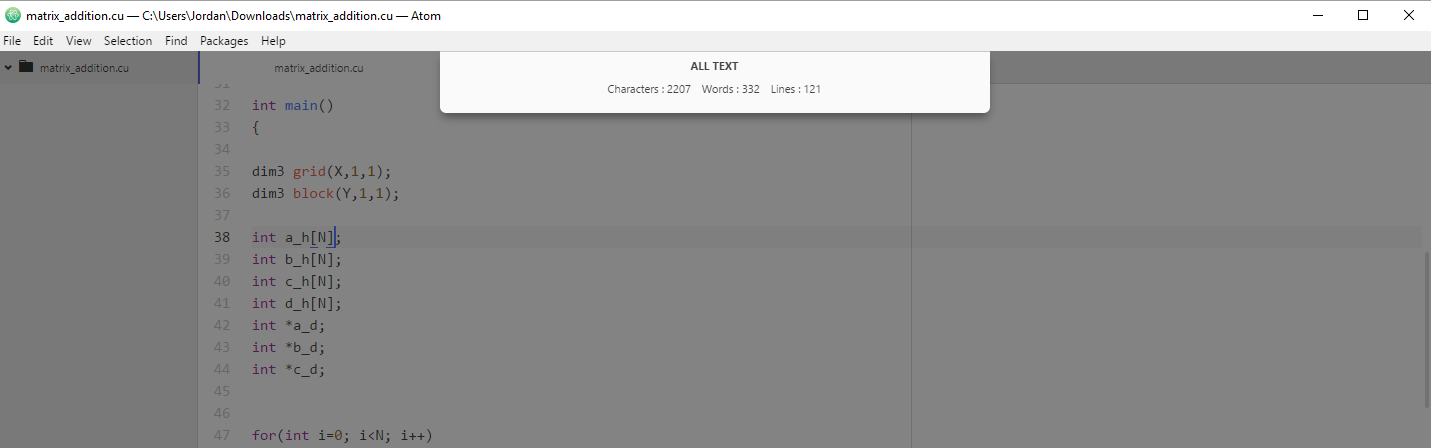
**Atom Package**

Atom is a relatively new open source text editor “A hackable text editor for the 21st century” (Atom Team 2016) growing in popularity amongst developers. Atom is built upon Electron which is an application allowing cross platform applications to be developed using web technologies (Github 2016). Atom uses packages to allow extensions and add-ons to be built directly into the editor using the atom package manager and generator. The package generator provides the package structure along with set up code and menu entries, Atom packages are written using CoffeeScript which transcompiles to JavaScript. Transcompiling is the process of taking one languages source code as input and returns the equivalent code in the new language.

As of 03/02/16 there are 4,828 packages publicly available for atom, some examples of popular packages are:

* Mini Map, Provides a preview of the complete source code of a file (Cai 2016)
* Merge Conflicts, Provides the ability to resolve Git conflicts within Atom (Wilson 2016)
* Linter, Provides a top-level API to its users that allows them to visualize errors and other kind-of messages (Brain 2016)

The User Interface for Atom must be written using the Atom API and CoffeeScript which is not yet heavily documented, the lack of documentation can be overcome by the quantity of packages that exists which are open source. The user interface can be expected to look similar to current packages, see below.



\*Example User Interface for Atom package.

With Atom and all of its packages being open sourced via Github (Kuychaco 2016) it will provide an excellent platform to make the application available to developers once completed.

**Command Line Interface**

A simple program that is called at the Command Line and prints its output to a file or to the console would achieve the goal of providing suggestions to a user and would allow for the application to developed in almost any language. The Command Line interface is not user friendly and would require the developer to exit their development environment to run the application thus interrupting their workflow.

**Implementation platform choice**

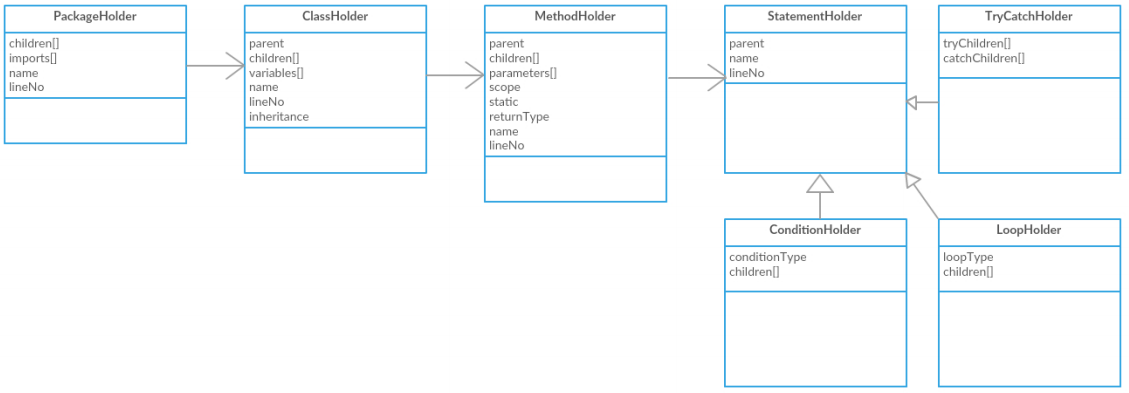
The application will be developed initially as a Command Line application, this application will then be integrated into an atom plugin. The atom plugin will have the ability to feature custom keyboard shortcuts for developers and maintain all of the developers coding inside a single editor. The application will have the ability to be indexed within all of atoms packages, making it easily accessible to Java developers.

# Design

## Abstract Syntax tree

The AST will hold the java code and make it easily accessible to the parser, each java container will have to be aware of its parent and children where applicable. The children will be implemented using Python lists which can expand and shrink as the java code is read in, parents will be implemented using object references. As Python uses Duck Typing the parent objects can be of any type thus allowing a statement to have a loop or method as its parent. Duck typing allows for a function to be applied to an object even if the object doesn’t formally meet the requirements of the function, i.e. the function will attempt to perform the operation no matter what the object is. The name Duck typing is derived from the expression, “If it looks like a duck and quacks like a duck, it's a duck” (Python Software Foundation 2016).

Class Diagram for Abstract Syntax Tree:

The name attributes of each class will be a string containing the actual line of Java code.

Java packages will be contained within the *PackageHolder* class, the application will only be designed to handle a single package so there will only ever be one P*ackageHolder*. The only special attribute a package will need is a list of imports, the name, line number and all of its children will also be stored. A *PackageHolder* will only have *ClassHolders* as children.

Classes are handled by the *ClassHolder* class which will need to be aware of special attributes of classes, inheritance (E.g. extends Runnable, Implements Thread) and class variables (class attributes). *ClassHolder* will also have to be aware of the class name, line number and all of its children. A *ClassHolder* will only have methods as its children although technically the variables (class attributes) are also children but will be separated into their own list of strings as they require no special operations.

All class methods will be held within *MethodHolders*, the *MethodHolder* will have to contain the parameters of a method, the scope (public/private), if the method is static (true/false) and the return type (int, String, etc.). *MethodHolders* will also hold the method name, line number and any children, the children of a method should only be statements.

*StatementHolder* will hold basic statements, it is a super class which extends to three child classes, *TryCatchHolder*, *ConditionHolder* and *LoopHolder*. The majority of lines within a Java file will be held in S*tatementHolders* where only the parent, name and line number are required.

Condition, Loop and *TryCatch* holders all inherit from the basic statement holder, they share the simple nature of having a parent, name and line number but they also require holders for their specific children.

White space and comments within the Java are recorded as *StatementHolders*, there is no need to create a separate class to hold them as no operations will ever be required of them.

## Parser

To store the Java code in the abstract syntax tree each line of the Java code must be evaluated to determine the type of line. For example both a C style for loop with an iterator and a for each loop must both be detected as for loops. To evaluate each line of code Regular expression and substring matching will be applied. The use of substring matching should be limited as to avoid matching patterns in comments, for example checking if a line contains the word “package” could detect multiple packages within the Java, instead the work package followed by white space and a string, then terminating with a semicolon will be much more reliable.

To build up and test the regular expressions the website RegExr will be used (Skinner 2016).

Python contains regex libraries and will only require the import library being imported, substring matching is also build into the default string class therefore no external libraries will be required.

When the parser detects a pattern match vie Regular expression or substring match a new object will be created, if the line of code is not a simple statement the parser will have to know a new block\* has been entered. All future statements found will then have to be associated with the new block via the children attributes and their own parent attribute. To determine the end of a code block the parser will have to be aware of closing block lines “}”, once a block closer is found the parser will have to return to the parent of the new block and then continue to parse and add new children to that.

\*block referring to code encapsulated within brackets { }

Reference the paper “Understanding Source Code Evolution Using Abstract Syntax Tree Matching” (Neamtiu, Foster and Hicks 2005)

## Interface

The application will produce a textual account of the optimisations it has found, it will be able to identify the location of the Java code the optimisation relates to along with providing replacement code when appropriate. An appropriate area for replacement code would be when a for loop can be unrolled, though this should be limited to only showing 5 iterations of the loop as not to bombard developers with data.

The interface will be presented within atom via an overlay popup which will output the suggestions, the suggestions will also be copied using atoms clipboard API to the developer’s clipboard giving them the ability to paste the suggestions into their code.

In order for atom to call the Python part of the application it will have to make use of “Buffered Processes” (Atom Team 2016), the buffered process will pass a command line call to the Python code and the output will be recorded in an atom variable ready to be printed out.

## Testing

Insert test plan here Actual test plan should be moved to appendix, just talk about what testing needs to take place and a few examples.

A helpful aid in testing software applications is to have the ability to build a stack trace and user trace of the application, these traces show the path through the code. Any failing tests will need to be fixed and the easiest way to identify the area of code responsible for a failure will be through trace files. As part of the application a tracing utility should be constructed, the utility will produce text files detailing what methods have been called and the parameters passed to them.

# Implementation

## Development approach

A software development approach is a structure applied to the development a specific project, a variety of different approaches exist which all value qualities such as Risk, Requirements and feedback differently (Jain 2011).

**Spiral**

The Spiral methodology utilises a series of phases that cascade as the development process progresses, the different phases are Planning, Risk Analysis, Engineering and Evaluation. The risk phase primarily involves gathering requirements, Risk Analysis focuses on research and consideration of different solutions, The Engineering phase is where the actual development and testing takes place and finally the Evaluation phase allows for review of the application before a new spiral starts. (Boehm 2002)

**Waterfall**

The Waterfall method defines several phases that are completed sequentially, one after the other and moving only to the next phase only when its preceding phase is completed (Bassil 2012). The Waterfall approach is best suited to projects that have fixed requirements, as the application is to be integrated into atom which is still in early stages of development the requirements for the project may change.

**Scrum**

Scrum is part of the Agile movement, Agile being the response to the failure of older static development approaches that can’t respond to changing requirements. The Scrum approach has 3 main roles, Product owner, Scrum master and Team; the Product Owner leads in terms of vision and direction of the project, Scrum Master ensures the team is not impeded in any way from achieving Sprint goals. The Team is responsible for self-organising and completing Sprints, Sprints are time periods in which a set of goals should be completed (James 2015).

The most suitable approach for the implementation of the application is the Spiral methodology, each stage of the application (Parser, Walker, GUI) will represent one spiral, as one stage of the application is completed only then will the next have its solution finalised. The Scrum aspect of a Sprint will also be utilised during the application development as it will provide deadline targets and ensure the development process doesn’t stall.

## Source Control

With the scope and modularity of this project it is essential that the code be developed with version control in mind, having a history and different branches will allow for bug fixes and feature additions whilst limiting the introduction of regressions. The tool I have chosen to use is Git, with Github being the repository provider. Whilst other source control applications exist such as Mercurial and Subversion, Git is the most popular and most heavily documented which will help when merge conflicts arise.

## Parser

Using Python as the implementation language allows for easy file reading, the “readLines” method in the standard python libraries allows for each line of a file to be added to a Python list. With the list containing each line of the Java code I will be able to iterate over it and test it against the different regex patterns for determining the properties of a line. When a line successfully matches a regex pattern the appropriate AST object is created and stored.

An example of the regex used to detect different java constructs,

re.search("(package)\s.+(;)", line):

re.search("((public)|(private))\s.\*(\().\*(\))", line):

re.search("((private)|(public))\s((static)\s)?[A-z]\*\s[A-z0-9]+\s\*((\;)|(\=.\*\;))", line):

re.search("((if)|(else if)|(switch))\s?(\()[0-9A-z\-\=\&\|\!\^\>\<\[\]\s\(\)\.\"\'\,\+]+(\))|(else)", line):

re.search("((for)\s\*(\().+(\)))|((while)\s\*(\().\*(\)))[^;]|(do)\s\*\{", line):

In line order: package, method, class attribute, condition, loop.

## Optimisation detection

To detect for loops that are suitable to be unrolled I will first get all the for loops within the AST, from this I will remove the iterable style for loops. With just the C-style for loops left each will need to have its iterating range checked, if the range is of a fixed amount (E.G. 0-5, int i = 0; i < 5; i++) then the loop can be deemed as unroll-able. With the suitable loops detected each one is analysed to determine if it’s suitable to output the unrolled alternative, the criteria for this being if the number of iterations is less than 5 as anymore would produce an unwieldy output. Loops too large to be unrolled will simply output a suggestive message.

In order to detect recursion I will get every method within the Java code, if the methods have any parameters then I will attempt to get the parameter types. Each statement within each method is then checked for the method name, if the method name is found the parameters of the call are then checked. The method name and parameter lists are then compared and if they match a recursion has been found.

## Atom Plugin

The user interface will be based off of Atoms developer tutorials, a simple overlay will appear with the application output. I’ll use the Buffered Process interface within the Atom API (Application Program Interface) to run the Python application which will take the Java files path as an argument, the file path will be gotten using the Atom API.

# Testing

A brief explanation of the testing process for the different aspects of the application is included within this section though a full breakdown of each test performed is available within the appendix.

Throughout the development of each module of the application testing has taken place, for example during the development of the parser each new Java structure (E.G. Package, Method, Loop) was tested to ensure it could be detected in any format. Once each individual structure had been tested independently they were then tested dependently on larger Java files in order to ensure different structures wouldn’t be detected incorrectly.

To test that no elements of the Java code where missing I would run a functioning Java class through the parser and once it had been stored in the tree it would then print it back out to the console. With the output I would then try to rerun it ensuring it functioned the same as the original Java code. This output also proves that the structure of the stored Java code matches that of the original, I.E a parent is aware of all of its children and stores them in the correct order.

As a result of the parser testing being very detailed, testing that for loops can be detected was not necessary when testing the for loop unrolling suggestions feature. Testing that a for loop could be unrolled successfully required presenting differently sized loops to the application, along with nested loops and loops that didn’t use a literal iterator limit in the condition (E.G. Literal: (int i = 0; *i < 5*; i++) Non Literal: for (int i = 0; *i < count*; i++)),

Testing the recursion aspect of the application required

Time efficiency stress testing

# Evaluation

To measure the success of the application as a whole I will refer back to the initial list of requirements outlined in section 3 along with evaluating the tools, techniques and development process utilised during this project. As I’ve been unable to find any similar applications I will be unable to compare the application with current industry tools. Finally I will evaluate the overall success of the complete application and its usability.

## Requirements

**Research a variety of “optimisations” (areas of improvement) that can be applied to given Java code**

I was able to discover over 20 possible optimisations that can be applied to Java code, the range of optimisations researched are of varying complexity, for example simply using start() on a thread over run() is much easier to implement then detecting objects will be fully dereferenced ready for garbage collection. The optimisations fall into two categories, runtime and compile time, runtime being optimisations that can only be suggested based on the execution of the Java code, compile time on the other hand are optimisations that can be applied based only on the Java code in its written form. An example of a runtime optimisation I discovered is using Appropriate Primitive Types, It’s not possible to accurately determine all the possibilities a variable can be assigned, An example of a compile time optimisation is For Loop unrolling, when the number of iterations of the loop is defined as a constant in the code It is always possible to unroll the loop.

The application has been developed to focus on compile time optimisations, specifically Recursion detection and For Loop unrolling as these are sufficiently challenging to demonstrate the thoroughness of the parser and walker. As a result of the high number of optimisations to pick from I feel this objective has been met to a very high standard allowing for two very interesting optimisations to be implemented that have previously not been integrated into an automated tool.

**Successfully read and store Java files of varying complexity**

To meet this requirement I’ve developed a custom parser and Abstract Syntax Tree (AST - data structure) to store Java code, each element of the parser and AST has been tested and results can be found in the Appendix. The test results show that all the desired aspects of Java code can be discovered, whilst this meets the requirements of this requirement the complexity of the parser is lacking in comparison to other tools. Tools such as Antlr are much more detailed in what the produce, for example Antlr is able to take a single statement and break it down in to individual components, so a statement such as 100 + 5 would be represented as digit – operator – digit – line terminator, and the same statement in my parser simply stores this as a statement object.

The varying complexity of Java files can be defined in many ways, for this requirement I will define a very complex Java file as one which creates a large AST (E.G. Several hundreds of individual nodes that require a depth of over 50 nodes). To determine the integrity of the tree with large Java files I’ve implemented the methods “getNodeCount” and “countChildren”, together these methods count the total number of nodes created. From visually counting the number of expected nodes in a large Java file and comparing that with the value returned by the aforementioned methods reveal that the application reaches this requirement. To ensure that the nodes are stored correctly and no data is lost I’ve implemented the ability to print out the Java file from only the data within the AST, a function “printNode” exists in each object holder to achieve this. By comparing the original data with the printed AST data I’m able to determine that no data is ever lost, further reassuring that this requirement is met.

**Explore the stored Java code whilst being aware of the context of each element**

By implementing my own AST walker the stored Java code can be fully explored, this is demonstrated in the application by the ability to pull out all the for loops when detecting loops to unroll. With pythons type checking facilities I’ve been able to determine the exact type of element the walker is visiting at any time allowing for easy tree traversal.

A drawback with the simplicity of the parser I discovered whilst implementing the walker was that matching elements such as method calls to their definitions required complex string matching and manipulation. Had I stored method calls more atomically (method name and list of parameters) instead of just as general statements I would have been able to implement a more powerful walker providing the ability to construct call trees and hierarchy. The walker is able to adequately provide an interface to the AST allowing for optimisations to be found so I deem this requirement to be sufficiently achieved though room for expansion is certainly available.

**Identify helpful areas of optimisation within the Java code**

With the two chosen optimisations being for loop unrolling and recursion detection I’ve been successfully able to demonstrate the ability to find areas for optimisation. Searching for for loops to unroll required checking that at compile time the number of iterations is fixed, by analysing the iterator and condition of the loop I’ve been able to reliably identify viable for loops. Identifying instance of recursion was much more difficult, it required analysing each method and determining its parameter types. I’ve been able to implement type checking in a number of ways, I can check for literal values (E.G. strings – “test” and ints – 56), method calls with return types and local variables. This type checking is sufficient for most Java files but unfortunately I’ve not been able to implement type checking for some of Java’s classes such as Calendar though by implementing String checking I’ve proven this would be possible. With a list of parameter types in a method call and a methods definition I’ve been successfully able to identify recursion providing proof this requirement has been fulfilled.

**Present the optimisations in a suitable and user friendly interface**

As discussed in the research chapter of this report I’ve chosen to use an Atom plugin as my user interface for this application, having used the Atom package generator I’ve been able to draw a simple

**Execution time of the application should be kept to a minimum.**

As detailed in the initial requirement description the time efficiency of the application will only become an issue when very large Java files are passed to it. During the development process I experimented with threading, specifically when searching the tree for objects I would have multiple threads running in parallel when conducting depth first searches. Unfortunately I was unable to successfully implement this and thus the functionality has been left out of the application.

To test that time efficiency is not an issue I’ve performed a series of stress tests of files in the range of 500- 2000 lines as described within the testing chapter of this report, the results show that even for large files the execution time for the application is almost instant and thus will not affect a developer’s progress. If the application were to be expanded to analyse multiple Java files or to incorporate further areas for optimisation suggestions parallelism and time efficiency improvements would become more important.

## Development process

Having chosen to adopt a Spiral development approach I’ve successfully been able to develop the different modules of the application to their fullest and have each component integrate well into the application as a whole. Initially I set myself the goal of completing each module of the application in a Sprint, each sprint was set to be a month, this approach was very successful and allowed for the bulk of the application to be completed within a few months. The difficulties in development came when tackling bugs, whilst I had allotted time to fix bugs I’ve made the decision to leave known bugs in the application. The bugs that have been left do not affect the core functionality of the application, for example a bug where block terminators “}” fail to be correctly aligned when outputting the contents of the AST.

# Reflection

## Emerging requirements

Throughout the development process of the project and countless meeting with my project supervisor some new requirements emerged, these requirements are based on the usability of the application.

**The application should not only make suggestions but also offer replacement code where applicable.**

It is not enough to just tell a developer that an area of their code can be optimised the application should be able to provide the optimised code. The optimised code should also be easily accessible, i.e. copied to the developer’s clipboard.

**The atom plugin should be fully portable.**

As atom is designed to be a portable editor it is essential to ensure that my application has no hard coded paths, as such I should ensure that the plugin is aware of the current file being edited and the runnable python application.

The new requirements were simple tasks to implement and greatly improve the legitimacy of the application,

## Personal Development

Having never used the Python programming language prior to this project it has been the primary learning experience, through the use of Python documentation and countless stack overflow questions I’ve become proficient in the Python language. Using the Regex python library I’ve been able to further my knowledge in the use of regular expressions and pattern matching. Throughout the development and testing process I’ve had the opportunity to further study debugging utilities, I’ve discovered the Python inspect library and have been able to create my own trace files, showing every method call throughout the execution of the program.

Throughout the research aspect of this project I was able to discover more of the inner working of the JVM, having previously never looked at Java byte code I’ve become aware of some of the simple instruction available. With this deeper Java knowledge I’m now aware of the best practices and how to improve Java application performance.

This application is the first time I’ve approached creating a plugin for an editor, the Atom documentation has provided an easy introduction to process along with the opportunity to learn CoffeeScript. As a result of this learning I feel comfortable producing further helpful plugins both for my own convenience, future colleagues and other developers.

## Further work

More optimisations

Parallel

Improved GUI

# Bibliography

ATOM TEAM (2016). *Atom Docs.* [online]. <https://atom.io/docs>

BASSIL, Youssef (2012). A Simulation Model for the Waterfall Software Development Life Cycle. *International Journal of Engineering & Technology*, **2** (5),.

BOEHM, Barry W. (2002). A Spiral Model of Software Development and Enhancement. *Computer*, **21** (5), 61-72.

BRAIN, Steel (2016). *Linter.* [online]. <https://github.com/atom-community/linter>

BRUCE TATE, Justin Gehtland (2004). *Better, faster, lighter Java*. O'Reilly.

CAI, Fangdun (2016). *Atom Minimap.* [online]. Last updated 2016 March. <https://github.com/atom-minimap>

CODACADEMY (2016). *codacademy.* [online]. Last updated 2016 February. <https://www.codecademy.com>

CODE.ORG (2016). *Hour of Code.* [online]. Last updated 2016 February. <https://hourofcode.com/>

DROZDEK, Adam (2012). *Data Structures and Algorithms in C++*. 4 ed., Cengage Learning.

GITHUB (2016). *Electron.* [online]. <http://electron.atom.io/>

JAIN, Namrata (2011). Software Development Life Cycle: A Detailed Study. *International Journal of Advanced Research in Computer Science*, **2** (3), 261-264.

JAMES, Michael (2015). *Scrum Methodology.* [online]. <http://scrummethodology.com/>

JYPHON, Python Foundation (2016). *Jyphon.* [online]. <http://www.jython.org/>

KUYCHACO, as-cii, joshaber, et al (2016). *Atom Github Repositry.* [online]. <https://github.com/atom/atom/>

LINDHOLM, Tim, et al. (2015). *The Java® Virtual Machine Specification.* [online]. <http://docs.oracle.com/javase/specs/jvms/se8/html/index.html>

MARK, Ronald (2009). *Writing Compilers and Interpreters: A Software Engineering Approach*. 3 ed., John Wiley & Sons.

MATTHEW ARNOLD, Michael Hind, Barbara G. Ryder (2002). Online feedback-directed optimization of Java..

MÄYRÄ, Hannu and RÖNKKÖ, Mauno (2007). Software Engineering Advances. Cap Esterel, 25-31 Aug. 2007. , 31-31.

MCCONNELL, Steve (2009). *Code Complete*. Microsoft Press.

MEYER, Jon and DOWNING, Troy (2016). *Jump if one integer is greater than or equal to another.* [online]. <http://cs.au.dk/~mis/dOvs/jvmspec/ref--26.html>

NEAMTIU, Iulian, FOSTER, Jeffrey S. and HICKS, Michael (2005). Understanding Source Code Evolution Using Abstract syntax tree matching. In: *International Conference on Software Engineering*, New York, ACM, 1-5.

NETBEANS (2016). *NetBeans.* [online]. <https://netbeans.org/>

NIEMANN, Benjamin (2010). *ANTLR Python API Documentation.* [online]. <http://www.antlr3.org/api/Python/>

OBJECT MANAGEMENT GROUP (OMG) (2011). *Architecture-driven Modernization: bstract Syntax Tree Metamodel*. [online]. <http://www.omg.org/spec/ASTM/1.0/>

ORACLE (2016). *Java.* [online]. <https://www.java.com/en/about/>

PARR, Terence (2016). *Antlr.* [online]. <http://www.antlr.org/>

PAUL, Javin (2014). *Java Revisited Common MultiThreading Mistakes.* [online]. <http://javarevisited.blogspot.co.uk/2014/09/common-java-multi-threading-mistakes-1-run-vs-start.html>

PAUL, Javin (2015). *Java Revisited How to reset arraylist in java clear vs removeAll.* [online]. <http://javarevisited.blogspot.co.uk/2015/09/how-to-reset-arraylist-in-java-clear-vs-removeAll-example.html>

PETERS, Tim (2004). *PEP 20 -- The Zen of Python.* [online]. <https://www.python.org/dev/peps/pep-0020/>

PYTHON SOFTWARE FOUNDATION (2016). *Python 3.5.1 documentation.* [online]. <https://docs.python.org/3/>

SIMMONS, Robert (2004). *Hardcore Java*. O'Reilly.

SKINNER, Grant (2016). *RegExr.* [online].

THE ECLIPSE FOUNDATION (2016). *Eclipse.* [online]. <https://www.eclipse.org/org/>

TIOBE (2016). *TIOBE Software.* [online]. Last updated January 2016 January. <http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

TROY DOWNING, Jon Meyer (1997). *Java Virtual Machine*. 1st ed., O'Reilly.

WALDO, Jim (2010). *Java: The Good Parts*. O'Reilly.

WIKIBOOKS (2011). *Python programming*. CreateSpace.

WILSON, Ash (2016). *Merge Conflicts.* [online]. Last updated 2016 March. <https://github.com/smashwilson/merge-conflicts>

# Appendix

## Research - Optimisations

**For Loop Unrolling**

For loops can be unrolled to increase efficiency if the number of iterations is known, the efficiency comes from the reduction in the amount of code required to run, no new variable is require, no condition tests, incrementing and GOTO statements are required (Troy Downing 1997). An example of loop unrolling:

|  |  |
| --- | --- |
| for (int i=0; i<3; i++){  checkStatus(i);  } | checkStatus(0);  checkStatus(1);  checkStatus(2); |

**Recursion Optimisation**

The JVM struggles to optimise recursion in terms of performance,

**JIT – Just In Time Compilation**

**Appropriate Primitive Types**

Correct data types used, long and doubles require 64bits to store, the JVM operand stack is only 32bits and thus two positions on the stack are required which eats up more memory, Longs and doubles should only be used when absolutely necessary. JAVA VIRTUAL MACHINE

**Full Object Dereferencing - Garbage Collection**

Objects can only be free for garbage collection when no more references to it exist, a common mistake is people leaving an object reference hanging.

E.g.: A class defining a football team holds a reference to the player objects, a player may retire and not have its reference removed from the team thus occupying memory.JAVA-THE GOOD PARTS

**Deprecated Classes**

Oracle warns against the use of deprecated classes as they may be removed from the standard JRE (Java runtime environment) at any time thus reducing the longevity of applications. As most IDEs already warn users against this I will no implement this. Better, Faster, Lighter Java (book)

**Cut & Paste code**

Cut and paste applications typically lead to bloated applications as not all parts of the cut code is truly required, all methods should be written from scratch. Better, Faster, Lighter Java (book)

**Printing Exceptions to Console**

Not all applications will have access to a console output so relying on this to print exceptions should be avoided, instead log files can be used to write exceptions to. Hardcore Java (book)

**Run()/ Start() for Threads**

When trying to start a new thread by using Run() instead of Start() no new thread is created, the Start() method is what creates the new thread and then calls Run() allowing the application to run concurrently rather than sequentially.

**ArrayList RemoveAll()/ Clear()**

When you wish to clear an ArrayList it is almost more efficient to use clear() over RemoveAll() as clear() gives you O(n) performance, while removeAll(Collection c) is worse, it gives O(n^2).

**Code In-Lining**

**Conditional Statement Ordering**

**Constant Folding**

**Constant Propagation**

**Strength Reduction**

I\*5 – i+I+I+I+I

**Common Sub-Expression Elimination**

(I+j) \* (i+j) – t = i+j; t\*t

## Test Results

|  |  |  |
| --- | --- | --- |
| Test Description | Expected Outcome | Actual Outcome |
| Parser | | |
| Detect Packages | Packages are identified and no instances of wrongful detection |  |
| Create *PackageHolder* | Package is stored with name and line number |  |
| Detect Imports | Imports are identified and no wrongful detections |  |
| Append imports to PackageHolder | Import line is stored in the imports list of PackageHolder |  |
| Detect Classes |  |  |
| Create *ClassHolder* |  |  |
| Associate Class with Package |  |  |
| Detect Class attributes |  |  |
| Detect Methods |  |  |
| Create *MethodHolder* |  |  |
| Associate Method with Class |  |  |
| Detect Method parameters |  |  |
| Detect Method scope |  |  |
| Detect if Method is static |  |  |
| Detect Method return type |  |  |
| Detect Statement |  |  |
| Create *StatementHolder* |  |  |
| Associate Statement with Method |  |  |
| Detect Try Catch blocks |  |  |
| Create *TryCatchHolder* |  |  |
| Detect statements and associate with try block |  |  |
| Detect statements and associate with catch block |  |  |
| Detect C style for loop |  |  |
| Detect *foreach* style for loop |  |  |
| Detect while loop |  |  |
| Detect do while loop |  |  |
| Create *LoopHolder* |  |  |
| Detect statements and associate with loop |  |  |
| Detect if statements |  |  |
| Detect else statements |  |  |
| Detect *elseif* statements |  |  |
| Detect switch statements |  |  |
| Create *ConditionHolder* |  |  |
| Walker | | |
| Get all Classes |  |  |
| Get all Methods |  |  |
| Get all For Loops |  |  |
| Get all Statements |  |  |
| Optimisation detect | | |
| Get for loops with constant number of iterations |  |  |
| Get child statements within for loop |  |  |
| Store un-rollable loop |  |  |
| Output unrolled version of loop |  |  |
| Detect a method call |  |  |
| Get parameters of method call |  |  |
| Get parameter types |  |  |
| Get all method names |  |  |
| Check method call and parameters against parent method |  |  |
| Store recursive call |  |  |
| Output location of recursive call |  |  |
| GUI | | |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |