

Thesis



Edge Hill University

Department of Computing

A Generic Framework Facilitating Automated Quality Assurance across Programming Languages of Disparate Paradigms

Darryl Owens

March 2016

St Helens Road
Ormskirk L39 4QP
United Kingdom
+44 (0)1695 575171

<http://www.edgehill.ac.uk/>

© 2016 Darryl Owens

This technical report is based on a dissertation submitted in March 2016 by the author for the degree of Doctor of Philosophy to Edge Hill University.

Dedication

This work is dedicated to Barbara and Glyn Owens, my parents, who have supported me throughout this work and all of my education. Without these two people, none of my work would have been possible; they passed on a drive for knowledge and deserve more credit for my success than myself.



In memory of Philip (Tompi) Thompson

13/10/1929-22/01/2016

A Tompi Grandson Production

Acknowledgments

Firstly, I would like to express my gratitude to Professor Mark Anderson, John Collins, and Professor Michael Inggs; I could not have asked for better or more supportive advisors. Additionally, I would like to thank David Gill of UCAR and Brian Farrimond of SimCon for accepting and assisting me as part of the QACC team. I would like to especially thank Professor Mark Anderson who, with his inspirational and motivational attitude, made my project completion possible.

This thesis was funded by Edge Hill University, and I would like to thank the organisation for the opportunity and support. As a member of Edge Hill University, I have been surrounded by wonderful colleagues; the community has provided me with support and guidance in both academic and personal development.

I would like to thank the supportive group that helped me survive the academic process and tolerated my incessant complaining. By name, these are Olushola Alexander Akinbi, Daniel Campbell, Daniel Kay, Peter Mattew, and finally my partner, Harriet Hamilton, who deserves an award for tolerance and compassion.

Finally, I would like to thank my family and friends for their tireless support and assistance not only over the time of my PhD but also throughout my education and my last six years at Edge Hill University, with a special mention to my Uncle Stan Hutchinson, who had the painful experience of proofreading this thesis.

A Generic Framework Facilitating Automated Quality Assurance across Programming Languages of Disparate Paradigms

Darryl Owens

Abstract

This research aims to outline a framework based on procedural and object-oriented Paradigms that facilitates generic automated quality assurance. Along with the outline, a skeleton framework has been developed to evaluate the research, and the final aim is to expand the footprint of the framework; theoretical inclusion of other programming paradigms has been discussed. This research developed a taxonomy of quality assurance techniques in order to identify potential candidates for generic quality assurance and also to minimise experimental requirements, as the taxonomy categories are generated based on implementation requirements; this means that a category can be deemed feasible within the scope of this framework if a single technique can be implemented. The novel aspects of this research are the taxonomy, paradigm-specific framework, and finally the theorised paradigm-generic framework. An experimental method has been used to provide evidence to support the claims made by this research, which is accompanied by a study of literature providing a foundation for all areas discussed. Although a paradigm-generic framework can be achieved, the internal representation used in this research showed that application of the logical paradigm would not be simple and has little benefit in the scope of automated quality assurance. This being said, procedural, object-oriented, and functional paradigms have been demonstrated as feasible with significant impact on programming language development and automated quality assurance of software.

Table of Contents

Chapter 1.	Introduction.....	18
1.1	Background and Motivation	18
1.2	Previous Work	20
1.3	Published Work.....	20
1.4	Argument	21
1.5	Scope.....	21
1.6	Aims and Objectives	22
1.7	Methodology and Methods	23
1.7.1	Epistemology	24
1.7.2	Requirements Gathering	25
1.7.3	Taxonomy	25
1.7.4	Design and Implementation	25
1.7.5	Evaluation	26
1.7.6	Exploratory Study	26
1.8	Original Contribution.....	26
1.9	Ethical Considerations	28
1.10	Personal Motivation	29
1.11	Thesis breakdown	29
Chapter 2.	Literature Review.....	30
2.1	Programming Paradigms and Languages.....	30
2.2	Programming Language Independence.....	34
2.2.1	Abstract Syntax Tree.....	36
2.2.2	Generic Abstract Syntax Tree Meta-model (GASTM).....	39

2.3	Software Quality Assurance	40
2.4	Static and Dynamic Analysis	42
2.4.1	Techniques and Taxonomies.....	43
2.5	The need for Automated Quality Assurance.....	47
2.6	Current Development in Software Quality Assurance.....	48
Chapter 3.	Language Independent Quality Assurance (LIQA) Outline	50
3.1	Proposal.....	50
3.2	Design	54
3.2.1	Methodology	54
3.2.2	Development Tools	55
3.2.3	Initial Requirements.....	56
3.2.3.1	Form Design.....	57
3.3	Implementation	57
3.3.1	Evolutions / Version	58
3.4	Tools	59
3.4.1	Modifications to Generic Abstract Syntax Tree Meta-model (GASTM)	
	60	
3.4.2	Limitations	62
3.4.3	Test Dynamic Analysis	64
3.4.4	Test Static Analysis.....	65
3.4.5	Overall System Description	65
3.4.5.1	Development Discussion.....	67
3.4.5.2	Finalized IR Interface.....	73
3.5	Modifications to Research	74

3.6	Summary	74
Chapter 4.	Taxonomy of Quality Assurance Techniques	75
4.1	Overview of Quality assurance	76
4.1.1	Detailed Static	78
4.1.2	Detailed Dynamic	79
4.2	High-Level Tool Analysis.....	80
4.2.1	Independent Tool High-Level Analysis.....	80
4.2.1.1	winFPT	80
4.2.1.2	Parasoft.....	81
4.2.1.3	Malpas	81
4.2.1.4	Polyspace.....	82
4.2.1.5	Cantata++	82
4.2.1.6	JNuke.....	82
4.2.1.7	TestingAnywhere	83
4.2.1.8	Critical Comparison	83
4.2.2	IDE High-Level Analysis	85
4.2.2.1	NetBeans	86
4.2.2.2	Eclipse	86
4.2.2.3	Visual Studio	87
4.2.2.4	Critical Comparison	89
4.3	Deep Analysis	90
4.3.1	NetBeans	90
4.3.1.1	NetBeans Java Hints.....	90
4.3.1.2	NetBeans Debugger.....	91

4.3.1.3	NetBeans Profiler	92
4.3.1.4	NetBeans JavaDoc Analysis	93
4.3.1.5	SQE (Software Quality Environment)	93
4.3.1.6	FindBugs	94
4.3.1.7	PMD Source Code Analyser	94
4.3.1.8	Dependency Finder	95
4.3.1.9	Checkstyle	96
4.3.1.10	JUnit and xUnit Framework.....	96
4.3.1.11	Techniques	98
4.3.2	Visual Studio.....	100
4.3.2.1	Debugger	100
4.3.2.2	Error Correction	105
4.3.2.3	Analyser	105
4.3.2.4	Issues and Limitations.....	109
4.3.2.5	Techniques	110
4.3.3	Integrated Development Environment (IDE) Comparison	112
4.3.4	WinFPT.....	113
4.3.4.1	Internal Representation	113
4.3.4.2	Features	114
4.3.4.3	Techniques	118
4.3.5	Polyspace	120
4.3.5.1	Features	120
4.3.5.2	Techniques	124
4.3.1	Critical Comparison	125

4.4	Techniques	125
4.4.1	Diagram Key	126
4.4.2	Static	127
4.4.3	Dynamic	130
4.5	Taxonomy of Techniques	131
4.6	Additional Tools	134
4.7	Category Summary	135
4.8	Explicit taxonomy	136
4.9	Summary	141
Chapter 5.	Framework	142
5.1	Implementation of Techniques	142
5.1.1	Static Analysis	144
5.1.1.1	Code Manipulation	145
5.1.1.2	Optimization	148
5.1.1.3	Data Flow Analysis	149
5.1.1.4	Static Metrics	153
5.1.1.5	Pattern Matching	157
5.1.2	Dynamic Analysis	157
5.1.2.1	Dynamic Metrics (Profiler)	157
5.1.3	Development Discussions	158
5.2	Discussion of Theoretical Techniques	159
5.2.1	Static Analysis	160
5.2.1.1	Visualisation	160
5.2.1.2	Artefact Generation	160

5.2.2 Dynamic Analysis	161
5.2.2.1 Unit Testing.....	161
5.3 Testing.....	161
5.3.1 Testing Plan	161
5.3.2 Testing Notes	166
5.4 Analysis of Results	167
5.5 Framework Conclusion.....	174
5.6 Summary	175
Chapter 6. Theoretical Discussion.....	176
6.1 Paradigms.....	176
6.1.1 Paradigm Discussion.....	176
6.1.2 Paradigm Analysis	177
6.1.2.1 Procedural.....	179
6.1.2.2 Object-Oriented Paradigm.....	181
6.1.2.3 Functional.....	182
6.1.2.4 Logical.....	189
6.1.2.5 Other Paradigms	192
6.1.3 Generic Quality Assurance	195
6.2 Complete Framework / Future Work.....	197
6.2.1 Direct Expansion.....	197
6.2.2 Grammar Prefix	199
6.2.3 Query Addition	200
6.2.4 Linking with IDE	201
6.2.5 Dynamic Analysis.....	202

6.3	Summary	204
Chapter 7.	Conclusion	205
7.1	Empirical Findings.....	206
7.2	Theoretical Implication.....	208
7.3	Recommendations for future research	209
7.4	Limitations of the study	210
7.5	Personal Reflection	211
7.6	Conclusion of the conclusion.....	212
Chapter 8.	Bibliography	214
Chapter 9.	Glossary	249

List of Tables

Table 2.1.1 - Language taxonomy	33
Table 2.4.1 – Initial Taxonomy Outline.....	46
Table 3.2.1 - Initial requirements table	56
Table 3.3.1 - Version list.....	59

List of Figures

Figure 1.8.1 - Minimal risk [8]	28
Figure 2.2.1 - Java hello world Abstract Syntax Tree (AST)	37
Figure 2.2.2 - C# hello world Abstract Syntax Tree (AST).....	37
Figure 2.3.1 - Scope and content hierarchy [42].....	40
Figure 2.6.1– Software Quality Assurance (SQA) current visualisation.....	49
Figure 3.1.1- System data flow	51
Figure 3.1.2 - Static analysis data flow	52
Figure 3.1.3 - Dynamic analysis data flow	53
Figure 3.2.1 - Initial form design	57
Figure 3.4.1– Generic Abstract Syntax Tree Meta-model (GASTM) SwitchStatement definition.....	61
Figure 3.4.2 – Language Independent Quality Assurance (LIQA) expected data flow diagram	66
Figure 3.4.3 – Language Independent Quality Assurance (LIQA) file structure ..	68
Figure 3.4.4 – Language Independent Quality Assurance (LIQA) class diagram Graphical User Interface (GUI)	69
Figure 3.4.5 – Language Independent Quality Assurance (LIQA) class diagram Control Flow Graph (CFG).....	70
Figure 3.4.6 – Language Independent Quality Assurance (LIQA) class diagram identifiers	70
Figure 3.4.7 – Language Independent Quality Assurance (LIQA) class diagram Internal Representation (IR) builders	71
Figure 3.4.8 - Internal Representation (IR) generator form.....	73
Figure 3.4.9 - Menu design	73
Figure 4.1.1– Hierarchy key	77

Figure 4.1.2– Basic analysis hierarchy	77
Figure 4.1.3– Static analysis hierarchy	78
Figure 4.1.4 – Dynamic analysis hierarchy	79
Figure 4.3.1 – NetBeans dependency finder	95
Figure 4.3.2 – NetBeans dynamic analysis techniques	98
Figure 4.3.3 – NetBeans static analysis techniques	99
Figure 4.3.4 – C# debugger test program	101
Figure 4.3.5 – C# debugger tracepoint message	102
Figure 4.3.6 – C# debugger variable watcher	103
Figure 4.3.7 – C# debugger assert classes	103
Figure 4.3.8 – C# debugger intelitrace	104
Figure 4.3.9 – Visual Studio 2012 performance analyser	106
Figure 4.3.10 – Maintainability index calculation [173]	107
Figure 4.3.11 – Visual Studio 2012 code clone detection	109
Figure 4.3.12 – Visual Studio 2012 dynamic analysis techniques	111
Figure 4.3.13 – Visual Studio 2012 static analysis techniques	111
Figure 4.3.14 – Warning identification in winFPT	116
Figure 4.3.15 – Dynamic analysis techniques for winFPT	118
Figure 4.3.16 – Static analysis techniques for winFPT	119
Figure 4.3.17 – Polyspace bug finder & code prover	120
Figure 4.3.18 – Polyspace static analysis techniques	124
Figure 4.4.1 – Tool key	126
Figure 4.4.2 – Static analysis 1	127
Figure 4.4.3 – Static analysis 2	128

Figure 4.4.4 – Static analysis 3	129
Figure 4.4.5 – Dynamic analysis.....	130
Figure 4.4.6 – Taxonomy of quality assurance techniques.....	133
Figure 4.4.7 – Taxonomy of quality assurance techniques with rules.....	138
Figure 5.1.1 –Quality assurance techniques being implemented.....	143
Figure 5.1.2 – Variable rename example	146
Figure 5.1.3 – Graphical User Interface (GUI) for variable rename	147
Figure 5.1.4 – Switch to If example.....	148
Figure 5.1.5 – For unroll example	148
Figure 5.1.6 – CFGObjects class diagram	150
Figure 5.1.7 – CFGObjects example [200].....	150
Figure 5.1.8 – Liveness analysis limitation example.....	152
Figure 5.1.9 – Dead code limitation example.....	152
Figure 5.1.10 – Dead code multiple instances	153
Figure 5.1.11 – Cyclomatic complexity example	154
Figure 5.1.12 – Halsteads complexity example [209]	156
Figure 5.4.1 – Equation 1.....	167
Figure 5.4.2 - Equation 2	168
Figure 5.4.3 – Pie chart showing individual test results	168
Figure 5.4.4 – Pie chart showing grouped test tests results	169
Figure 5.4.5 – Bar graph showing categorised test results	170
Figure 5.4.6 – Bar graph showing categorised pass results	171
Figure 5.4.7 - Additional Profiling Test with original calculation	172
Figure 5.4.8 Additional profiling test code.....	173

Figure 6.1.1 – Paradigm Breakdown [215].....	178
Figure 6.2.1 – Language Independent Quality Assurance (LIQA) grammar prefix	199

Chapter 1. Introduction

This thesis covers a wide canvas under the area of software engineering, including programming languages and their respective paradigms, as well as automated Software Quality Assurance (SQA). Therefore, it must be clear exactly what is discussed and intended by these terms. In this research, automated SQA is any written and implemented programming technique that is free from human intervention, and either informs or modifies a program's source code with the intent of improving its 'quality'. There are many ways to improve the 'quality' of software, and therefore this will be discussed at length in the literature review. The second point of interest in this research is programming languages and paradigms. The meaning of 'programming language' is self-explanatory; however, the paradigms are a topic of discussion, as many programming languages branch over multiple paradigms. A paradigm can be described as a way of approaching a problem, or a way of thinking [1]. With this in mind, to classify a programming language as within a paradigm, it must have features that enable the approach of a problem following the paradigm's ideals. Further discussion of paradigms will be included in the literature review.

With the description of the areas above, the aim for the research can be better understood. This research aims to create a framework that could be used to enable automated SQA techniques to be applied to a program regardless of programming language or paradigm. This is a major task and will therefore be split into several aims and further into objectives later in this chapter. It is important to say that to test this framework, a 'skeleton' version will be built. This will provide a point of discussion and allow quantitative analysis on real data.

1.1 BACKGROUND AND MOTIVATION

Automated SQA is a set of techniques that are applied to a program to improve its 'quality'. Quality is generally defined by individual projects and their domains, and is a set of categories that have been placed in priority order. These characteristics set out various standards used by businesses and standards bodies; examples of categories include performance,

maintainability, and functionality. The applications of these techniques are implemented through Integrated Development Environments (IDEs), plug-ins for IDEs, and standalone programs.

There are several motivating reasons for this research to be conducted. Understanding the background is key in recognising the wide impact that this research could have on the culture of software quality assurance.

Firstly, consider that built into most IDEs are automated quality assurance techniques that are taken for granted by software developers. These IDEs, regardless of which programming language they are specialised for, have similar ways of identifying issues with source code. It also happens that the IDEs identify the same problems. However, there is no middle ground in which these techniques are implemented. The techniques are just recreated for each IDE. This means that wide adoption of new automated quality assurance techniques takes time and requires many different people to agree and implement upon their product.

Secondly, we can consider programs that are written in multiple programming languages, such as distributed systems of large complex software such as WRF, a weather forecasting model. Systems such as these can require multiple tools to assure their quality and, in doing so, may apply different techniques to each programming language.

The current focus of automated SQA is on specific techniques to be applied to different domains, whereas this research is designed to change the fundamentals of the field targeting the platform in which these techniques are developed and deployed.

A framework is an effective solution to these issues; for starters, a framework can be incorporated into IDE systems as well as analysis tools which that simplify applications for new languages as well as new techniques. The framework acts like a socket for both languages and

techniques to be plugged into. A major point for this project is multi-programming language interoperability, which would allow the framework to be adaptable to any programming language, encapsulating part of the original contribution to knowledge provided by this thesis.

1.2 PREVIOUS WORK

Existing project work at Edge Hill University (EHU) has considered the application of automated SQA techniques in highly specialised fields. Work on the Quality Assurance in Climate Codes (QACC) project, currently underway at EHU, aligns well with this research. Not only does the QACC project give this research access to a large suite of scientific software, but also access to winFPT, a substantial SQA tool. winFPT is recognised internationally as a significant scientifically based SQA tool that uses an internal representation (IR) to analyse FORTRAN code, which is in alignment with this research.

1.3 PUBLISHED WORK

As part of this work, a number of papers have been published (some in collaboration with other researchers) in peer-reviewed academic conferences and workshops:

- J. Collins, B. Farrimond, M. Anderson, D. Owens and D. Bayliss, "Automated Quality Assurance Analysis: WRF–A Case Study," *Journal of Software*, vol. 8, no. 9, pp. 2177-2184, 2013.
- D. Owens and M. Anderson, "A Generic Framework for Automated Quality Assurance of Software Models: Supporting Languages of Multiple Paradigms," *Journal of Software*, vol. 8, no. 9, 13-14 April 2013.
- D. Owens and M. Anderson, "A generic framework for automated Quality Assurance of software models-Application of an Abstract Syntax Tree," *Science and Information Conference (SAI), 2013*, pp. 207 - 211, 2013.
- D. Owens and M. Anderson, "A Generic Framework for Automated Quality Assurance of Software Models - Implementation of an Abstract Syntax Tree," *International Journal of Advanced Computer Science and Applications (IJACSA)*, vol. 5, no. 1, 2014.

1.4 ARGUMENT

There seems to be a fundamental flaw in the development of automated analysis of code for quality assurance purposes. All programming languages developed in the procedural paradigm use the same theoretical basis for automated quality assurance techniques, although implementations are mostly language-dependent. If all these techniques are repeatedly implemented in different programming languages, then why can a common ground not be made to allow these techniques to be language-independent? With the use of an internal representation into which all procedural source codes can be converted, the techniques could then be independent of these languages, eliminating the need for reimplementing of each technique. Automated analysis is used heavily in both academic and industrial settings, especially in cases where the system under testing is safety critical, e.g. nuclear systems or real-time avionics / simulation avionics. Because of the extent to which this research must spread, the process must be open, using or establishing standards wherever possible. An open approach will allow for the research to extend to its fullest possible application and promote the inclusion that must take place to counteract the current method of automated quality assurance that has been implemented into industry.

1.5 SCOPE

It is important for this research that an unambiguous scope is outlined in order to allow this research to be assessed accurately and not allow tangents to become the focus. The major theme of this thesis is that automated SQA techniques should be applicable to any programming language. As SQA and programming languages are wide and disparate areas, it is important that a baseline is created to build upon. The baseline for this research is the programming paradigms that are widely used for development, these being object-oriented (OO) and procedural/imperative. The purpose of assessing these two paradigms as a baseline and not one paradigm is due to the constructs being closely related. Other paradigms being considered are logical and functional in a theoretical capacity for applicability and extensibility. This baseline must be tested; therefore, development of this framework is key. When considering this development, a subset of programming language features that span over the selected paradigms

are the focus; therefore, the developed framework will be a skeleton version, converting a subset of the Java language into an internal representation.

The focus of automated SQA techniques is also required, due to the number of reported techniques. Therefore, a taxonomy of OO and procedural/imperative automated SQA techniques will be developed. This taxonomy will allow a subset of each category to be selected for implementation on the skeleton framework, allowing for generalisation of the testing results to a wider range of SQA techniques. Finally, to suggest that this framework is truly generalised, a theoretical discussion of the other major paradigms will take place, considering the compatibility in regards to the framework.

1.6 AIMS AND OBJECTIVES

It is important that the aims and objectives of this research are clearly outlined. These will provide the basis of all activities during the research. The aims along with the scope will create an outline discussing precisely what this research will accomplish. Aims will be given the code '**Ax**', and each objective related to that aim will be given the code '**AxOn**' where '**x**' and '**n**' are integer values. Each aim clearly states the goal of each distinct phase of the research that could be described as exploration of the area and idea behind the research, development of proof of concept, theoretical implications, and scope for development.

A1 – Outline a framework based on procedural and object-oriented paradigms.

A1O1 – Review current definitions for programming paradigms and outline the progression of this research based on their individual attributes.

A1O2 – Review current uses and development of SQA.

A1O3 – Review current internal representation for source code.

A1O4 – Decide on how analysis will be applied to the internal representation.

A2 – Develop a skeleton framework.

A2O1 – Generate the taxonomy to reduce the required number of automated SQA techniques required to test the framework feasibility.

A2O2 – Construct a working skeleton framework.

A2O3 – Assess the framework based on implemented automated SQA techniques, against calculations and results generated by other automated SQA tools.

A3 – Expand the footprint of the framework by discussing the theoretical inclusion of other programming paradigms.

A3O1 – Functional paradigm discussion.

A3O2 – Logical Paradigm discussion.

A3O3 – Noteworthy and additional Paradigm discussion.

1.7 METHODOLOGY AND METHODS

This work is based on the experimental method of research, which is used for testing hypotheses derived from a conceptual framework [2] through controlled experiments. The experimental method was chosen to secure the foundations of this research, providing empirical evidence to support the claims made by the proposal for the framework. Although this method could not provide evidence for all of the claims made in this study, specifically those depicting expansion into further paradigms, the foundation requires proof so that any further claims have a foundation of relevance to support them. Using a theoretically based method alone would not allow for the generation of empirical evidence, because the initial framework did not exist, it would have made the research more prone to criticism or dismissal. A further overview of the individual methods to be used within the research is given in order below.

The initial stage of this research consists of an exploratory study, which is defined as exploring a problem or situation [3]. This is the technique used to perform qualitative research related to secondary data, such as literature. Qualitative research is recognised as being prone to bias; this will be managed by all qualitative findings being approved by the supervisory team.

This study will facilitate Aim 1 and each of the Objectives. Aim Two, Objective One will require a further explanatory study, defined as identifying relationships between two or more *things* [3], facilitating the creation of taxonomies used to design the framework.

Aim Two, Objectives Two and Three require a built framework; therefore, the methodology for developing this skeleton framework is evolutionary, which can be described as an iterative method that has initial requirements. However, through the process, the critical areas of the framework will develop, essentially repeating the development process in an iterative manner until the framework is completed [4]. This developmental method is being used because the methodology allows for the development of requirements during the building process. This method requires that there be an initial foundation of knowledge to instigate the development, meaning that as the research progresses, if any changes occur in the area or if new information becomes relevant, then this build methodology can incorporate and adapt. After the implementation of the framework, quantitative data is produced via tests, in order to evaluate the framework and consequently the QA techniques used, which will be used to cover Aim 2 Objective 3.

Finally, a further exploratory analysis will be used to facilitate the completion of Aim 3 and each of its objectives.

1.7.1 Epistemology

This research will be based on the Positivist Epistemology [5] that seeks to generate observable evidence for its claims, therefore requiring the production of a framework to generate some results in order to support its validity. If this framework was not supported with evidence in this research, its claims would not be scientifically accurate and therefore could easily be refuted.

1.7.2 Requirements Gathering

The requirements gathering stage will be the qualitative analysis of secondary research, such as literature (A1), and analysis of tools that automate software SQA. The research at this stage will facilitate the creation of a taxonomy of SQA techniques (A2O1). These will then be assessed, and generic techniques will be established to be incorporated in the framework (A2O3).

1.7.3 Taxonomy

As described in Aim Two, Objective One, this research will propose a taxonomy to substantially reduce the number of automated quality assurance techniques that need to be implemented to justify proof of concept and evaluate the framework. ‘A taxonomy ... is a way of structuring your data, your information entities, and giving them ... semantics’ [6]. This taxonomy will classify quality assurance techniques in a novel format, an approach that will classify techniques into the tree structure based on underlying requirements of that technique. This will imply that all subclasses of a particular superclass will use the same representation or methods to allow the technique’s implementation. It must be remembered that ‘Each information entity is distinguished by a distinguishing property that makes it unique as a subclass of its parent entity’ [6]; in this case, the subclass’s distinguishing property is what it uses its superclass’s method or representation to achieve. Due to this taxonomy, only a select few techniques must be implemented from each category in order to infer that all techniques held within that category are feasible under the framework.

1.7.4 Design and Implementation

The design phase of this research will consist of working out a method of implementing the key services provided by the framework; these key services are the language independence via the taxonomy on languages and the secondary research. Another key service is the design-enabling techniques of QA to be implemented within the framework (A2O2).

1.7.5 Evaluation

The evaluation technique that will be used to achieve Aim 2, Objective 3 has the results generated from the techniques implemented in the skeleton framework compared with results of other SQA tools, i.e. a metric should have the same value when used to analyse the same code, or reports from data-flow analysis should be the same. This should provide the foundation for an evaluation of the framework based on these comparative results. Where a direct comparison cannot be drawn, a calculated result shall be used in its place.

1.7.6 Exploratory Study

The final section of this research, to achieve Aim 3, is to explore the programming paradigms not included in the framework, e.g. Logical and Functional, and review their compatibility with the proposed framework to achieve a theorized full programming language generic framework.

1.8 ORIGINAL CONTRIBUTION

The current state of affairs with regard to Quality Assurance tools and techniques is that there are many techniques for quality assuring various programming languages, some of which obviously overlap and some which are specifically for designated programming languages or programming paradigms [7]. These techniques are implemented in numerous ways, generating reports, highlighting code, or editing source code directly.

Currently, there are techniques that fall under different categories or definitions, but there is not a standard method of determining in what way a technique is implemented and whether other techniques use this method. This leads to the first of the original contributions to knowledge of this work:

Taxonomy of Quality Assurance techniques and tools

This taxonomy will be created through an in-depth analysis of tools currently designed to implement some form of Quality Assurance and automated quality assurance, allowing these techniques to be categorised according to implementation.

Returning to a statement made above, each technique can be applied to a programming language or several programming languages, and this can be seen either by tools performing the same technique on multiple languages or by different tools performing the same techniques. However, a particular tool may analyse more than one language, but this is usually still limited, e.g. C and C++ or .NET languages [7], which leads to the second original contribution to knowledge:

Procedural language-independent framework for automated quality assurance

The framework will be designed to allow almost any programming language constructed in procedural and object-oriented paradigms to be quality assured based on automated techniques; this will have an internal representation that will have to accommodate many disparate programming languages. The Quality Assurance techniques will be informed by the taxonomy of quality assurance techniques created before the framework is completed. This framework will also be tested through experimental techniques that will require a bare-bones example of the framework to be implemented; this tool shall be named LIQA (Language-Independent Quality Assurance).

Finally, the current tools that do analyse more than one language are limited to a single paradigm; therefore, the final contribution will be [7]:

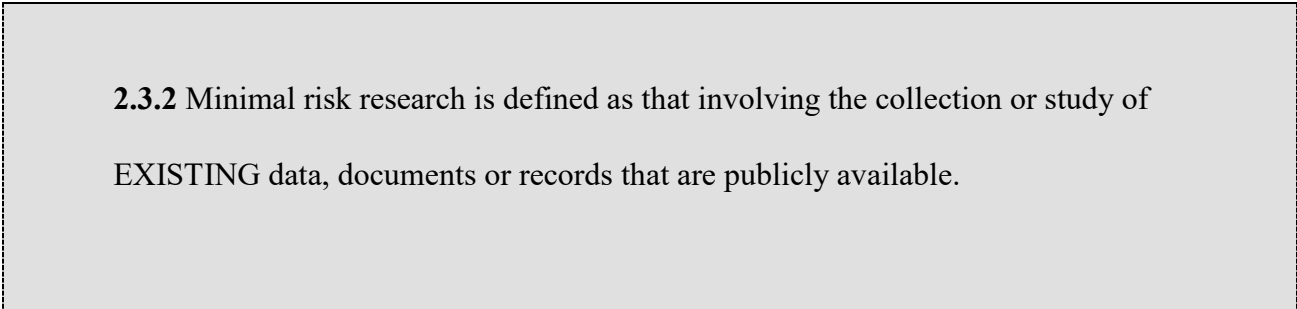
Theoretical discussion of a paradigm-generic framework

After LIQA has been tested and the framework proven suitable, further discussion of the different language paradigms will be completed, which will allow these paradigms to be included

within the internal representation used by the framework. This will result in a paradigm-generic quality assurance framework.

1.9 ETHICAL CONSIDERATIONS

As this research is based in the analysis of tools, techniques, and principles, and is far from any area that would require contact with human subjects, there are no ethical implications for this work. The research is classed as minimal risk, as decided after a meeting with the supervisory team for this research. Minimal risk is defined as follows from the ethical guidelines established by the Computing Department at Edge Hill University.



2.3.2 Minimal risk research is defined as that involving the collection or study of EXISTING data, documents or records that are publicly available.

Figure 1.8.1 - Minimal risk [8]

This project has been deemed minimal risk, as **no subjects** are used. The analysis within the research is of existing material that is publicly available e.g. QA tools. Any data that will be collected will be done so in an experimental environment. This data is collected **not using subjects**, only a comparison results from Quality assurance tools against the framework results.

Though this work has been deemed minimal risk, ethical approval from the Ethics Board at Edge Hill University is still required and was achieved before this research proceeded.

1.10 PERSONAL MOTIVATION

This section has been included to give my personal opinion and motivation for the research in an informal manner. With the assistance of John Collins of SimCon who has been in the quality assurance industry since 1988 and developing WinFPT over the last 20 years, I feel confident that this research is as up to date in this field as is possible, and also Collins's assistance has given me great insight into the development of an older programming language and how issues have developed within that. Considering this, my motivation for this research was heavily influenced by my enthusiasm for programming and working with those developers whose expertise is in another field, be that business or scientific; I do not believe that software quality should be sacrificed for those who do not develop software as their main profession.

1.11 THESIS BREAKDOWN

Chapter 2: contains the literature review, covering all areas and discussing key points.

Chapter 3: has an outline of the framework as well as the details of the implementation of the internal representation for the skeleton framework.

Chapter 4: contains the development of the taxonomy of quality assurance techniques, assessing literature as well as techniques extracted from software quality assurance tools.

Chapter 5: has the framework development of quality assurance techniques, with respect to sampling as well as testing and evaluation.

Chapter 6: contains the theoretical discussion section of the research, deliberating on the application of further paradigms to the framework and evaluating their respective constructs and quality assurance techniques.

Chapter 7: concludes the research, discussing limitations of the study, success, future research, and personal reflection on the PhD process as a whole.

Chapter 2. Literature Review

The purpose of this literature review is to evaluate software quality assurances as a whole. The complexities of each component involved in this research will be analysed. The review will identify accepted facts as well as potential gaps in knowledge. Important areas to consider are programming language paradigms as well as automated software quality assurance (SQA). Other areas include SQA assurance techniques and taxonomies related to those techniques; also included are standards and how these are built into SQA.

2.1 PROGRAMMING PARADIGMS AND LANGUAGES

As they have common features, individual programming languages can be considered under several programming paradigms; there are many programming paradigms, with four that could be considered the ‘main’ paradigms [9]. The following section will consider only these paradigms, although others will be discussed towards the end of this research. These four programming paradigms are:

Object-oriented – The most recent paradigm uses ‘objects’, rather than the conventional models. ‘Objects’ are instances of classes consisting of variables and methods as well as interactions [9]; other features that define OO are inheritance of objects and polymorphism.

Procedural –The earliest well-known programming paradigm, where a program is described in terms of statements, each of which is a sequence of instructions for the computer to perform; also based upon the concept of the *procedure call* [10].

Functional – Emphasises the use of functions, treating the program as the evaluation of mathematical functions, avoiding states and data capable of change [9].

Logical – Consists of a set of axioms and goal statements. The rules of inference are applied to determine whether the axioms are sufficient to ensure the truth of the goal statement [11].

Although these descriptions above are accurate, they do not depict the defining characteristics that separate them from one another. A starting point for this discussion is to determine what makes a programming language procedural/imperative. Imperative programming relies on the ‘state’ of a program, where state is defined by a number of variables. These variables can be changed at any point during the program, to store values relevant at that point in time. Therefore, an imperative program is one that moves from one ‘state’ to another based on what the program does [12]. This relates to object-oriented programming, as it is generally accepted that this type of programming is a development from imperative and procedural programming [13]. The difference is that although state is important, object-oriented programs are data-driven where the variables are organised into objects and these objects are manipulated [14]. A completely different approach is called *declarative languages*, which fall under two main categories, the logical paradigm and the functional paradigm. The functional paradigm works against the imperative fundamentals by being stateless, where there are no variables [15]. Instead, functions are declared based on mathematical equations where the same input will always yield the same output [13]. These functions are then built upon each other to create a program. The logical paradigm, on the other hand, is considered rule-based programming [13] where a set of relationships between values (rules) are inputs and the program generates an output based on previously defined goal statements. Although the logical and functional paradigms do not utilise ‘states’ as they are immutable, they are very different logical programming methods utilising predicates, whereas functional programs are built on top of functions.

The title ‘main’ is given to these paradigms not only because they could be considered the most popular, but also because they are considered to be the basis upon which other paradigms are built. The key concepts of these paradigms are essentially built from other programming paradigms [16], an example of which is the functional logic programming

paradigm that is based upon a combination of the key concepts outlined in the logical and functional paradigms. One approach to make the framework more comprehensive, with regards to programming paradigm, is to utilise multiple programming languages based in different paradigms; this is applied where the concepts provided by each paradigm align with the specific feature in a project [17]. Considering program popularity at this point, the object-oriented paradigm has gained success in being considered a ‘main’ paradigm, as it has become very widespread. To put this in perspective, the procedural programming paradigm was the basis for one of the first programming languages (FORTRAN) and the object-oriented paradigm has been adopted, via modification to the language, by this programming language in recent years. Again, it needs to be emphasised that though these are the ‘main’ paradigms, there are many other language paradigms targeted at specific areas, e.g. language-oriented programming [18].

A single programming language can have more than one paradigm, making it a more general purpose programming language [19]. An example of this is Java, which can be used as an object-oriented programming language but also as a procedural programming language, and more recently, Java has started supporting functional paradigm concepts. In Table 2.1.1, the languages listed may be more general purpose in nature, although they have been listed alongside the paradigm they are most commonly used for.

	Object oriented	Procedural	Functional	Logical
C				
C++				
C#				
COBOL				
F#				
FORTRAN				
Java				
Prolog				
Python				
Haskell				
Datalog				

Table 2.1.1 - Language taxonomy

The present research must outline the specific scope of programming languages to be examined, as setting out to implement all would be a gargantuan task. Instead, since the focus of this research has scientific software as a foundation, the procedural paradigm will be the starting point for the development. The second and most obvious paradigm that will be taken into account when developing the framework is object-oriented, as this paradigm is becoming increasingly popular in scientific software development as well as in software development within business. That leaves the logical paradigm, represented by the programming languages Prolog and Datalog, and the functional paradigm, represented by the programming languages Haskell and F#.

It is important to remember that in this first stage of the research only the procedural and object-oriented paradigms are being taken into account for the framework; at a later point, it will be necessary to place the ideas of this research within both logical and functional programming paradigms. However, in this instance, the most popular language paradigms must be taken into account first (e.g. Java), as well as those that are less popular but nevertheless the basis for scientific development (e.g. FORTRAN) [20].

Assessing a tool designed to automate quality assurance analysis and testing, based on which programming languages it can analyse and, furthermore, what programming paradigm these languages adhere to, is an interesting point of evaluation. From this perspective, a significant link can be seen. Quality assurance tools could be aligned with particular programming paradigms; for example, JNuke [21] and Cantata++ [22] seem to analyse object-oriented languages, e.g. Java, .NET, C++, and C. The C language, in this instance, is being viewed as an object-oriented language based on the evaluation of three other object-oriented languages being analysed by Parasoft. Another link can be made between Malpas, Polyspace, and FPT [23] [24] [25], as these tend to analyse procedural programming languages, e.g. Ada, FORTRAN, C, etc. Cantata++ [22] is the exception to the rule, which can be used to evaluate object-oriented and procedural language, which will be done in a theoretical discussion. However, this is a unit tester and it does not analyse the code, but rather only allows the automation of tests.

2.2 PROGRAMMING LANGUAGE INDEPENDENCE

Since the programming paradigms are so disparate, a look initially at procedural paradigms with object orientation shall be taken. We will review how these programming languages can be placed into a representation that is generic enough to include as many programming languages as possible, however detailed enough for both analysis and conversion back into the original programming language.

For software quality assurance to be promoted to a generic state, a distinct separation between the source code and the application of the individual techniques must be made. This can take the form of an internal representation that would essentially create the language-independent gap that would be required for this framework to be successful. Several types of internal representation have been used in the past with varying degrees of success, and these will be reviewed within this section.

Language independence, in the form of an intermediate language, has been attempted, with an example in software quality assurance such as Malpas [23]. However, there are examples of this in other environments for other purposes such as in the .Net platform, where each language supported (e.g. Visual Basic .NET, C#, etc.) is converted into the Microsoft Intermediate Language before being compiled [26]. This method of language independence has its inherent flaws, and an intermediate language can only be used in the sense that the intermediate language itself must have a paradigm and it will therefore not be able to generalise to all paradigms. Also, programming languages can have ambiguities and other possible flaws. An intermediate language does not appear to be sufficiently abstract from source code to support the level of generic characteristics that would assist this research.

Another way in which language independence could be implemented is by the use of an internal representation in the form of tokens and symbol tables. This method has been used within automated quality assurance tools but not for language independence; the tool that uses this method is FPT [25]. The internal representation may not remove all programming language-specific issues and therefore would not be suitable for programming language independence for the purposes of this research.

2.2.1 Abstract Syntax Tree

A further form of language independence could be observed through abstract syntax trees (AST) as a tree structure designed to represent code via the removal of syntax or ‘a formal representation of the software syntactical structure’ [27]. A node type depicts different code constructs such as expressions or condition statements. Similar constructs exist throughout each paradigm of software development, so it could be said that within a single paradigm, in this case procedural, ASTs from different programming languages could be the same if the semantics of the program were the same. However, the production of an abstract syntax tree involves several key influences. Depending on the programming language, the context-free grammar (which is used to define the syntax of the language [28]) can cause differences between the generated abstract syntax trees. It is also pertinent to mention that some languages are unable to use grammars to define them, e.g. FORTRAN, which was written in a context-sensitive grammar [29]. The tool used to generate the lexer and parser, which themselves create the abstract syntax trees, can be seen as another issue. The same semantics in two programs written in a programming language with similar syntax could generate completely different abstract syntax trees. Abstract syntax trees, however, still have their use because, though they are a static structure, these types of syntax trees can still represent code written in dynamic programming languages [30], such as PHP [31]. The reason why these two factors can cause issues is that there is no standard for abstract syntax trees and the amount of detail can vary dramatically.

Considering the abovementioned issues and a simple ‘Hello World’ program written in Java and C#, whose syntaxes are almost identical, the expected representations may be remarkably similar. However, if ANTLR [32] is used to generate the abstract syntax trees for these programs using grammars listed in the ANTLR repository [33], Figures 2.2.1 and 2.2.2 are generated. Though these figures are significantly different, they could be simplified into the same design with the same amount of detail.

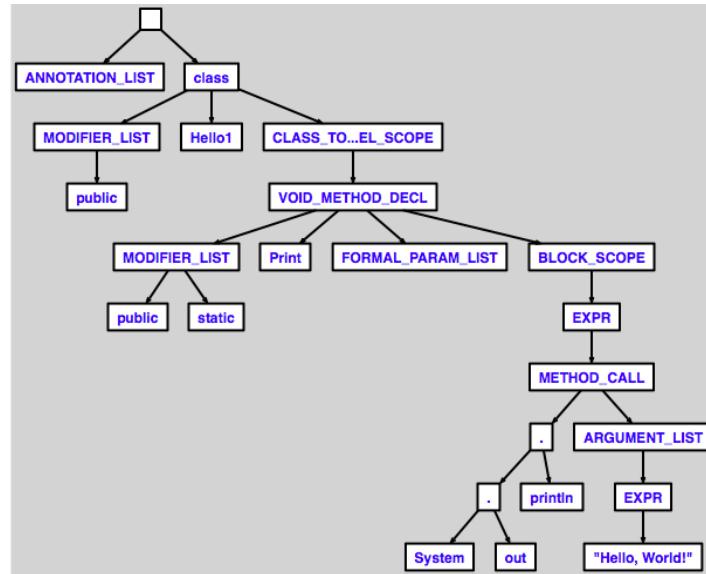


Figure 2.2.1 - Java hello world Abstract Syntax Tree (AST)

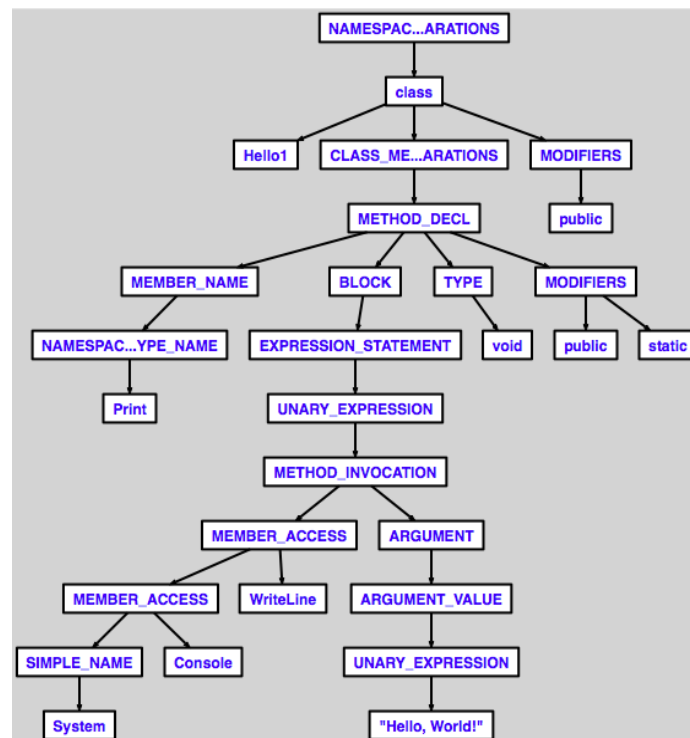


Figure 2.2.2 - C# hello world Abstract Syntax Tree (AST)

There are many uses for ASTs, especially in language-related tools such as interpreters, syntax-directed editors, document editors, etc. [34]. However, the initial development of an

abstract syntax tree lies in compiler tools, as the representation of code was built from tokens after lexical analysis of the source code [35].

There are several major techniques directly related to quality assurance that make ASTs proven to yield enhanced results rather than using the source code itself, so long as care is taken in the design of the abstract syntax tree [34]. Traversing a tree is much more efficient than parsing lines of code. Some forms of program analysis such as metrics can easily be implemented using counting of nodes rather than specific syntax-related tokens. Not having comments and disregarding layout, metrics are more reliable for comparison than when collected from source code [35].

As the above has established, abstract syntax trees have a use within automated quality assurance, but their use within language translation is key. One form of programming language translation is the production of an initial abstract syntax tree, then walking that tree and outputting using the second programming language's syntax [36] [37]. This seems relatively simple and, in the case of Java being converted to a procedure blueprint, it is straightforward [38]. Automatic program translation is another form of language translation and is similar to this, though it differs during the parsing of the abstract syntax tree. The initial abstract syntax tree is mapped to an abstract syntax tree that is defined by the outputting programming language, which is then parsed and outputted in the preferred programming language [39]. Mono utilises a similar system [40] but includes a mapping of library method calls to the secondary programming language; these library calls are similar, e.g. `System.out.println()` and `Console.WriteLine()`.

2.2.2 Generic Abstract Syntax Tree Meta-model (GASTM)

Though ASTs, as outlined above, have been used to break language barriers, a single representation has still not been achieved. Though the representation required for this research has not been achieved by previous works, there is still the possibility of implementing an abstract syntax tree with sufficiently generic nodes to cover procedural constructs. Looking further into literature on abstract syntax trees has revealed research by the Object Management Group (OMG), which provides the Abstract Syntax Tree Metamodel (ASTM). The Abstract Syntax Tree Metamodel defines the principles for an extendable structure that could represent the semantics of procedural programming languages via the removal of syntax. The ASTM was designed by the Object Management Group for language-based tools [41]. A set of core objects is defined in the Abstract Syntax Tree Metamodel documentation as the GASTM. The GASTM core objects are defined as a set of programming language-independent constructs that are used throughout procedural programming languages such as Ada, C, C#, Fortran, Java, etc. [35]. These programming languages align with the initial aim of this research, finding an intermediate representation for procedural and object-oriented programming languages that analysis could be performed upon.

2.3 SOFTWARE QUALITY ASSURANCE

Moving away from programming languages, another key area for discussion is software quality assurance and its place as a significant area in software development [42]. Figure 2.3.1 shows that testing and quality assurance lie within the engineering branch of software, and analysis could be seen to lie next to testing as an additional form of quality assurance.

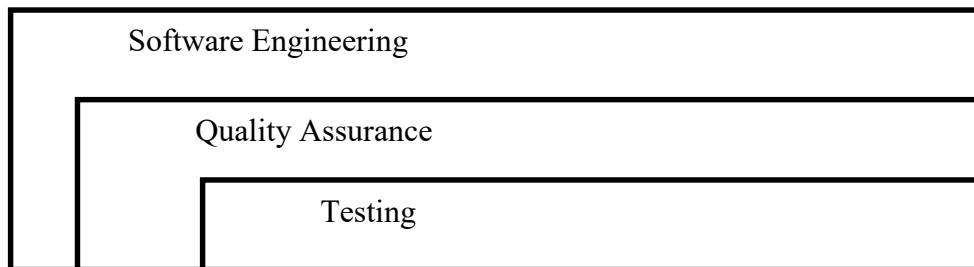


Figure 2.3.1 - Scope and content hierarchy [42]

Not only is identifying where quality assurance lies in terms of software development a theoretical issue but also a practical one, as there are varying opinions regarding when software quality assurance and testing should take place. It was common practice, at one time, that testing of software took place at the end of development, and it was considered an independent stage of the software engineering life cycle [43]. As software development techniques and procedures evolved, larger and more complex systems were created, and opinion changed about software quality assurance's position in the development life cycle. A common view concerning software quality assurance at this point was that it should take place throughout development [44].

Another key issue about which there is little agreement is how to define software quality. It could be said that software of a high quality can be based on the degree to which customer expectations are met with regard to cost and areas of functionality, reliability, availability, and supportability [45]. A different approach could be to define the characteristics of quality software, as this would replace differing opinions with a direct definition of what constitutes quality [46]. These characteristics could be identified by looking for common attributes that are striven towards in quality software [42]. Though there are those who agree with this ideal [42]

[46], the exact characteristics can be disputed. Some commonly agreed characteristics are functionality, performance, reliability, availability, and supportability [45] [47]. Some of these characteristics are contained within other lists that attempt to outline the quality of software, although others include usability, efficiency, maintainability, and portability, whilst arguing that performance, availability, and supportability are of lesser importance [42] [46]. These lists of characteristics can be also found in the standard of software quality ISO-9126, although it is argued that ISO – 9126 does not cover all software quality characteristics and that there are other frameworks and standards that need to be taken into consideration [42]. ISO-9126 could be seen as a base, whilst adding additional characteristics from other frameworks, depending on the situation, would be a better solution [47]. An example of this in practice is IBM's internal framework CUPRIMDS (capability, usability, performance, reliability, installation, maintenance, documentation, and service), which has similar and additional characteristics to ISO-9126 [42]. There are clearly reasons for such differing opinions, for example the domain in which the software is being developed. This is even more true when companies' and communities' individual frameworks are reviewed, as each prioritise differing characteristics depending on the domain [42] [47]. An example of such a community is BITS Financial Services, which has written a software quality assurance framework for financial institutions focusing on software security with significantly different characteristics such as IT risk controls embedded within core business processes, techniques, practices, and tools that identify security vulnerabilities, integrate software from third parties, and invest in the development of resilient software components [48].

Due to domain being a key factor in the choice and prioritisation of quality assurance characteristics, it is important to choose the areas on which this research should focus. ISO-9126 defines three areas on which this research shall focus: functionality, reliability, and maintainability; however, additional performance may subsequently be included. These are three critical areas that are affected by software testing and analysis, and as ISOs are recognised industry standards, it is important that this research reflects upon this. Only a subset of the criteria outlined by the ISO has been chosen. This is because scope can become too large in an area of this size[49] [42].

2.4 STATIC AND DYNAMIC ANALYSIS

The analysis of software is very important to software quality assurance because not only is it utilised to help identify areas of concern in software, but it is also used to generate reports for standards certification. There are two types of analysis, static and dynamic, both of which have their benefits and are discussed in this section.

Static analysis is the examination of source codes without execution, using formal methods and abstract implementation. The techniques that are defined under the static analysis umbrella are usually automated via the use of a built-in or plugged-in tool within an integrated development environment or software quality assurance tool [50]. Static analysis has a range of potential problems that can be identified and, in some cases, corrected, including memory corruption errors, buffer overruns, out-of-bound array accesses, or null pointer de-references [51]. Early adoption of static analysis in the development life cycle is better than later adoption or no use of static analysis. In some situations, where static analysis has not been applied, issues that could have been identified early on have been missed, and these issues can be potentially be substantial [52]. There are many examples of static analysis being applied [53] [54] [55] [56] [23] [57] , and the integration of static analysis has been incorporated in every stage of software development; automated tools have also been created to maximise the effectiveness of static analysis [21] [24] [25] [58].

Dynamic analysis is the analysis of properties of a program during execution, and is essentially derived from test plans and run in test cases, followed by an evaluation of the results [59]. Dynamic analysis has been used to perform functional, logical, interface, and bottom-up tests, to list a few [59]. As dynamic analysis is performed upon currently executing code and does not rely on abstract execution, it has the advantage of precision [21] [60]. Though static analysis is more popular, dynamic analysis has been performed in a variety of scenarios [57] [61], and listed here are multiple cases where dynamic analysis and static analysis are implemented together [21] [22] [25].

The advantages of static and dynamic analysis are clear: static analysis is an efficient process detecting defects over the entire project with reasonable accuracy, whilst dynamic analysis, due to its accuracy and lack of context, can be utilised more effectively to drill down to identify exact issues. There are significant advantages in using each form of analysis, and using either type of analysis would improve any system's quality. However, to create a more comprehensive tool, more than one type of analysis must be used [62] [61]. Taking previous research into account, this project will apply both static and dynamic analysis in order to enable the research to cover a larger number of issues within automated quality assurance.

2.4.1 Techniques and Taxonomies

What constitutes automated software quality assurance are the techniques built upon certain methods; these techniques are designed to improve a program. The 'improvement' is whatever the user thinks the software needs and can be aligned with one or more of the characteristics described above. Although the techniques apply to a characteristic, they cannot be used as a taxonomy, as some techniques overlap, causing a lack of specificity.

Though the characteristics are not a taxonomy, there are titles that developers and assessors use to categorise results of techniques usually present in reports. Sonarqube provides an example of this, presenting 7 axes of software quality. These are comments, coding rules, potential bugs, complexity, unit tests, duplications, architecture and design [63]. The issue with these categories is that they are designed for the developer to help diagnose or identify an area of code that requires attention, or the type of issue that is present. The purpose of this taxonomy is to categorise the techniques so that they use similar representations or processes to implement those techniques. This allows general statements to be made about a single category, such as whether a technique in a category is feasible to implement on this representation, in which case all techniques in that category can also be implemented upon that representation. A simple example of this is metrics, e.g. if the number of classes can be counted, the number of methods could also be counted. Metrics are also a good example of what could be a conflict with the

Sonarqube categories, as metrics can provide information concerning complexity, and can also design and identify architecture issues.

The ISO/IEC 25010 product quality model has several main categories, which are: Functional suitability, Reliability, Performance efficiency, Operability, Security, Compatibility, Maintainability, and Transferability [64]. These, again, do not match the criteria for the taxonomy set out by this research, as items such as dataflow analysis provide information that is pertinent to both maintainability and performance efficiency. These are just two examples of quality assurance that define the categories based on the type of error or issue, so that users of these reports and standards can then prioritise issues and correct accordingly. The purpose of this taxonomy is to categorise each technique into an individual area based on its implementation and implementation methods, which does not fit with the ISO/IEC 25010 categories. The ISO/IEC 25010 categories are more aligned with the purpose or outcome of a quality assurance technique, which would not work with this research, as the focus is on how a technique is implemented. Although all of the techniques included in this taxonomy could be mapped to the ISO/IEC 25010 model, the results would not mirror those in the taxonomy produced by this work.

A key part of generating a taxonomy is to have a set of criteria and aims to judge the contents against the categories. The focus in this instance is the automated SQA techniques in literature and implemented within tools such as IDEs and SQA-specific tools. To be included, the techniques must be automated and in some way inform, measure, or contribute to improving the quality based on one or more of the categories that form the various software quality standards, e.g. performance, maintenance, functionality, etc. This base component will allow the techniques to be entered into the taxonomy; however, the aim and categories of the taxonomy itself need to then be expressed. The aim is from an implementation approach; this implies that the categories will have representations or processes required for the technique to be implemented. A single example of this could be the technique detection of ‘Dead Code’ [65], which is a form of ‘static analysis’ (category one), meaning that it is implemented on non-running code. Furthermore, this technique is part of ‘Data Flow Analysis’ (category two), which

is the use of a control flow graph as a representation. The effect of this taxonomy should mean that any technique tested against the framework from a single category implies that the entire category is feasible.

The taxonomy will have to be built upon, as new techniques appear to use new methods for implementation. However, the initial review of literature (discussed in Chapter 2) produced these categories and criteria upon which to judge the techniques.

Category	Description
Static	run against source code
Static: Data Flow Analysis	requires control flow graph
Static: Metrics	requires a counting mechanism
Static: Type Analysis	requires type information
Static: Type Analysis: Pattern Matching	matches present example code against source code
Static: Type Analysis: Static Type Analysis	compares variable type information
Dynamic	run during execution of program
Dynamic: Metrics	requires a counting mechanism
Dynamic: Testing	is comparison against expected results
Dynamic: Testing: Objectives	explained in appendices
Dynamic: Testing: Levels	explained in appendices
Dynamic: Testing: Methods	explained in appendices
Dynamic: Type Analysis	requires type information

Table 2.4.1 – Initial Taxonomy Outline

These categories are loosely defined and will be refined; the goal for Chapter 5 is to end with a set of categories with explicit rules making them completely independent of each other. Something important to mention here is that testing could be considered non-automated; however, there can be automated techniques that utilise the different objectives, methods, and levels; therefore, these have been included in the taxonomy. One example of this is an automated test case generator [66]. In summary, the criteria for the taxonomy must be developed as no definitive list of implementation requirements exists without the assessment of each technique

being applied to the taxonomy; therefore, each technique will be compared with the existing set of criteria, and if no suitable category exists, the taxonomy will be extended appropriately.

2.5 THE NEED FOR AUTOMATED QUALITY ASSURANCE

A need for software quality assurance has been identified; however, automation is also an important area. Systems are being made larger and more complicated due to improvements and availability of hardware, and therefore an increase in the quality is required in order to reduce errors and improve software maintenance. Much interest and importance is understandably being attached to the quality of software [67] [68].

Some organisations have encountered difficulties when attempting to integrate what could be considered ‘popular’ quality assurance methods into their development processes [52]. Those heading the development of such systems have described testing as ‘requiring experts to tackle this complicated yet creative task’ [68]. It can be seen that an increasing number of organisations, since the millennium have either launched or are in the process of implementing more rigorous quality assurance guidelines in order to improve the quality of not only the software developed but the process of that development [69].

Most traditional quality assurance is manual testing, which is not only difficult but a heavy workload to which a large amount of time has to be dedicated [43]. Automating this process has become a priority, and the key areas that push this development are cost, time, and the elimination of human errors [43]. If the individual tester could be removed from the testing process, the output would be much more consistent and the end product would not be dependent on a tester’s reviewing skills [43].

The adoption of automated tools is significant within larger businesses, but the cost of the tools that automate quality assurance has been an issue for small to medium businesses [43]. Cost is a particularly important consideration for smaller businesses, as these businesses are

usually working on smaller projects. A trend can be identified with regards to the size of the company and number of processes that are automated. This identifies that the value of automation increases based on the number/size of projects to which these techniques can be applied [43], which justifies the initial cost of tools providing the automation.

Testing is very difficult and requires not only a lot of skill to master and, due to the size and complexity of current software systems, especially scientific software, tools have become essential [70]. Research into testing automation is of significant interest [71] but is not sufficiently developed, and there is a lack of studies and reports on testing automation [72] to cover the issues raised within the scope of this research [7]. It could be argued that reviewing how businesses are adopting automated quality assurance is not relevant as this research is aimed at scientific software development, although the trends and costs in business directly affect not only this research but the availability of tools being developed.

2.6 CURRENT DEVELOPMENT IN SOFTWARE QUALITY ASSURANCE

So far, the discussion of software quality assurance has established a foundation, the need for and application of, as well as identifying generally accepted facts. An important part of research is to cover the current landscape of the subject and identify the focus of current research. If we visualise the area of SQA based on some of the criteria discussed previously and also consider the methods used to implement the techniques, we could produce something like Figure 2.6.1.

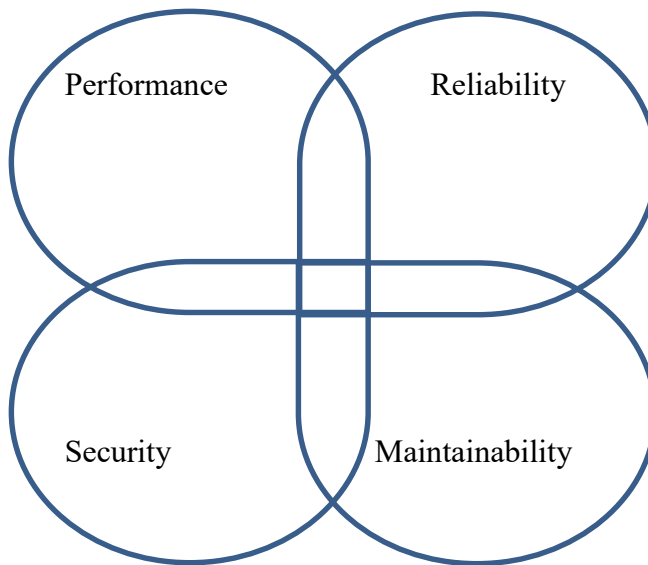


Figure 2.6.1– Software Quality Assurance (SQA) current visualisation

Each outer section is a criterion that holds its own set of techniques used to improve a program. If we considered all of the criteria, the visualisation would be 3D and each criterion would overlap with each other. What is important in this research is the centre of the model, as this would hold the methods used to implement the techniques such as static and dynamic analysis, control flow graphs, metrics, etc.

The general trend of current research is based in the techniques themselves and furthermore focused on improving their defining criteria. An example of such focus is the progression towards highly parallelisable programs (which would target the criterion performance). One technique that is being developed splits algorithms of a program into triplets that are completely independent of each other. This is achieved by using temporary variables, and essentially creating many small equations over a single algorithm. This technique would allow a program's algorithms to be run over several processors in a concurrent fashion. Another example of such progression is in both the criterion functionality and security where there is a move into automated generation of tests; one such example of this is MISTA [73]. There is also a move towards the automated generation of unit test-cases [74].

Chapter 3. Language Independent Quality Assurance (LIQA)

Outline

This chapter addresses the development of a tool representing a skeleton version of the framework known as LIQA (Language Independent Quality Assurance). LIQA shall be required to input source code and process it into an internal representation, which in theory all procedural and object-oriented programming languages will be able to translate into. On the other side, the representation must be sufficiently detailed to permit quality assurance techniques to be run against it and finally converted back into source code of the specific language. In this instance, Java will be the language that LIQA is built to process, a discussion of which is included within the chapter.

3.1 PROPOSAL

This proposal fits the Generic Abstract Syntax Tree Metamodel into a framework that processes source code into the internal representation, then utilises automated quality assurance techniques and applies them to this (see Figure 3.1.1). Discussed in the previous chapter is the possibility of using abstract syntax trees for static analysis, which is an analysis type within automated quality assurance. However, though there has been work to use abstract syntax trees for static analysis, on Java for example [75], there are difficulties that arise when attempting this. These difficulties could be due to some abstract syntax trees being very complex, and the parsers that are designed to convert source code into these structures have to be extremely sophisticated, especially considering the various quirks of different programming languages [76]. The advantage of using an abstract syntax tree, in this case the Generic Abstract Syntax Tree Metamodel, would be the language independence of this, which, as discussed in the initial argument, is a significant change and novel approach to automated quality assurance. The use of the Generic Abstract Syntax Tree Metamodel will allow language independence but at a price. As these are only the core components of procedural programming languages built into the Generic Abstract Syntax Tree Metamodel, not all constructs of a programming language may be adopted into the core components of this model. Though this will require the use of a subset of a programming language in this research, the entirety of any programming language should be

applicable if the Abstract Syntax Tree Metamodel is followed when adding additional features to the internal representation.

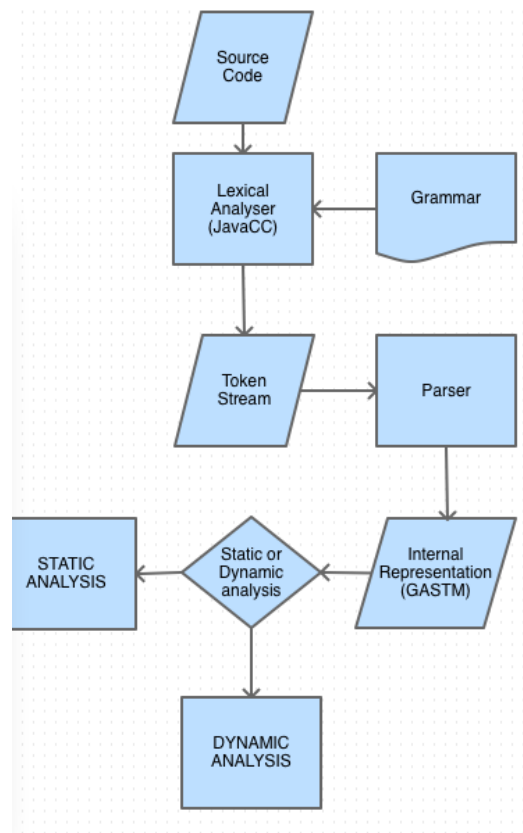


Figure 3.1.1- System data flow

Through further research, it can be seen that static analysis being performed upon an abstract syntax tree structure is a common trait [77] [78], and therefore Figure 3.1.2, which is an extension of Figure 3.1.1, requires no expansion. A similar issue, pointed out above, is still relevant here; though many tools utilise the abstract syntax tree structure before static analysis, it is unclear how much detail and what constructs are included within the abstract syntax tree for each tool and, furthermore, each programming language. It is common to see abstract syntax trees take on forms that a programmer of that specific programming language would expect them to contain [79], e.g. a Java programmer would expect a node depicting a ‘for’ loop. The novel implementation of automating static analysis in this case is the combination with dynamic analysis using a standardised generic AST, the GASTM.

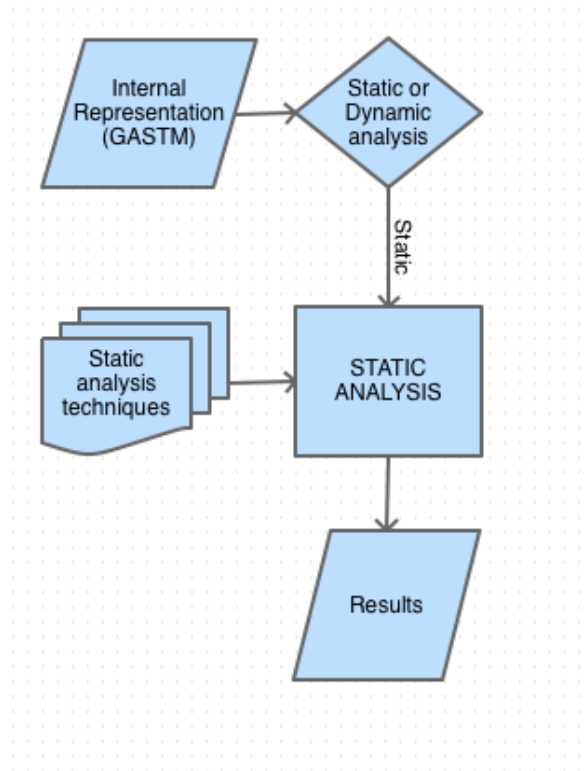


Figure 3.1.2 - Static analysis data flow

Though static analysis using representation of source code is standard, dynamic analysis via a representation such as an abstract syntax tree has not been implemented, and therefore a process to achieve this must be theorised. It could be said that automated dynamic analysis in its base form is the monitoring of properties of a program at runtime [80]. However, non-automated dynamic analysis techniques may be considerably more complex. Utilising the Generic Abstract Syntax Tree Metamodel as the internal representation, a pre-written ‘Monitor’ class could be injected into a program undergoing analysis. After parsing the abstract syntax tree for properties that need to be monitored, lines of code referencing the static methods in the monitor class are inserted, passing the data in question to the monitor class and outputting them in an interface or recording them for use later. Figure 3.1.3 is the data flow representation of this method of analysis and is an extension of Figure 3.1.1. An additional area included in the data flow representation is the conversion of the Generic Abstract Syntax Tree Metamodel into a runnable form. There are suggestions that a tool could convert an abstract syntax tree into runnable Java byte code, such as ASM [81]. These tools introduce their own issues, and adding another layer of

complexity is not advisable; therefore, the GASTM should be converted back into the original programming language.

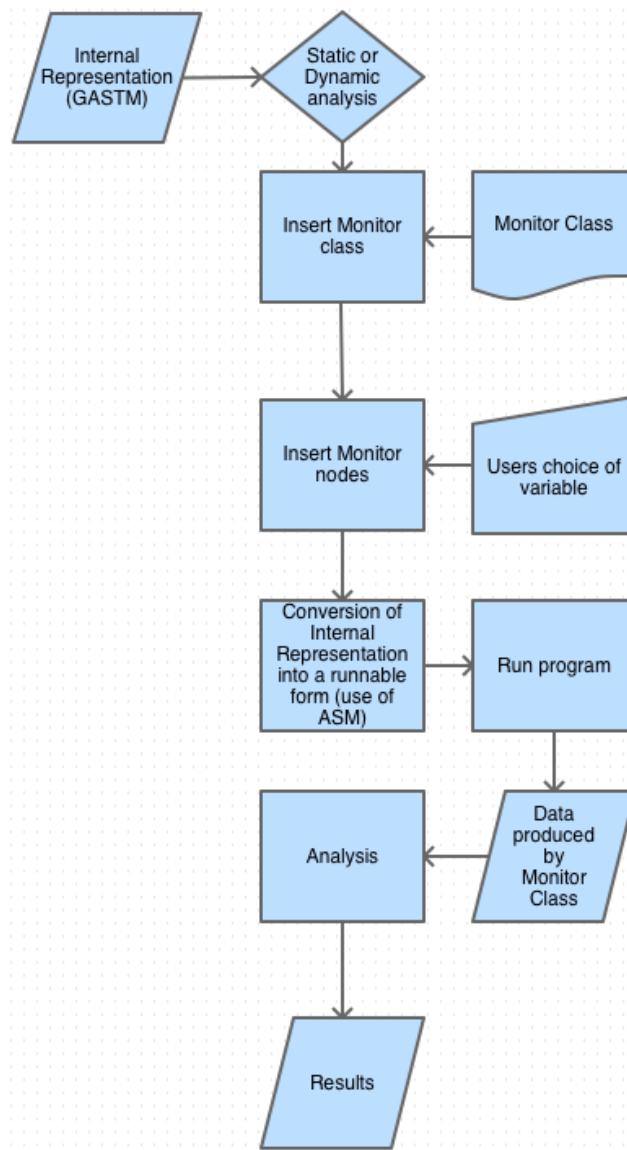


Figure 3.1.3 - Dynamic analysis data flow

3.2 DESIGN

3.2.1 Methodology

There are many methodologies that are used to develop software, although not many align well with software that is not planned out completely at the start of the implementation, or align themselves well with the significant changes that could occur when a project is based alongside research.

Use of a classical model like the incremental or waterfall models [82] would not lend itself to research-based development, as all of the requirements of the program have to be outlined and planned before the start of the project or must be introduced at the next planning phase following the completion of the current phase [82]. Agile methodologies allow for change during development to better align with the surrounding work [83], whether that be where a client has changed their mind about certain functionality of the program or, as in this case, where the research may take a different approach, as was planned due to the influence of research development. This makes agile methodologies a more suitable fit for the development of LIQA.

The methodology that will be used to produce LIQA is evolutionary [84], which will allow requirements to change as the research develops and also, unlike other agile methodologies, will allow for significant additions of functionality [85]. As the taxonomy of quality assurance techniques is developed and those chosen quality assurance techniques are assigned to LIQA, more functionality must be adopted. The evolutionary model will also allow for version control and prototyping [84] so that each quality assurance technique, and indeed the IR phase, can be prototyped separately.

3.2.2 Development Tools

There are several choices that must be made before development can begin, some of which include the operating system (OS), programming language, IDE, toolsets, and libraries. The OS is an essential factor in this research for, as the focus of this research is branching differences between languages and allowing for a generic system, the system itself must be generic and should run on a variety of platforms. Those OSs used that are widely employed for development are Windows [84], OSX [86], and the various distributions of Linux [87], and therefore would be used for quality assurance of code [88]. These three OSs therefore must be able to run the LIQA system. Some slight OS-dependent features should be permitted with regard to handling files, but this should not affect the framework overall. This factor leads directly into the choice of programming language due to certain languages only being able to work on certain OSs, e.g. Objective-C is designed for OSX and must be ported to be run on Windows [89]; or the .NET framework, which is a Microsoft development containing several languages designed for use on a Windows PCs but can be ported to Linux and OSX via the Mono framework [90]. Java is a programming language that bridges the gap between these platforms [91], and therefore would be an ideal choice for the development of LIQA. As Java runs in its own virtual machine, it is abstracted from its platform and can run on Windows, Mac, and Linux [92]. An assumption is being made, in the case of dynamic analysis, that the user has a platform that can run the project being analysed in LIQA at the time of analysis, therefore allowing LIQA to utilise the platform to run the project when performing dynamic analysis.

Now that the programming language has been established as Java, the IDE for development must be identified. In the case of LIQA, NetBeans [93] is being utilised. This may seem like an odd choice since the Modisco library [94] (discussed later) that contains the Java representation of GASTM is designed for Eclipse [107]. The reasoning here is due partially to personal preference but also because, with the Modisco library, it is simple to break down and retrieve the files required for GASTM development. However, some of the other toolsets focusing on different areas of LIQA are designed specifically more for NetBeans and would be more difficult to break down, e.g. Batik [95].

The toolsets and libraries are dependent on the language and OS decided upon above; the specific tool sets and libraries used to develop LIQA are discussed later under the Implementation phase.

3.2.3 Initial Requirements

There are few initial requirements, as this is a formal test of the IR proposed above. Only a few features are required to implement the Java-to-GASTM conversion. Initially, only the IR must be created; no analysis is to be planned at this point, as the taxonomy will infer which analysis techniques are to be implemented for testing and evaluation of the framework.

ID	REQUIRMENT	CATEGORY
R1	Allow for input of source code	Essential
R2	Convert source code into token stream	Essential
R3	Parse Token stream into IR	Essential
R4	Visually Represent the IR	Desirable

Table 3.2.1 - Initial requirements table

3.2.3.1 Form Design

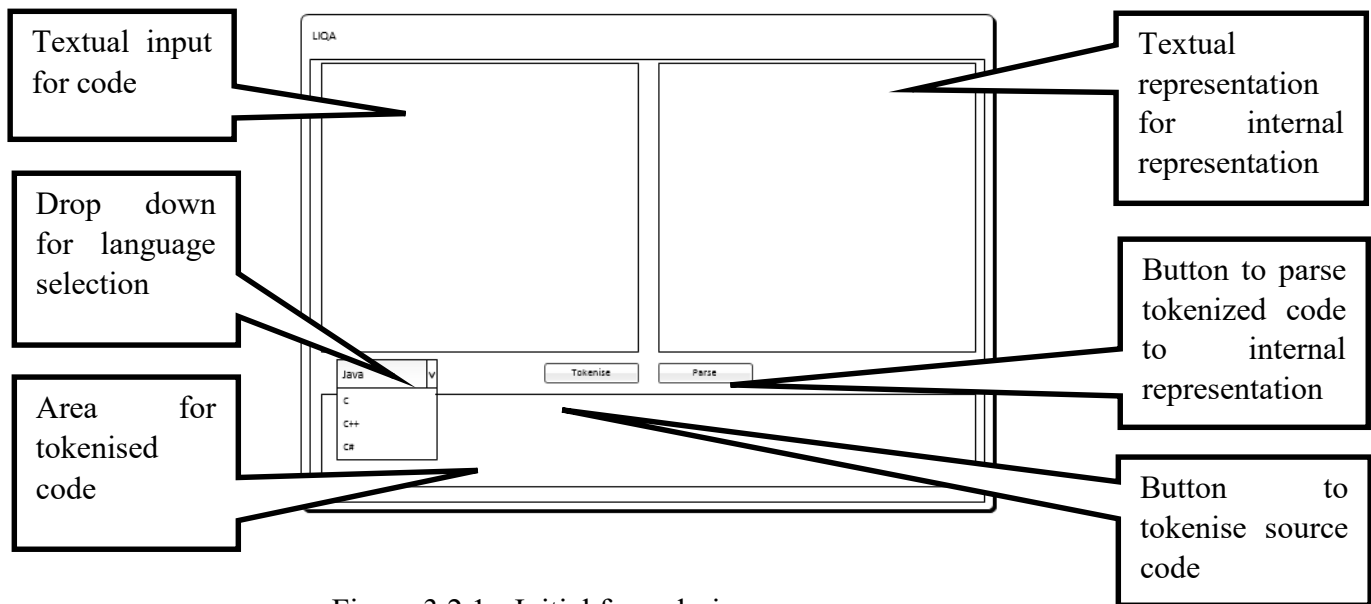


Figure 3.2.1 - Initial form design

The initial form design shown in Figure 3.2.1 was a simple design intended to illustrate the different stages of data flow required to form a GASTM IR from source code. Though only Java, at this stage, is being parsed into IR, the inclusion of language selection is intended to identify that it would be possible to include other languages within LIQA.

3.3 IMPLEMENTATION

LIQA's initial design was a simple outline for its implementation of the IR. At this stage, only the IR is of interest, as the analysis and metrics sections of LIQA will be developed for testing the techniques that are deemed appropriate after the production of the taxonomy in the later stages of this research. LIQA was developed in an evolutionary style, allowing the inclusion of different functions as they were deemed appropriate. LIQA v1.7 has included all necessary functions to assess the capability and completeness of the IR.

3.3.1 Evolutions / Version

As LIQA was being developed, certain factors arose that changed the course of the development, enabling improvement to be made on the initial requirements; this was expected and planned for, as the Evolutionary methodology encourages it.

VERSION COMPLETED TASKS

1.0	<ul style="list-style-type: none">• All initial requirements (essential) met [R1, R2, R3]
1.1	<ul style="list-style-type: none">• Removal of token stream display and integrated tokenising and parsing to one button• Added the function of browsing and selecting a file to auto-fill the source code textbox
1.2	<ul style="list-style-type: none">• Introduced the menu as a starting point to allow for separation of the program• Introduced the save and load functions (added to main menu)• Included buttons on menu as placeholders for future sections
1.3	<ul style="list-style-type: none">• Removal of textual representation of the IR• Replaced with XML output and SVG using XsdVi and Batik tools [96] [95]. This visualisation was preferred to textual output, as issues within the parsing process could be spotted more easily and corrected accordingly.• All initial requirements (desirable) met [R4]
1.4	<ul style="list-style-type: none">• ‘About’ form added to track progress and list limitations of LIQA
1.5	<ul style="list-style-type: none">• Parser for Java fully completed (with limitations)
1.6	<ul style="list-style-type: none">• Sample software metric (logical lines of code) included as example using tree walker
1.7	<ul style="list-style-type: none">• Variable monitor and method monitor, and sample dynamic analysis techniques, developed and included in LIQA as proof of concept for dynamic analysis on a generic IR theory

Table 3.3.1 - Version list

3.4 TOOLS

To simplify development, several steps were taken to incorporate tools into the development of the Language Independent Quality Assurer, aka LIQA. These tools have different purposes, some of which are not core to the functioning of LIQA; however, they serve the purpose of making LIQA easier to test. The tools used are listed below:

- Modisco - GASTM Core Model [97]
- JavaCC - Produced tokeniser for Java (Grammar from library) [98]
- XsdVi - Used to generate a .svg file from .xsd [96]
- Batik - Toolkit to visualise .svg file in JFrame [95]

The GASTM core objects have already been implemented in the Modisco library, and utilising this will reduce development time. To populate the internal representation, initially the source code must be parsed into a token stream. The JavaCC tool provided a grammar for Java, and making use of this, and JavaCC, itself generated a tokeniser. This had a small drawback, where the tokeniser removed the comments in the source code.

Modisco and JavaCC were utilised to simplify and increase the implementation speed of LIQA and therefore of the research. The additional two tools, XsdVI and Batik, were used in the development to make LIQA more user-friendly and, furthermore, to create a simpler environment to debug, consequently finding errors in the conversion of source code to internal representation. The XsdVi library was used after the internal representation object had been generated. The abstract syntax tree is then walked, outputted in a structured .xsd format, and saved to a known location. The XsdVi library is then called to convert the .xsd into a .svg format, essentially a graphical representation of the XML in the .xsd file. Following this, the Batik library was implemented into the graphical user interface of LIQA to allow the .svg file to be displayed and navigated with ease via its JSVGScrollPane and JSVGCanvas objects.

3.4.1 Modifications to Generic Abstract Syntax Tree Meta-model (GASTM)

The Generic Abstract Syntax Tree Metamodel core objects, defined by the Object Management Group [99] and implemented by Modisco [97], have been implemented, although several modifications had to be made. Due to Modisco implementing the classes but not, what they call, a discoverer, which takes source code and converts it into the model, a slight modification had to be made so that the object would be simpler to store for use later. To do this,

making several objects had to have `java.io.Serializable` implemented, and these were: `GASTMFactoryImpl`, `GASTMObjectImpl`, `GASTMPackageImpl`, `GASTMSemanticObjectImpl`, `GASTMSourceObjectImpl`, and `GASTMSyntaxObjectImpl`. The following changes were made to better represent programming languages, which is the aim of this research; these might have been missed, as the Generic Abstract Syntax Tree Metamodel is in only its first version. Classes in languages such as Java, C#, and C++ have an access modifier, e.g. `Public`, `Private`, etc., and the Generic Abstract Syntax Tree Metamodel did not represent this, so both the `ClassType` and `ClassTypeImpl` objects had an additional property added that was of type `AccessModifier`. Another modification made to the Generic Abstract Syntax Tree Metamodel is the property `IsStatic` of type `Boolean`, added to the `FunctionMemberAttribute` and `FunctionMemberAttributeImpl`. Again, this relates to programming languages such as Java, C#, and C++ in which functions can have a static modifier.

The final modification is due to an error in the documentation of the core models. In Figure 3.4.1 is provided the definitions for the Switch Statement and Case Objects.

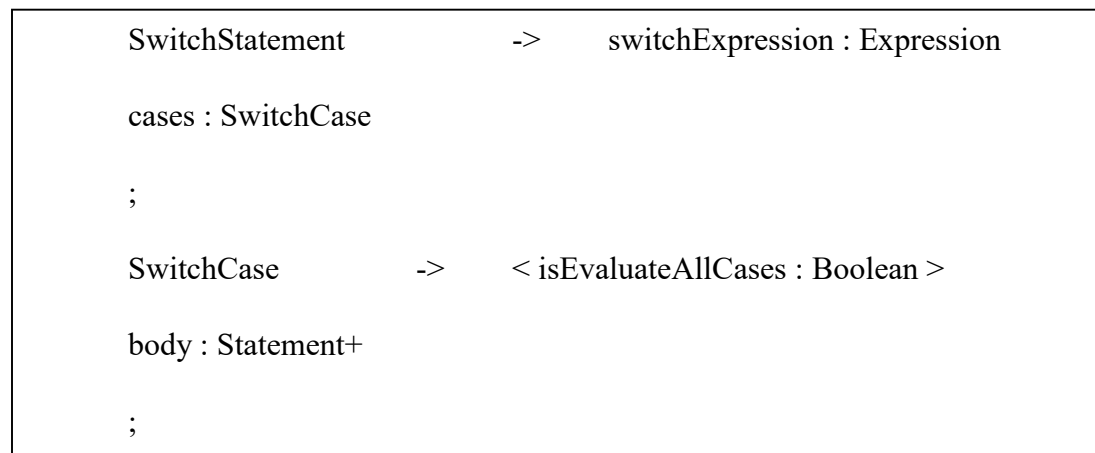


Figure 3.4.1– Generic Abstract Syntax Tree Meta-model (GASTM) SwitchStatement definition

The text on the left of the ‘->’ denotes the name of the object, and on the right, its variables. ‘=>’ depicts an object that extends the object on the left. The variable’s names are on the left of the colon and its type is on the right; a ‘+’ after the type states it is a list of 1 or more. Missing from this definition is a plus next to ‘cases:SwitchCase’ to define multiple cases within a single switch; this has had a role on effect, as the GASTM model developed by Modisco represents switches with a single SwitchCase; consequently, this had to be modified.

3.4.2 Limitations

It is expected that there will be limitations not only with the Generic Abstract Syntax Tree Metamodel, as it is in its first iteration, but also with LIQA due to its rapid development and significant size.

As LIQA is only a proof of concept and does not have to be a commercial product, the scope of the related study has to be limited. As a result of this, LIQA cannot handle multiple files of source code. This is only a limitation of LIQA and not the framework, as multiple files could be handled so that an entire project/program could be inputted.

The Generic Abstract Syntax Tree Metamodel has a few limitations; as it is only a set of core objects, it is not expected that it would cover all possibilities within the Java language. The following operators do not have objects to represent them in the Generic Abstract Syntax Tree Metamodel: <= , >= , += , -= , /=, and *=. However, in this case, if it were desired, these could be represented with an object defined with specification acquired from the Abstract Syntax Tree Metamodel. An alternative would be to preprocess the source code and modify the code where these operators are used to simplify their representations using basic operators, e.g. ‘x += 1’ = ‘x = x + 1’. In the case of LIQA, the lack of importance to include these operators meant that they will be excluded from the implementation.

Another Generic Abstract Syntax Tree Metamodel issue that was identified during a discussion with peers working in this field is the non-support for scripting languages. Though scripting languages are not discussed in the ASTM documentation [99], it is clear through the core models outlined that scripting languages may not have been an included form of programming. This is clear because at the highest level, the 'Project' object has one or more 'CompilationUnit' objects. Through this definition, we can see that interpreted languages, such as scripting languages, were not an included form of programming language within the Abstract Syntax Tree Metamodel. At this juncture, it can be said either that scripting languages are out of scope, or that a way of including scripted languages can be considered. In this case, considering that the scope of this research does not include testing of a scripting language, a theorised implementation would be appropriate to substantiate the claims of this representation. Accordingly, it could be identified that the main issue of including a scripting language is that a file could contain no class but could contain functions and statements; this is not supported by the Generic Abstract Syntax Tree Metamodel, but could be with only a small modification. If the name of the object 'CompilationUnit' is ignored (or an alternate Object made with the same attributes, e.g. 'InterpretedUnit'), the progression is to its contained list of fragments, which is a list of type 'DefintionObject', which has the subclass 'TypeDefintion', which has the subclass 'AggregateType', which has the subclass 'ClassType', which is fine, but this would not allow statements to be saved in a file without a class. The way to correct this is to have a 'ScriptType', which is a subclass of 'TypeDefintion' of which this 'SciptType' has a body that consists of one or more 'Statements'. This handles the issue of not having a container but also allows functions to be written through the 'DeclarationOrDefinitionStatement', making a seemingly perfect move to include scripting language.

The following limitations are due to time constraints and though they could be implemented within LIQA, they are not necessary for testing the framework at this stage:

- Operators that are not implemented are ‘?’ and ‘!’
- List types are not implemented, e.g. ‘List<String>’
- Re-type casting has not been implemented, i.e. ‘String str = (String) x;’
- The assignment of arrays via block statement has not been implemented, e.g. ‘int[] x = {3,2,1};’
- Inline if statements have not been implemented, if statements must have a block containment i.e. ‘if (condition) statement;’ is not supported and ‘if (condition) {statement}’ is supported.

[100]

One final subject to discuss that could cause some challenges with regards to reporting issues within the source code, is the inability of LIQA to report the line and token in which an error has occurred. This is because the Generic Abstract Syntax Tree Metamodel within LIQA has not had the location object implemented as a property in most of the objects under this metamodel. This, with hindsight, would have been an ideal way of linking any issues back to the source code; however, this is not the case for LIQA. It would be considered vital to add this feature to a reworked version of the framework.

3.4.3 Test Dynamic Analysis

Implementation of the monitor theory discussed above, to allow dynamic analysis to be used on the generic IR of LIQA, has yielded some interesting developments. The theory is sound, as it is possible to monitor different properties of a program using generic ‘monitor’ class calls and to send information through as parameters. However, this is limited by several factors. Firstly, to run any program that has been parsed into the IR, it must be, in most cases, outputted back into the original programming languages, the few exceptions being when a program is fully independent of libraries (that is to say, no language-specific library calls are made). Because

language-dependent libraries are used in almost every program written, this limitation must be overcome. A tree walker designed to output the IR in a target programming language could be written. This may seem like a large task, but the most difficult part of including a programming language in LIQA is writing the parser to convert the source code into the IR. Writing the monitor class and tree-walker are significantly simpler tasks. A second issue is that the monitor class is likely to require stack traces to achieve more complex dynamic analysis techniques. This will limit the range of languages that can be included within LIQA, although the simpler dynamic analysis techniques, such as simple profilers and variable monitors, can be achieved without stack traces and with only the generic ‘monitor’ method calls.

3.4.4 Test Static Analysis

A static metric, Logical Lines Of Code (LLOC), was implemented as a proof of concept to assess whether some forms of static analysis were possible, when applied to the IR. This was fully successful, and it is predicted at this point that most forms of static analysis should be applicable to the IR; those that are not will be identified in the taxonomy.

3.4.5 Overall System Description

Over the development of LIQA, several factors from the initial proposal had to change to account for new information acquired as a result of the research. The ideas presented in the proposal were sound, and only small modifications were required, such as including a library of monitor classes rather than just a general monitor class represented in the IR to account for language-dependent method classes. The other modification made was the inclusion of the conversion of the IR into a Control Flow Graph (CFG), a representation required for data flow analysis. Both changes have been made, and the modified DFD of the overall system is shown in Figure 3.4.2.

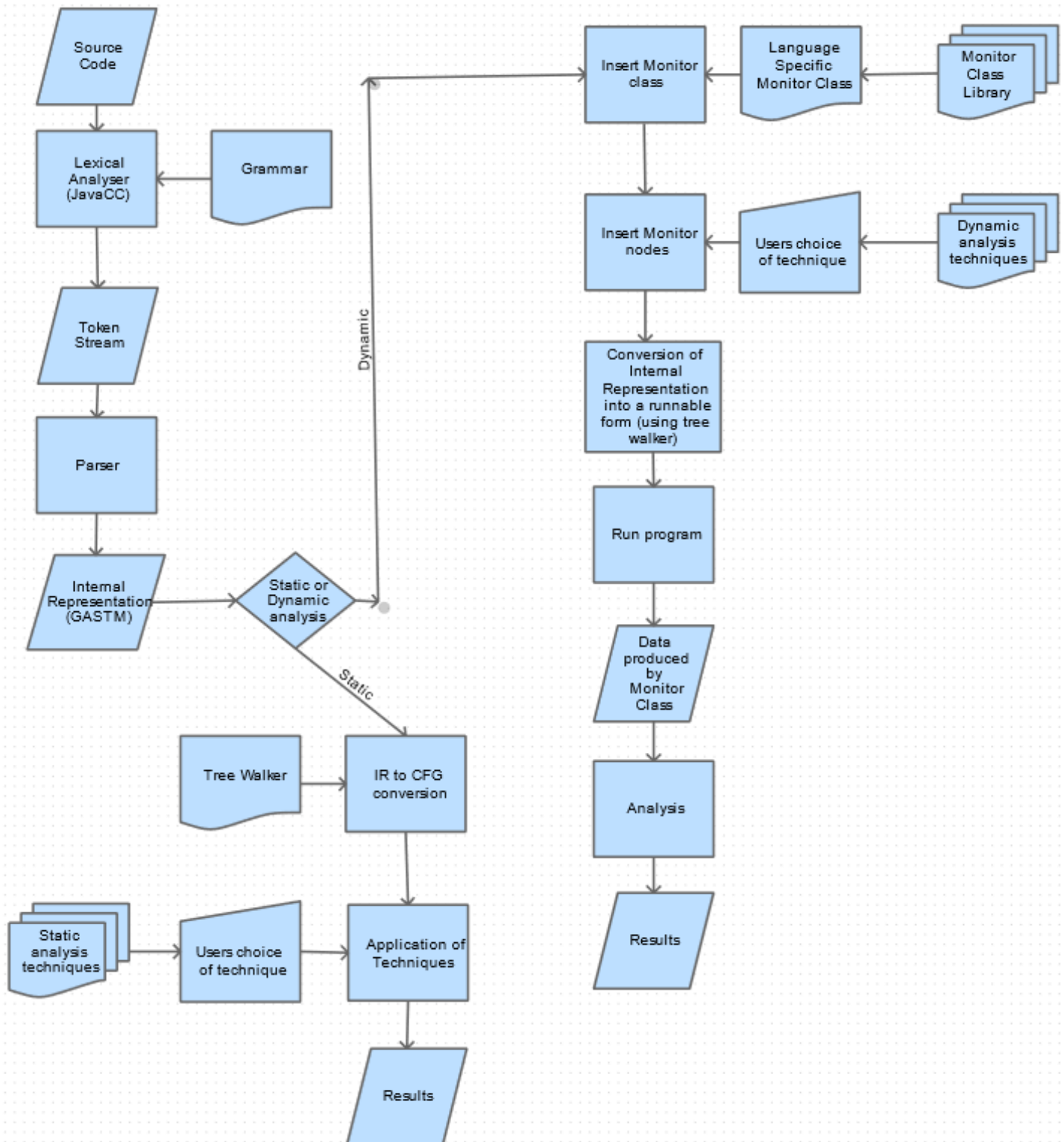


Figure 3.4.2 – Language Independent Quality Assurance (LIQA) expected data flow diagram

3.4.5.1 *Development Discussion*

The development of LIQA was an extensive and exhausting research effort. The agile nature of the evolutionary methodology had a significant impact in decreasing the development time; however, a slightly more structured approach may have had its benefits. An example of this could be seen in increasing the planning time for each iteration, resulting in not only a more robust approach but also resulting in a more maintainable code base with better documentation. Contrary to this was the development itself being used as a point of understanding the complexities of the Java language, which had an impact on the ability to plan in the first place.

The major components, with the exception of the GASTM, have been broken down into packages related to their purpose, including GUI, IR (with IR.Token), LIQA, etc. The GUI package (standing for graphical user interface) contains all of the JForms used for LIQA, which was developed from a single initial form into five separate forms to allow for simpler debugging. However, an official build of the framework would also require more than a single form, as some techniques, such as variable remaining, require user input and selection. The IR package contained classes and a sub-package called Token, of which Token has a sub-package of Java. The Java packages contain classes that implement the tokenisation of the Java programming language gathered from JavaCC [98]. If other programming languages were implemented, these would be contained in their own package here. The class in the Token package contains constructs to direct tokenisation based on the source code and programming language selected. The classes contained in the IR package are builders of two types: a single generic builder, to direct flow, and programming language-specific builders to take tokens and form the internal representation. The package LIQA is essentially LIQA-dependent code, i.e. the initialiser for the program as well as a project information store. All of this can be seen in the file structure shown below:

- GUI
 - GUI_About.java
 - GUI_Dynamic.java
 - GUI_IR.java
 - GUI_Menu.java
 - GUI_Static.java
- IR
 - Token
 - Java (+)
 - TokenGenerator.java
 - IRBuilder.java
 - IRBuilderJava.java
- LIQA
 - LIQA.java
 - Project.java
- QA
 - DynamicAnalysis
 - Processes.java
 - Identifiers
 - CFG
 - CFGblock.java
 - CFGdfi.java
 - CFGgedge.java
 - CFGproperties.java
 - Dynamic
 - Tag.java
 - My
 - My.java
- MyClass.java
- MyMethod.java
- MyVariable.java
- StaticAnalysis
 - Metrics
 - MetricHalsteads.java
 - MetricLLOC.java
 - Patterns
 - Netbeans
 - PMSysouterr.java
 - PMfinalclassandmethods.java
 - PMfinalmethodinfinalclass.java
 - PMmethodparameters.java
 - PMmethodprivatefinal.java
- VisualStudio
 - PMparamarray.java
 - PMvarcase.java
- FortoUnroll.java
- SwitchtoIf.java
- TreeWalkers
 - Modifiers
 - CodeManipulation
 - ControlConstructs
 - ConvertLoop.java
 - ConvertSwitch.java
 - ReplaceCalls.java
- ReplaceLoop.java
- RplceSwitch.java
- Dynamic
 - DynamicMethodCallCounter.java
- VariableRename
 - RenameCalls.java
 - RenameClasses.java
 - RenameMethodVariables.java
 - RenameMethods.java
- Copy
 - CopyNode.java
- Outputs
 - IRtoCFG.java
 - IRtoJAVA.java
 - IRtoXSD.java
- Retrievers
 - Declarations
 - GetClassVariables.java
 - GetClasses.java
 - GetLoop.java
 - GetMethodVariables.java
 - GetMethods.java
 - GetSwitch.java
 - GetHalsteadsOperands.java
 - GetHalsteadsOperators.java
 - GetVariableUse.java
- Org (+)

Figure 3.4.3 – Language Independent Quality Assurance (LIQA) file structure

It is important to point out that the classes containing quality assurance techniques are very similar in design. The class will have one main method for input; this will accept several parameters specific for that technique and the GASTM representation of the source code. The technique will then be performed and produce its result. This allows each technique to be separated and debugged individually and would also mean that if other techniques were included they could be implemented simply using this approach without affecting other techniques or the

overall program. Similarly to this the converters are built with modularity in mind being store in the Outputs package.

Several class diagrams describing the relationship for other parts of the program have been included, as well. An overall class diagram would be impractical to include due to its size. Again, due to the size of the program, classes have been represented without their fields or methods.

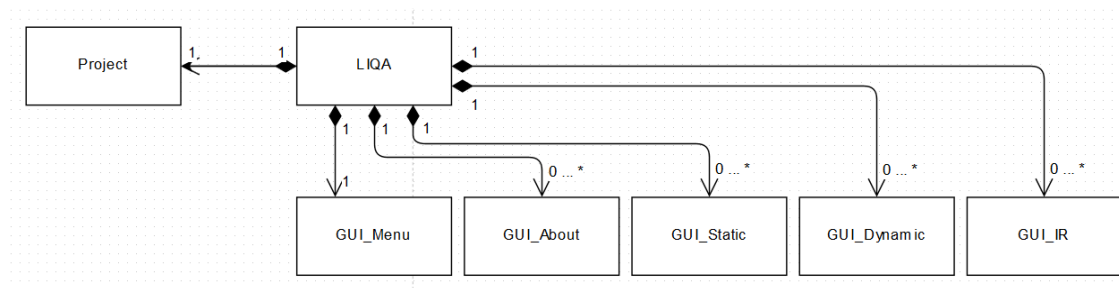


Figure 3.4.4 – Language Independent Quality Assurance (LIQA) class diagram Graphical User Interface (GUI)

To keep the state of the program managed, the LIQA class controls the GUIs so that regardless of the number of instances of a particular GUI that are in use, they all affect the same data.

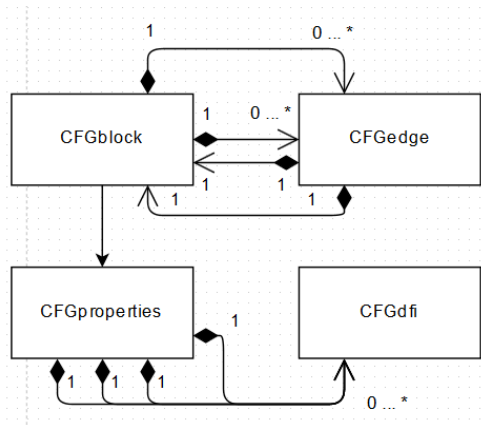


Figure 3.4.5 – Language Independent Quality Assurance (LIQA) class diagram Control Flow Graph (CFG)

The control flow graph (CFG) is a representation used by dataflow analysis techniques and, simply put, each CFGblock holds statements until a decision has to be made, and then CFGedges are used to branch to other blocks of statements, hence the complex relationship. CFGdfi is used just to store block IDs for quality assurance techniques to use, and CFGproperties is used to compute and store lists of GEN, KILL, IN, and OUT. This will be discussed in more detail in the quality assurance techniques chapter.

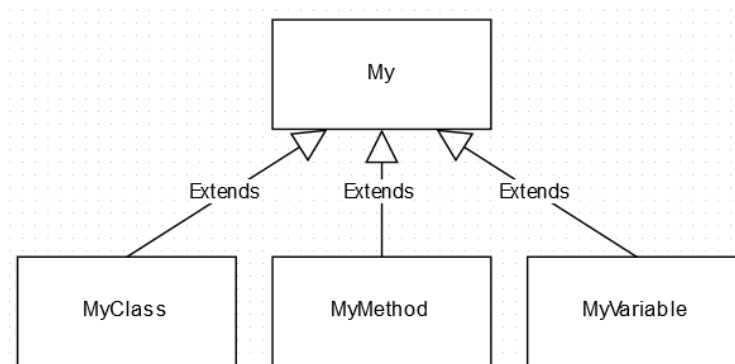


Figure 3.4.6 – Language Independent Quality Assurance (LIQA) class diagram identifiers

The class diagram in Figure 3.4.6 shows a small OO-based approach to identify different declarations used in various techniques. These were stored in the QA package, meaning they were used on several techniques in a specific area; one example of the use is renaming.

IRBuilderJava and IRBuilder interact in an interesting way; IRBuilder gets given a token list and programming language. This passes the token list to the Builder for that language, in this case Java. IRBuilderJava then processes the tokens and pulls out constructs and relevant data; this data is pushed back into IRBuilder, which creates the GASTM object using GASTMFactoryImpl. The GASTMFactory Impl is just an object used to generate the GASTM object, which then can be populated. The relationship looks something like this:

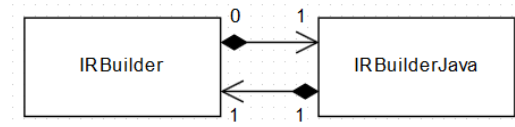


Figure 3.4.7 – Language Independent Quality Assurance (LIQA) class diagram Internal Representation (IR) builders

One of the improvements of having a more structured iteration would allow for a better written parser, as the current one required reworking several times when bugs were identified in development. Outlining and planning of this section of code would have decreased development time and increased the maintainability of the code. Considering that the parser in particular required a lot of exploration of the Java programming language, a more structured approach than the one used may have been difficult.

At this point, LIQA only contained the test techniques, and the quality assurance techniques that will be implemented are still to be identified.

- Total Lines of Code: 72,382
- Total imports: 1,867
- Total methods: 3,126
- Total classes: 231

After the addition of the techniques that are discussed and implemented in later chapters, these metrics changed dramatically:

- Total lines of code: 96,632
- Total imports: 3,066
- Total classes: 263
- Total methods: 4,158

These statistics are significant, as they provide an insight into the sheer size of the project, especially with the decompiled Modisco library that had to be navigated and modified.

- Approximate Written Lines of Code: 33,160
- Approximate Written Classes: 48

It is noted that LIQA's own code is small in comparison with the complete project; however, is still is of significant scale. They are estimated conservatively, as some decompiled and imported classes were modified but not counted; also, any class that included auto-generated code such as the GUI JForms were exempt from the calculation.

- Average cyclomatic complexity: 3.57

This shows how well the Modisco library is written, as the code for LIQA is raising this figure, purely as the decision tree for the parse alone has to be of high cyclomatic complexity.

3.4.5.2 Finalized IR Interface

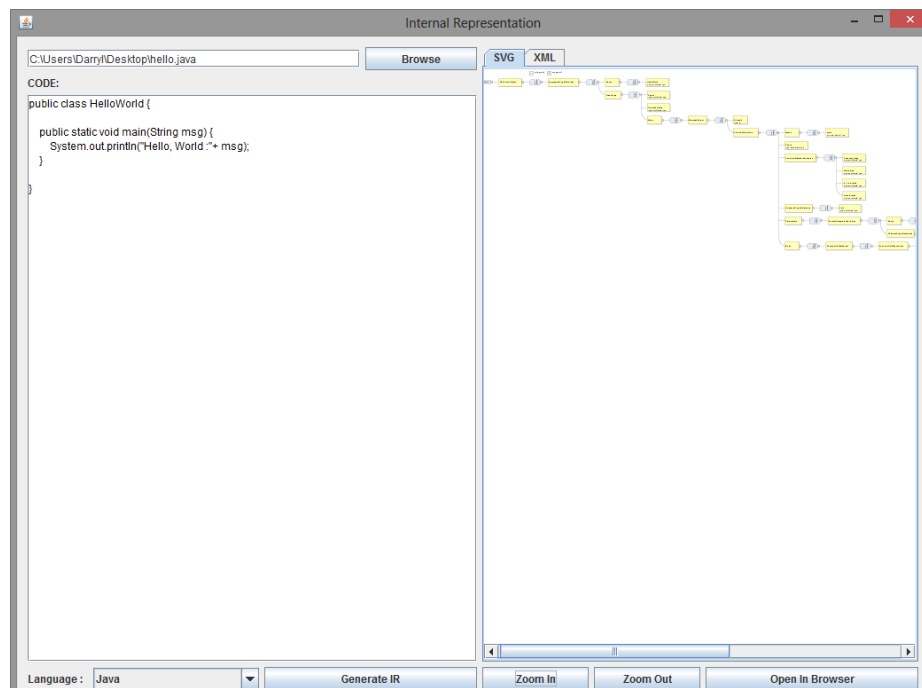


Figure 3.4.8 - Internal Representation (IR) generator form

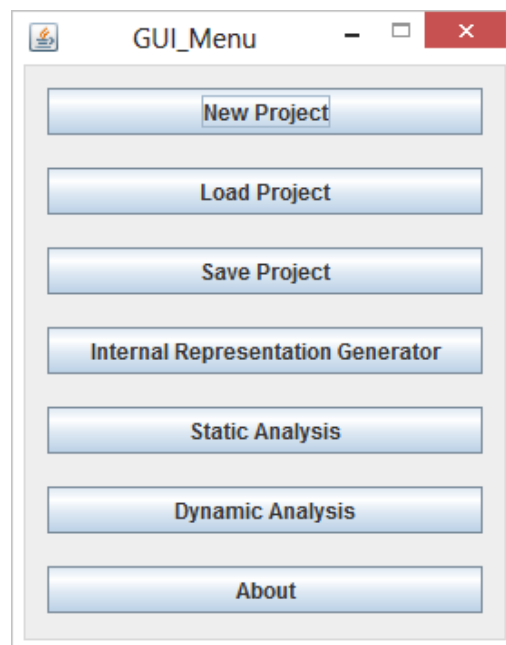


Figure 3.4.9 - Menu design

Figure 3.4.9 shows a finalised interface has been included. Although the design does not play an important role in the project, it does demonstrate the separation of the different components as well as additional functionality that had to be added to make LIQA easier to use. This includes project management, e.g. saving and loading of projects. The major components can also be seen in the menu, specifically the IR generator, Static analysis, and Dynamic analysis.

3.5 MODIFICATIONS TO RESEARCH

Due to the development of the IR using the GASTM core model, a change can be made to the scope of the research. Instead of implementing multiple procedural languages to test the IR, the GASTM core model is stated to represent most procedural languages, e.g. C++, Java, Ada, C#, VB, VB.Net, C, COBOL, FORTRAN, Jovial, VMSVAX BASIC, RDBMS, PL/1, JCL, etc. [99]. Because of this, the research can progress to the next stage, which involves the development of the taxonomy of quality assurance techniques.

3.6 SUMMARY

This chapter was designed to present an overview of the internal representation used in the framework and further demonstrates the skeleton build of the framework LIQA that was used to validate and evaluate the framework at the end of this research. Other aspects that this chapter aimed to point out was the sheer size and complexity of just a simple implementation of the framework and the modifications to the internal representations that were required to build LIQA.

This chapter sits within the research as the description of one of the main components, the internal representation, and not only its theoretical implementation but its practical use. Following this chapter, the focus shifts onto the other end of the framework, the software quality assurance techniques, and how these can be applied to this framework to demonstrate its broad capability.

Chapter 4. Taxonomy of Quality Assurance Techniques

To evaluate which quality assurance techniques are available and to assess which could be sufficiently generic to apply to this research and framework, several studies must be completed at varying levels. These studies will result in a taxonomy that shall be used to evaluate the framework. The resulting taxonomy will be categorised by a method of implementation; therefore, categories of techniques can then be generalised as applicable if one or more can be demonstrated as working within LIQA.

The automation of quality assurance on software has its obvious advantages, reducing completion time for software development by reducing the number of errors and minimising errors before the manual testing phase. There is a large variety of testing software and toolkits available to implement automated software quality assurance, and some trends can be drawn from simple analysis of these tools. Toolkits that target multiple programming languages do so on a small scale, i.e. 2 – 5 programming languages of the same programming paradigm. Another trend is that tools that do analyse a significant number of programming languages usually focus on a single type of testing [7].

An investigation of automated software quality assurance tools reveals that there is significant evidence that suggests critical fragments of a framework could be made to support programming language-generic automated quality assurance. It was previously stated that ‘Functionality, reliability and maintainability, can be achieved if a deep level of testing and analysis can be performed. For this to be successful the analysis tools must support a range of levels, objectives and methods.’ [7]. This is correct, although manual testing, in terms of black box and white box, which are defined to have levels, objectives, and methods, are not the focus of the framework as these are manual techniques. It would be more accurate to say that a deep level of analysis could be performed if a wide variety of base automated techniques, which in turn further facilitate software quality assurance techniques, could be implemented.

It is widely known that automated quality assurance can be divided directly into static and dynamic analysis [80]. Some tools specialise in one of these, although they can be combined, which in turn would allow a more thorough form of quality assurance [61] [62]. Dynamic analysis is analysis of properties of the software at runtime, which can be considered the more accurate form of analysis [59]. Static analysis, on the other hand, is abstract, analysing source codes before compilation [50]. This analysis identifies the potential of issues utilising functional, logical, interface, and bottom-up tests with an extensive list of possible outcomes with examples like: memory corruption errors, buffer overruns, out-of-bound array accesses, or null pointer dereferences [59]. Combining both forms of analysis would cover more automated quality assurance techniques and therefore would be considered a more comprehensive tool [61] [62].

This chapter will outline the current state of automated quality assurance. It will initially examine the breakdown of analysis and use secondary research to draw out common techniques used. Following this will be the high-level analysis of automated quality assurance tools to verify which should be taken forward to deep analysis extracting techniques for inclusion of those techniques within the taxonomy.

4.1 OVERVIEW OF QUALITY ASSURANCE

This overview is an outline of the current state of affairs, based on secondary research, decomposing quality assurance into sub-categories and defining the categories with common examples. The initial breakdown in Figure 4.1.1 shows the two highest-level categories, Static and Dynamic Analysis. Figure 4.1.2 is the key for the diagrams; any colours not shown in the key represent the level of depth of the sub-categories and techniques, i.e. from highest level to lowest blue, red, yellow, green, etc. A table has been included in the appendices that breaks down the techniques found in literature, and the loose reasoning behind each category has been included below. These categories are not strictly defined because further techniques and research may change the landscape of the taxonomy and it needs to be adaptable.



Figure 4.1.1– Hierarchy key

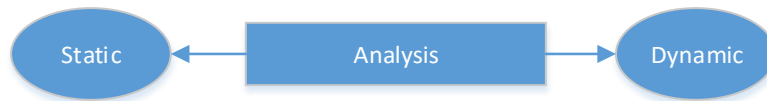


Figure 4.1.2– Basic analysis hierarchy

4.1.1 Detailed Static

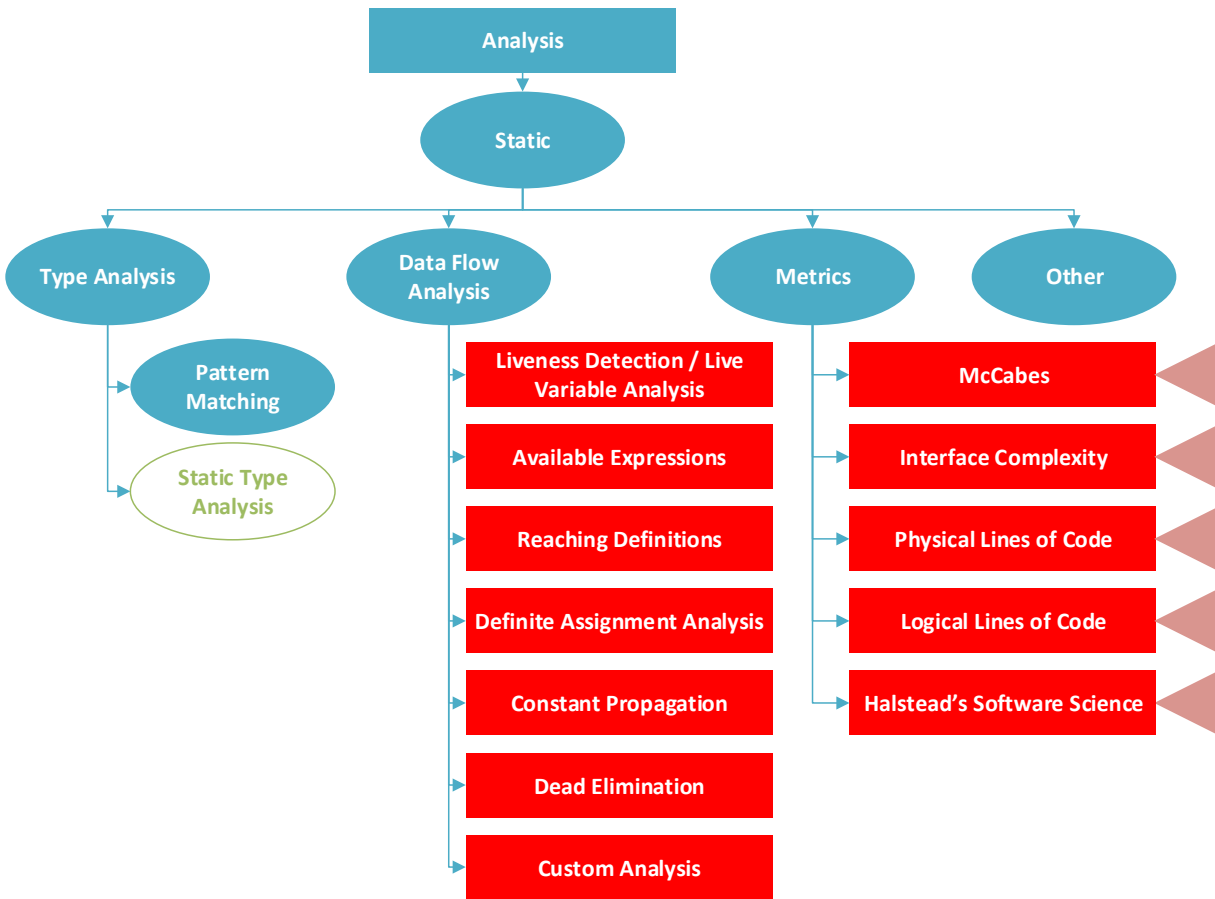


Figure 4.1.3– Static analysis hierarchy

As mentioned previously, the descriptions and definitions for each individual technique can be found in the appendices. The first draft of the static analysis categories (Figure 4.1.3) was created based on the implementation requirements of the techniques sourced from literature. Type analysis and those categories contained within require identifiable constructs and datatypes, therefore providing almost direct access to the source code. On the other hand, dataflow analysis requires a control flow graph to be generated, and the techniques are applied to this. Metrics are counting mechanisms that require generically identifiable nodes on a basic level. Finally, another section has been included to gather techniques that cannot be identified with any of the previous categories and will be dealt with at a later point.

4.1.2 Detailed Dynamic

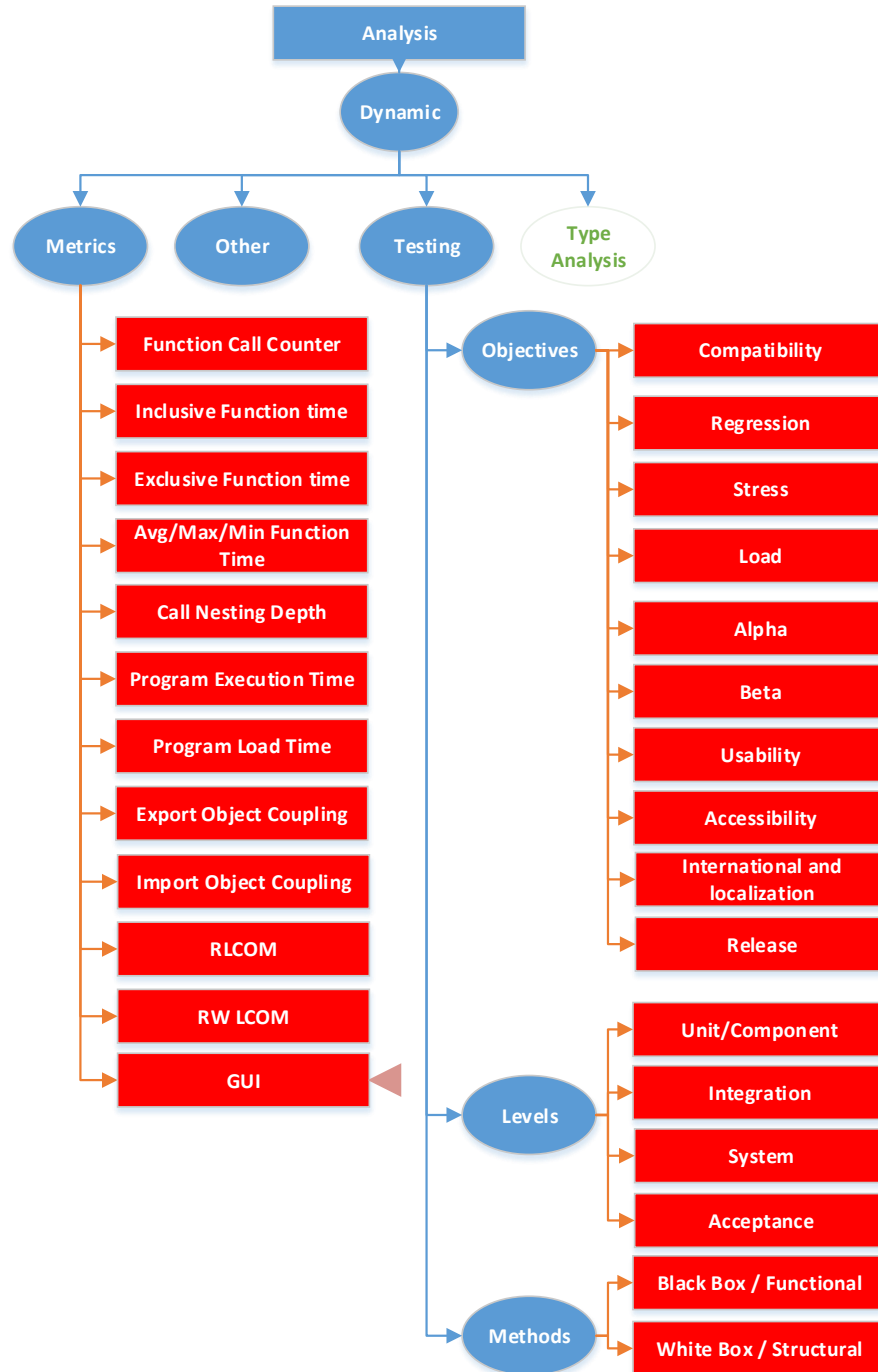


Figure 4.1.4 – Dynamic analysis hierarchy

Within the dynamic analyses categories lies Metrics and Type Analysis; these relate directly to the static analysis categories as they require the same level of access to be

implemented. The same reasoning has also led to the inclusion of the ‘other’ category. Testing performed with methods, levels, and objectives has been included; however, the techniques contained within are not directly automated. Nevertheless, there have been many attempts to automate various combinations of them, and this is an extremely important stage of SQA automation.

4.2 HIGH-LEVEL TOOL ANALYSIS

Before an in-depth analysis of relevant tools can be completed, an initial analysis of tool documentation must be carried out to select which tools will be extended to in-depth analysis; this is due to the wide variety of tools that implement some kind of quality assurance, either directly like WinFTP [101] or indirectly like an IDE such as NetBeans [93] used to assist with writing code.

The high-level analysis will consist of an overview using documentation or advertisement provided for the tool from various sources.

4.2.1 Independent Tool High-Level Analysis

This initial investigation includes independent tools that are used by various developers, quality assurance engineers, and companies. These tools were identified either in literature or through online searching, and were selected based on their feature list alone.

4.2.1.1 *winFPT*

FPT is a tool designed to analyse FORTRAN code utilising dynamic and static analysis. winFPT is the same tool with a graphical front end that is designed to be used with a Windows operating system [101]. The list of techniques implemented within FPT is extensive and includes measurement and assessment code metrics, report generation, error checking, formatting, pretty-printing, structural engineering, run-time testing, optimisation, software migration, and security

[25]. This research will utilise WinFPT as a tool for further study as its role in the QACC (discussed previously) project is key and aligns very well with the ideals behind this research. The analysis of the weather forecasting model, WRF, is a case study as well-aligned with this research as could be found. The application of a quality assurance tool upon a software model designed by non-programming experts is an example of why these tools are necessary. Following this, WRF utilises C code as well as FORTRAN, meaning that the results generated by winFPT do not take into account any bugs or issues within the C-coded sections.

4.2.1.2 *Parasoft*

Parasoft has developed several tools: Jtest, dotTest, and C/C++test. These tools provide automated quality assurance techniques to the programming languages Java, the .Net platform, and C and C++, respectively [58]. These tools apply the same automated quality assurance techniques to their respective languages; the tests these tools can perform are as follows: security static analysis, data flow analysis, software metrics, unit testing, component testing, code coverage analysis, and regression testing [102]. Unlike other commercial tools, Parasoft is discussed widely in several academic papers for its static analysis techniques and has even been rated as one of the best tools for static analysis upon the programming language C. The utilisation of the same automated quality assurance techniques over multiple languages suggests an inherent level of broadness within the techniques that have been implemented within the Parasoft tool range. This aligns well with this research and therefore will be taken into consideration for further analysis.

4.2.1.3 *Malpas*

Malpas provides automated quality assurance services for several programming languages; these are Ada, C, and Pascal. To do this, it utilises an intermediate programming language [103]. Several types of technique are implemented within Malpas that include control flow, data use, information flow, and compliance. Static analysis is utilised to administer these techniques [56]. Malpas has been used to quality assure safety critical systems as well as large scientifically based systems [23], which are the focus of this research. As Malpas can provide

automated quality assurance for multiple programming languages via the use of an intermediate language, this would suggest that this intermediate representation is applicable to a variety of programming languages and therefore generic, aligning this tool with the research.

4.2.1.4 Polyspace

Polyspace, designed to quality assure Ada, C, and C++, utilises static analysis techniques upon code designed for embedded systems, detecting issues based on coding standards such as MISRA C. These standards require several forms of analysis including arithmetic overflow, buffer overrun, and division by zero [24]. These techniques, though in this tool used for embedded code, are applicable to a wider variety of domains. Polyspace uses the same techniques upon multiple programming languages; therefore, this tool falls well within the scope of this research for further analysis.

4.2.1.5 Cantata++

Utilising test cases, Cantata++ can perform unit testing, integration testing, and code coverage analysis on C and C++ programming languages [22]. Due to the types of testing (unit and integration) performed, it is unclear if Cantata++ provides enough automated forms of program analysis to be useful within this research, as it seems essentially to be an automated unit tester plugged into an integrated development environment.

4.2.1.6 JNuke

JNuke's use of 'generic analysis', which essentially combines static and dynamic analysis [49], allows a more thorough form of overall analysis including techniques such as run-time verification, explicit-state model checking, and counter-example exploration on Java code [21]. As JNuke is targeted at a single programming language, it would be interesting to see if its techniques can be extrapolated to other programming languages or programming paradigms. The

combination of static and dynamic analysis is an interesting area that aligns with this research. For these reasons, JNuke will undergo further analysis.

4.2.1.7 *TestingAnywhere*

The following is a list of compatible software that TestingAnywhere supports: VB.NET, Win32, ActiveX, Delphi, 32 bit apps, PHP, .NET, C#, VB6, JavaScript, Java, 64 bit apps, Python, Silverlight, C++, AJAX, HTML, Perl, Oracle Forms, WPF, Macromedia Flash 1.0-8.0, and Adobe Flash 9.0 and later [104]. This is essentially any development environment that runs on Windows XP or later. Automated software testing, automated web testing, distributed load testing, regression testing, functional testing, black box testing, acceptance testing, keyword-driven testing, unit testing, data-driven testing, smoke testing, integration testing, compatibility testing, performance testing, system testing, GUI testing, automated Flex testing, Java application testing, Silverlight application testing, WPF testing, mainframe application testing, and third-party .NET supported testing are all supported testing types by TestingAnywhere utilising dynamic analysis to implement these [105]. TestingAnywhere's programming language independence is due solely to the SMART tool, which is a system recorder capturing properties of Windows in focus as well as permitting macro-like functionality recording changes to modified inputs. This tool is implemented in a similar way to macros in Microsoft Office in that the tests are recorded via recording of Windows events that allows testing anywhere to repeat tests and edit inputs, which is essentially an automated unit tester. Because this software, though extensive, relies on Windows events, thus tying it to the Windows operating system, this tool is not appropriate for this research, as most scientific software is developed in LINUX-based environments and Windows has stopped supporting languages such as Fortran. Another major disadvantage of this software is that it does not analyse code in any way and its techniques are all based on outputs from other development environments.

4.2.1.8 *Critical Comparison*

FPT [25] and Malpas [23] are both key tools within this research as they both match the scope very approximately. These tools are used within the scientific industry, and assess large

programs for quality. Malpas, however, only uses static analysis, which could imply that FPT covers a wider variety of issues, as FPT uses both static and dynamic analysis. On the other hand, Malpas is used for the analysis of safety-critical systems, which could indicate that Malpas is linked with specific industry standards, whereas FPT is not identified to adhere to any standard. A final point on these two tools, though FPT does not analyse more than one language, both Malpas and FPT use a similar technique to create some language independence; FPT uses an internal representation to analyse Fortran code and Malpas uses its Intermediate Language to analyse Ada, C, and Pascal; this key area links in directly with part of this research, language independence.

Polyspace [24] is similar to Malpas in that it is a static analysis tool for scientific software. Like Malpas, Polyspace adheres to standards although it is specifically for embedded systems, which may reveal some techniques for specific programming language paradigms. Like Polyspace, Parasoft [58] can be used for embedded systems, testing for adherence to standards, although Parasoft is not limited to that specific domain, showing an overlap of some techniques from embedded software to other areas. Unlike Malpas, which uses an intermediate language to create some language independency, Polyspace embeds into specific IDEs. However, this does not create programming language independence and therefore amounts to different programs for each programming language. This indicates that the techniques can be ported and could therefore be programming language-independent.

JNuke [21] and FPT are the only two tools that use both static and dynamic analysis. FPT uses dynamic analysis to create a wider area in which the tool can check for quality. JNuke uses its ‘general’ analysis (a combination of static and dynamic analysis) to make its tests more robust and accurate. JNuke, like FPT, analyses one language but uses no language independency whereas FPT uses an internal representation. JNuke is tied directly with Java as it has a very novel approach, because some of the analysis that JNuke performs requires specific capabilities that the normal JVM does not have (e.g. backtracking). The developers of JNuke have created their own VM written entirely in C to grant them the capabilities to analyse Java code more thoroughly.

TestingAnywhere [104] and Cantata++ [22] are very similar when overiewing, as both focus on using GUI input to allow users to automate tests. TestingAnywhere has its unique SMART tool that records Windows events, much like macros. The user can then edit these in order to change the values of tests. Cantata++ provides similar GUI testing; however, white box testing is also advertised, which does not seem to be based on linking a testing system to an IDE, which is the approach taken by TestingAnywhere to provide some level of white box testing. However, this form of testing could also be described as not white box but just several series of small black box tests in series.

WinFPT and Polyspace are both tools that will be taken forward for deep analysis due to their relevant features and access to these commercial tools being provided. JNuke and TestingAnywhere will not be included. JNuke will not be covered because of its similarities to already chosen tools. TestingAnywhere, though an interesting approach, has been deemed inappropriate due to its inability to automate analysis. Malpas has useful ideas and is of historical importance, whereas Parasoft and Cantata++ are both commercial products and difficult to acquire for this work; therefore, these tools will be discussed but not included as part of the in-depth analysis.

4.2.2 IDE High-Level Analysis

The IDEs use analysis indirectly initially to assist with writing code, but they also use further analysis for debugging and quality assurance. If those features are not built-in, then plugins are used, which will also be looked at within the high-level analysis.

4.2.2.1 *NetBeans*

NetBeans is an IDE developed for Java, providing support for several areas of Java, i.e. JDK 7, Java EE 7, and JavaFX 2 [106]. NetBeans provides support for many programming languages; however, it states superior support for C/C++ and PHP. Some other programming languages supported by NetBeans are XML, Groovy, Ada, and Fortran. [106] The list of languages is quite extensive, but the analysis of these languages is via toolsets specific to each language. Without any additional plug-ins, NetBeans has two forms of quality assurance for supported languages, static analysis in the form of NetBeans Hints (NetBeans Java Hints, for Java) which has 217 forms of hint/inspection [107], and the debugging system built into NetBeans serves as a form of dynamic analysis allowing variable watching and most forms of testing.

Plugins provide further forms of analysis, and examples are FindBugs performing additional static analysis [108], JUnit assisting in repetitive unit testing [109], Profiler project allowing for the use of profiling tools to monitor CPU usage amongst other aspects of program runtime [110], and many more. Due to the large library of languages and additional library of third-party plugins, NetBeans is a prime candidate for further analysis.

4.2.2.2 *Eclipse*

Although developed initially for Java, Eclipse-based language IDEs have been developed for popular languages including AspectJ with the AJDT (AspectJ Development Tools Project), C/C++ with CDT (C/C++ Development Tools) aimed at developing for Linux, and COBOL with the COBOL IDE [111]. As well as the fully integrated languages, third parties have used the Eclipse platform to develop plug-ins for custom versions of Eclipse such as the ADT (Android Development Tools) [112] and PDT (PHP Development Tools) [113].

Like most IDEs, Eclipse has its debugging tool as its main form of dynamic analysis, which includes variable watching and break pointing. Eclipse uses a hint system for code

development, which it is its main form of static analysis, or Quick Fixes [114]. As well as the code assisting annotations, Eclipse also has compiler errors and warnings to assist with the improvement of code quality, which constitutes another form of static analysis.

There are also many plugins for Eclipse to allow for other forms of analysis and improvement on quality of code, a few examples of which are TPTP (Test and Performance Tools Platform) [115]; EcEmma, for code coverage tests [116]; FindBugs, further static code analysis (byte code analysis) [108]; EclipseMetrics, for common metrics [117]; JDepend4Eclipse, for circular dependencies [118]; PMD for static code analysis [119], etc. Because of all of its possible additions from both a language perspective and plugin perspective, Eclipse is another prime subject for further analysis. Eclipse and NetBeans are very similar; accordingly, only one of these needs to be further analysed to extract the quality assurance techniques used. The one to be taken forward will be NetBeans due to Eclipse being less accessible, as the plug-ins are developed by companies and are not open source.

4.2.2.3 *Visual Studio*

Although Visual Studio, developed by Microsoft [120], has a selection of fully supported languages, it differs in a significant way from other IDEs. Visual Studio's main focus is the support and development of the .NET platform, into which many languages can be integrated [121]. For example, the fully supported languages that Microsoft itself has developed are Visual C#, Visual Basic, Visual C++, etc. [122]. Along with these fully supported languages, Microsoft partners and other companies have brought their languages to the .NET platform that can then use Visual Studio as their IDE, examples including COBOL, APL, Pascal, etc. [123]. Due to the languages being developed into the .NET platform, before being executed, the source code is compiled into the Common Intermediate Language (CIL), which means that there are links with language independence within the .NET platform [124]. Other than the .NET platform, the other languages supported within the IDE are scripting languages such as Windows Script Host (WSH), VBScript, and Jscript .NET. There are other languages, as well, such as Visual J++, Transact-SQL, and Extensible Markup Language (XML) [124].

Like most IDEs, Visual Studio offers a variety of built-in and third-party forms of quality assurance for code. There are a variety of forms and techniques used by Visual Studio that are separated appropriately and described well within the Microsoft Developer Network [125]; these include forms of static analysis such as the Code Analysis Window used for displaying static analysis issues within the project [126], Code Clone Detection [127], managed code analysis that uses rule sets to target specific coding issues [128], and the Visual Studio Application Life Cycle Management that allows for the production of static metrics based on the solution/project [129]. There are also built-in tools for dynamic analysis such as Intelitrace, which records code execution [130]; Test Explorer designed to assist with the creation of unit tests [131]; and Profiling Tools [132].

As well as the comprehensive built-in tools, there are in addition third-party plugins to assist developers with quality assurance; examples are CodeRush, which statically analyses the code for some additional forms of best practice [133]; Parasoft's dotTEST, which provides additional .Net static analysis, unit testing, and code review [134]; and finally NDepend, used for analysing and visualising dependencies for complex .Net applications [135].

Visual Studio contains quality assurance techniques. This, with additional plugins providing more quality assurance techniques and the fact that Visual Studio is used to develop on the .NET platform that contains multiple programming languages, of which some are not based in the same paradigms, makes Visual Studio a considerably generic quality assurance tool. Though Visual Studio develops on the .NET platform, it is uncertain how generic the quality assurance techniques are, due to inconsistencies between available analyses for different programming languages, e.g. the profiler has differences between C# and F#. As programming languages run within Visual Studio are converted to the Common Intermediate Language (CIL), it will be interesting to see which of the techniques are performed by Visual Studio on the CIL or using the source code, that could highlight which techniques can be extrapolated onto multiple programming languages and further this framework.

4.2.2.4 Critical Comparison

As quality assurance becomes more integrated into the development life cycle rather than just a stage at the end [136], IDEs such as those briefly reviewed above have become even more important. IDEs have the advantage of including quality assurance techniques from a static and dynamic perspective due to the access to the source code and usually the ability to compile and run code quickly after small modifications. Additionally, in most cases, IDEs contain some form of debugger.

Due to the nature of the IDEs, most types of quality assurance are applied via built-in functionality or via third-party plugins that all of the IDEs above support. The main difference between the IDEs is the platform itself. The NetBeans platform is based on a modular system allowing plugin ability, incorporating all stages of development into one usable system [106], which is very similar to the Eclipse platform. Both are designed as a Rich Client Platform (RCP) using an editable but universal workbench to develop applications in many languages [137]. The two platforms are extremely similar in many ways, with only a few insignificant differences [106]. Visual Studio, however, has a very different platform as, rather than using pre-developed compilers, the .NET platform uses its own or third-party compilers that allow the source code of a supported language to be written into code that is adopted into its CLR [138]. This is an interesting difference, as quality assurance techniques can be applied to CLR via the Microsoft research development Phoenix project and, more significantly, the Shared Source Common Languages Infrastructure (SSCLI) [139]. This would allow a partially programming language-independent form of quality assurance although conversion from and to the CLR would have to be simple and significant, with reviews of modifications made that the developer could understand in the programming language in which they wrote the program. Alternatively, a more effective solution would be to tie all modification and identification of code for review back to the original programming language.

4.3 DEEP ANALYSIS

In this section, the in-depth analysis of the selected tools will be discussed, breaking down the techniques that the tools use and describing how they are implemented. Some of the tools discussed are commercial, and therefore it is difficult to assess their inner workings, although we can infer using other tools and secondary research. The method used for the deep analysis was based on the documentation provided; initially, a review of the features documented gave a list of techniques and areas of the tools that required discussion. Further, from this a general use approach, where use of the application and external documentation discussing the use of the tool, e.g. tutorials, allowed anything that may have been missing from the feature lists to be added to the areas that needed analysis. The breakdown occurred through access to source code, if permitted; otherwise, literature was used to infer the implementation of the technique. This literature was composed of documentation provided for the tool, scholarly literature, and finally developer blogs where the author was a developer of the tool or part of the company that provides the tool.

4.3.1 NetBeans

For the in-depth analysis of QA techniques found within NetBeans and its plugins, the focus was set on Java-specific analysis. The focus on Java was due to NetBeans being initially designed to support Java; therefore, it is likely that there will be more complete support for QA.

4.3.1.1 *NetBeans Java Hints*

Java Hints is a list of different ‘hints’ presented to the user when writing code using the NetBeans IDE. They are displayed by underlining specific code related to the hint and then, once hovered over, present some information about how the code could be modified or why it could pose an issue with the running of the program. Java Hints, whilst containing a large volume of built-in hints, as listed in the extended appendices, also allows users to write their own ‘Hints’ [140].

Java Hints works by pattern matching code against prewritten sample code with wild cards, essentially looking for similar code with only slight differences, e.g. there may be an issue with a declaration or assignment of a variable, and this can still be detected regardless of what the variable name may be. These hints can be used to inform the user of potential compiler issues all the way to best practices and specific project development requirements, e.g. company coding policy. As well as informing the user of issues, the hints system also allows for auto correction, e.g. if an instance of a class is referenced but the class does not exist in the context of the program, the skeleton of the class can be auto-generated.

4.3.1.2 *NetBeans Debugger*

NetBeans actively allows users to visually debug a program using a system of breakpoints. Using a breakpoint allows a user to stop a program during runtime in a ‘frozen’ state whilst details about the program can be viewed and in some cases modified [141]. To assist users, there are several windows available to display relevant information for debugging:

- Watches
- Variables
- Call Stack
- Loaded Classes
- Threads
- Sessions
- Analyse stack trace

Like most debuggers, NetBeans allows users to step through, into, and over lines of code after a breakpoint has been hit, thus enabling users to gain a better understanding of why a program could be causing issues. Utilising the Variables window, the user can see exactly what variable instances are available in the current scope, and certain variables can be selected and displayed in the watches windows to cause less confusion in larger programs. As well as viewing variables, they can also be modified in the watch window to cause and fix issues during testing.

The Stack window allows users to view function calls that currently lead to the compiler error or breakpointed code, or Call Stack, a convenient way for a user to identify what code is currently being executed and how the program came to this point.

The Loaded Classes debugger window allows users to view how many instances of classes there are running in the current program at the current point of computation, and the instance percentage can also be viewed showing users the percentage of instances that that class currently takes up. This window could be used for memory performance increase by detecting unused class instances so that these can be minimised.

The NetBeans debugger allows the user, utilising the ‘threads’ windows, to pause at a breakpoint and essentially break down different threads whilst other threads are running. This would allow the user to test for individual issues whilst ignoring other threads, or to test debug multiple threads at the same time, as well as testing for race conditions and shared resources [142].

The debugging window ‘analyse stack trace’ is an interesting feature, mainly included to save time debugging. This feature allows any stack traces outputted, usually when there is a compiler error during running or debugging, to be copied from the output panel into the analyser window. The feature then breaks down the stack trace, turning the different locations into links to the location to which the stack trace refers [143].

4.3.1.3 *NetBeans Profiler*

The NetBeans Profiler has several options available depending on the user’s objectives. The three choices given when using the profiler are Monitor, CPU, and memory, which provide a variety of different tables and graphs of running information. All of these run live, allowing users

to monitor larger programs as they run [110]. The monitor option allows users to view, live or after the fact, virtual machine telemetry, which shows heap space usage over system time. Threads windows display the thread state during the execution of the program, i.e. running, sleeping, waiting, parked. Finally is the Lock Contention, which is a view to show if two threads are accessing the same resource at the same time where one is locked until the other thread has finished.

Another option the Profiler gives users is CPU, which has only one window displaying individual methods, classes, or packages and their time running and total CPU time. Finally, the Memory option for the profiler is similar to the CPU monitor showing methods, classes, or packages broken down and their total impact on memory measured in bytes and instances of the object.

4.3.1.4 NetBeans JavaDoc Analysis

Though this may not seem directly involved in the analysis of a program, documentation plays a key role in the process of quality assurance and the maintainability of a program [144]. When using JavaDoc, NetBeans requires a certain level of comments describing the units of a program, including methods and functions with information about returns and parameters, etc. The analysis in NetBeans identifies all non-commented methods, functions, and classes and then can auto-insert blank comments in taking users directly to the area for input [145].

4.3.1.5 SQE (Software Quality Environment)

SQE is a plugin for NetBeans that packages several different quality assurance tools into a single easy-to-include plug-in [146]. SQE includes four different tools, FindBugs, Checkstyle, PMD, and Dependency Finder, all providing different facilities for quality assurance, and each is individually discussed below.

4.3.1.6 FindBugs

FindBugs analyses Java bytecode using BCEL (Apache Byte Code Engineering Library) and uses the bug patterns concept [147]; the techniques used for this are pattern matching as well as dataflow analysis [148].

Extensive bug detections are broken down into 9 categories [149]; similarly to most source code-level static analyses, FindBugs informs the user about issues through underlines. However, another form of result for the user is a report broken down by Project, Package, Class, and then individual error. This report then links to the line of source code that is causing the issue.

Because FindBugs runs on Java bytecode, it can produce different errors from those that run on source code or internal representations generated from source code, making it an excellent additional tool (justified by it being added into several IDEs via plug-ins [150]) to use when quality assuring software. However, using the pattern matching concept means that some of the methods used to find bugs could be adopted by LIQA. Determining the extent to which this is possible would require a detailed analysis of each individual method and pattern used. On the other hand, the dataflow analysis techniques should be easily adopted by LIQA due to dataflow analysis using CFG, to which the GASTM can be converted.

4.3.1.7 PMD Source Code Analyser

Like FindBugs, PMD is a static analyser that syntactically checks source code whilst not having a dataflow analysis component [148]. PMD looks for erroneous code as well as code that may be deemed incorrect under standards or preferences in developer writing, i.e. Using StringBuffer over += for string concatenation [119].

Users can add rules to PMD using Java. PMD breaks down source code into an AST form of internal representation, but because rules can also be written in XPath, it can be assumed that this AST is stored in an XML format. This method of quality assurance essentially works by pattern matching against previously established ‘rules’. Due to this format of internal representation, it can be assumed that most, if not all, of the quality assurance methods that can be performed by PMD could be transferred and performed by LIQA.

4.3.1.8 *Dependency Finder*

Though Dependency Finder itself claims to be a powerful tool that has many features [151], these features are limited to a single one by the NetBeans plugin. The Dependency Finder generates a graph of the selected project depicting dependencies between packages, with optional areas to include in the graph, which are JDK and external packages.

A screenshot of the Dependency Finder in use can be seen in Figure 4.3.1:

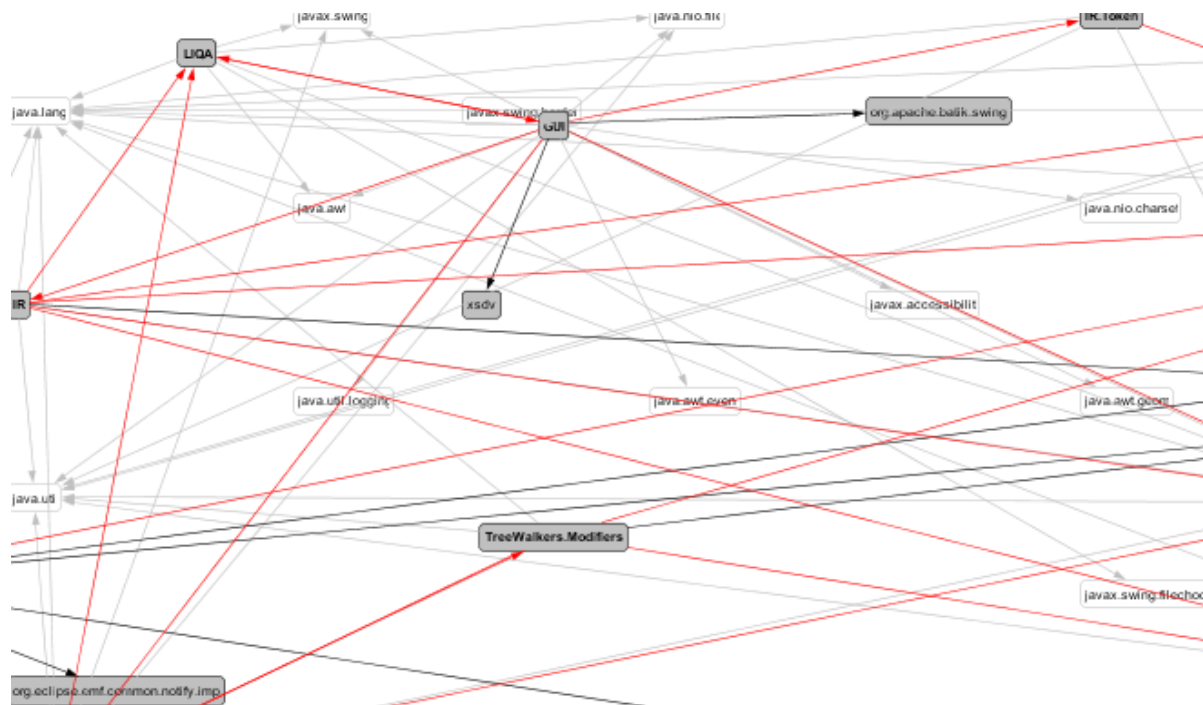


Figure 4.3.1 – NetBeans dependency finder

The red arrows depict internal package dependence, black lines depict external dependencies, and the faded gray lines depict JDK dependencies.

Depicting dependencies in this form allows developers to quickly assess where dependencies lie and therefore better manage their code base. Though this is primarily a concern for object-oriented languages, the technique used to extract the dependency information and generate graphs could be implemented upon the GASTM structure, though this would be a large project to undertake.

4.3.1.9 *Checkstyle*

Checkstyle features can be broken down into three types: duplicate code check, class design problems, and bug patterns. Checkstyle is a highly configurable form of checking source code for coding standards, allowing users to write their own checks as well as enabling and disabling ‘standard’ checks [152].

Checkstyle utilises ANTLR [32] to form an AST from the Java source code of a project and then parses that project using a ‘TreeWalker’ that allows the checks to be performed [153]. The checks are based on pattern matching, looking for a predetermined pattern of code that does not adhere to the coding standards selected. Users can add to the standards by first learning how to navigate the AST and then implementing a check by writing it in Java. This works in the same way as LIQA, although Checkstyle is Java only and does not use a sufficiently generic AST to be performed upon other languages. As Checkstyle already runs using an AST, it could be modified to run upon the GASTM.

4.3.1.10 *JUnit and xUnit Framework*

JUnit is based on the better-known xUnit framework that allows unit tests to be automated [154]. The framework is imported as a library and tests are written in classes that

extend 'TestCase', or using the newer annotations to set up, run, and break down the test. Various types of test can be run, such as true/false tests, equality tests, null test, same comparison, etc. These tests are written in the language of the application under testing in a separate class, and each test is divided into its own method [155].

JUnit, as stated above, is based on xUnit. There are many other unit test applications/plugins that have been developed from xUnit such as CppUnit, NUnit, SUnit, etc. [156]. This shows that the unit testing framework is sufficiently generic to be ported between programming languages; however, it has been ported and is not a single library that all programming languages access. This is obviously due to the difference in programming languages and incompatibility between them. Another reason for the dispersion of xUnit into programming languages is due to development, allowing programmers who are used to a particular programming language to write tests in that programming language. However, as xUnit is a framework and could be considered as a single source, its inclusion within this project's own framework is plausible. However, how applicable it may be for implementation in to a single tool, i.e. LIQA, is questionable, to say the least. Another issue that can be highlighted is that xUnit is already a highly used and widespread framework. Usually, unit tests are done throughout development, and therefore it may not be reasonable to include a unit tester in an automated QA tool.

4.3.1.11 Techniques

The following tables describe the quality assurance techniques extracted from NetBeans, broken down into the categories generated by the review of literature. The initial red box contains the name of the tool; following this, the techniques are divided into dynamic and static (see the relevant figure) analysis following which the categories, in blue circles, further dictate the requirements of implementation until the techniques, in red squares, can be categorised.

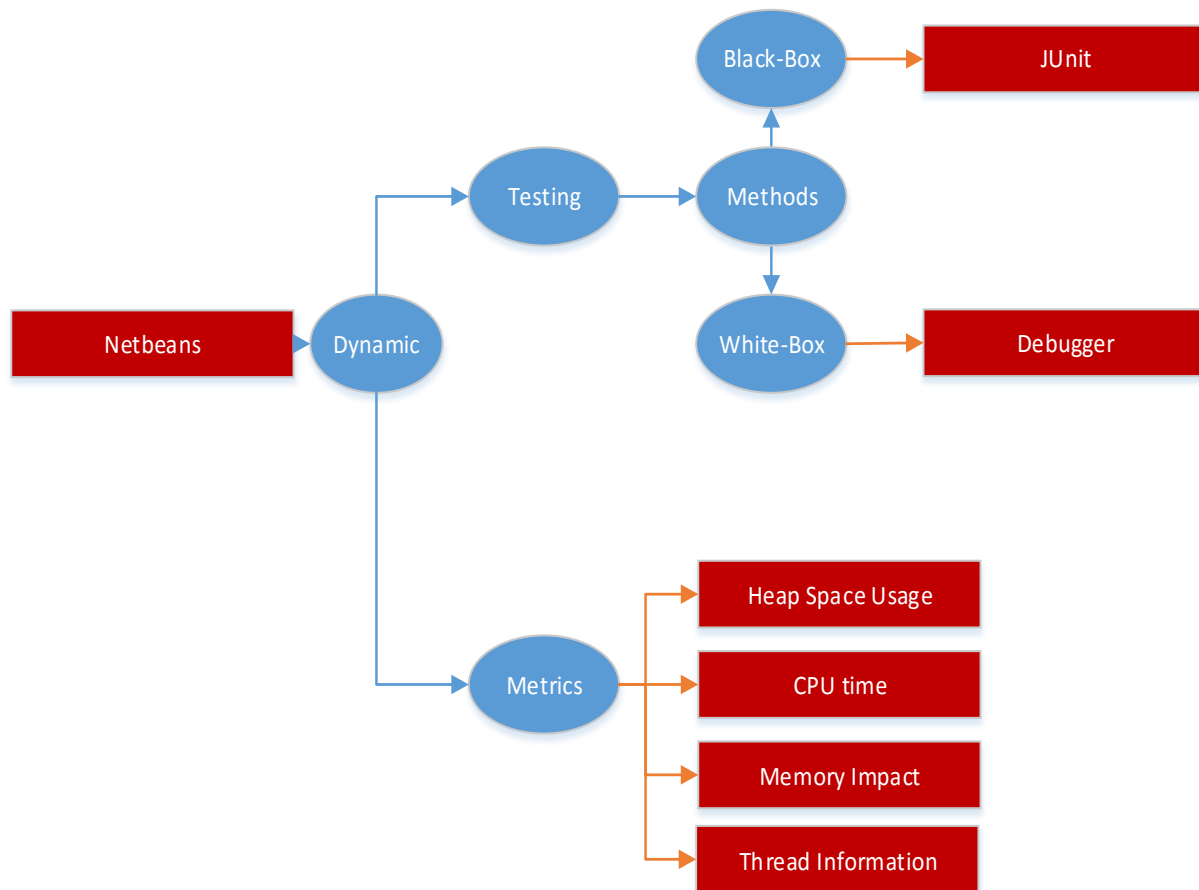


Figure 4.3.2 – NetBeans dynamic analysis techniques

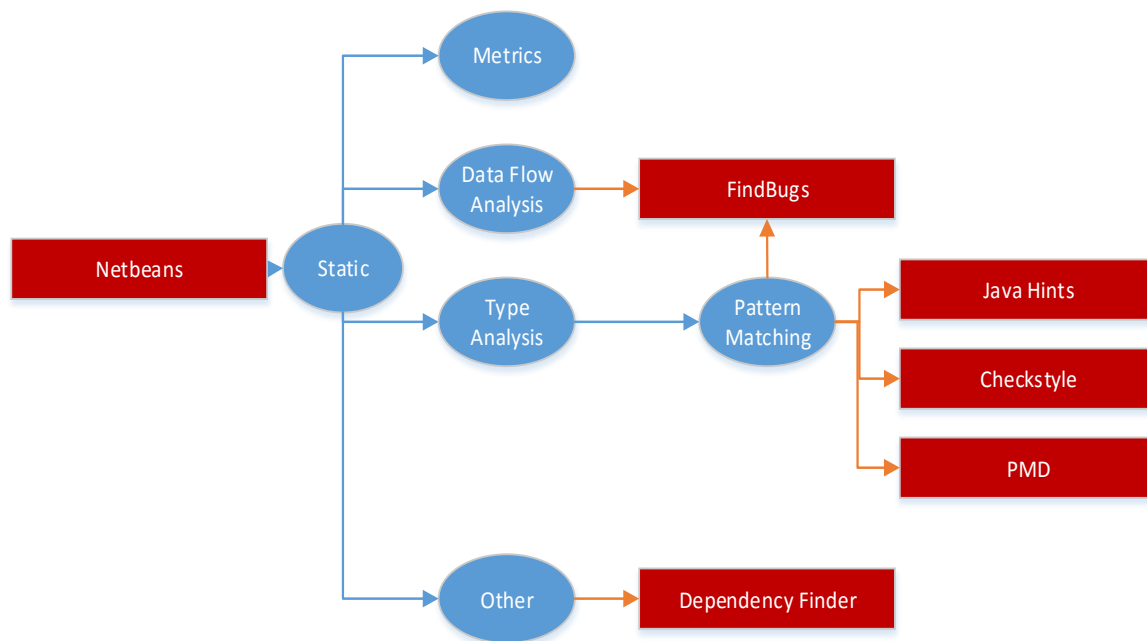


Figure 4.3.3 – NetBeans static analysis techniques

Findbugs is an exception to the rule of taxonomy, coming under more than one heading; this is due to Findbugs being a container for multiple techniques that individually come under pattern matching or dataflow analysis, but not both.

4.3.2 Visual Studio

The in-depth analysis of Visual Studio will be performed on Visual Studio 2012 Ultimate. This is due to some of the features, such as IntelliTrace, only being available on the most expensive version of this software. Furthermore, the programming language C# will be the initial language discussed because its similarities to Java would assist in a direct comparison of Netbeans. Furthermore, due to Visual Studio supporting a multitude of programming languages such as F#, C++, VB, and more, some comparisons will be drawn.

4.3.2.1 *Debugger*

The debugger has a variety of specific smaller tools to enable debugging of different types of applications, e.g. managed code, mixed code, DirectX graphics, GPU code, and many more [157]. The focus of this part of the analysis will be on four sections of the debugger: the variable watcher [158], breakpoint and tracepoint, the Assert Classes [159], and IntelliTrace [160]. These are the core components that are most commonly used and are most likely to be applicable to multiple programming languages.

To demonstrate the core components of the debugger, Visual studio 2012 Ultimate [161] will be used and a sample console application will be run, as shown in Figure 4.3.4.

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.VisualStudio.TestTools.UnitTesting;
namespace TestApp
{
    class Program
    {
        static void Main(string[] args)
        {
            //declaring variables
            string vTest = "";
            char[] testArr = new char[] { 'T', 'e', 's', 't' };
            //set chars into string
            foreach(char c in testArr)
            {
                vTest += c;
            }
            //writeout string
            Console.WriteLine(vTest);
            //Assert to cause break
            Assert.AreEqual(vTest, "Test?");
            //stop program from ending display
            Console.ReadKey();
        }
    }
}

```

Figure 4.3.4 – C# debugger test program

Breakpoints are the most basic of all debugging features. They allow the user to set points in the code where the application will stop running, which permits the user to view variables and run the program line by line, jumping over certain lines, stepping into method calls, and continuing to run the program [162].

A tracepoint is a slightly different feature in that it works like a breakpoint but, rather than stopping the application from running, it executes an action and/or stops the application, as well [162]. The simplest use of the tracepoint allows users to print a message when hit, replacing the need to add console print lines and then remove them when the application is finished [163].

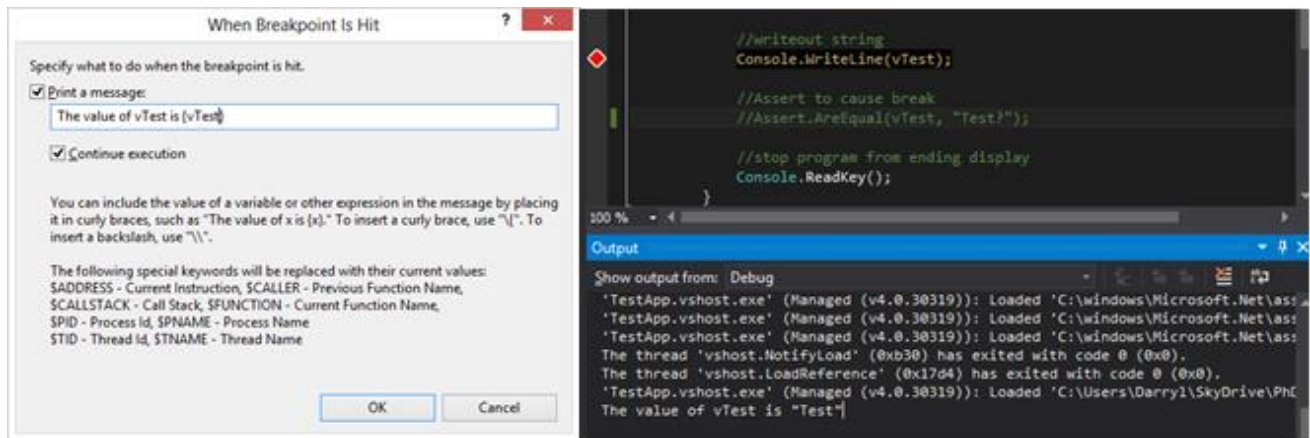


Figure 4.3.5 – C# debugger tracepoint message

Figure 4.3.5 shows the Visual Studio 2012 trace point menu, which has been modified since Visual Studio 2010, by removing the ability to run a macro based on the activation of the tracepoint. This may seem like the removal of a useful feature, but it makes more sense as, with the inclusion of ‘smart’ messaging, this allows the user to print runtime information like stack and variables without the need to retrieve this information in a macro [164].

The variable watching technique implements a separate window within the IDE once a breakpoint has been activated. This then allows users to set a variable of interest, which will then display the values they contain [158]. This is especially useful if a user is managing large or complex data structures. Another feature of the watch windows is the inclusion of expressions that are separate from the program running, as shown in Figure 4.3.6 with the expression ‘VTest + “ing”’.

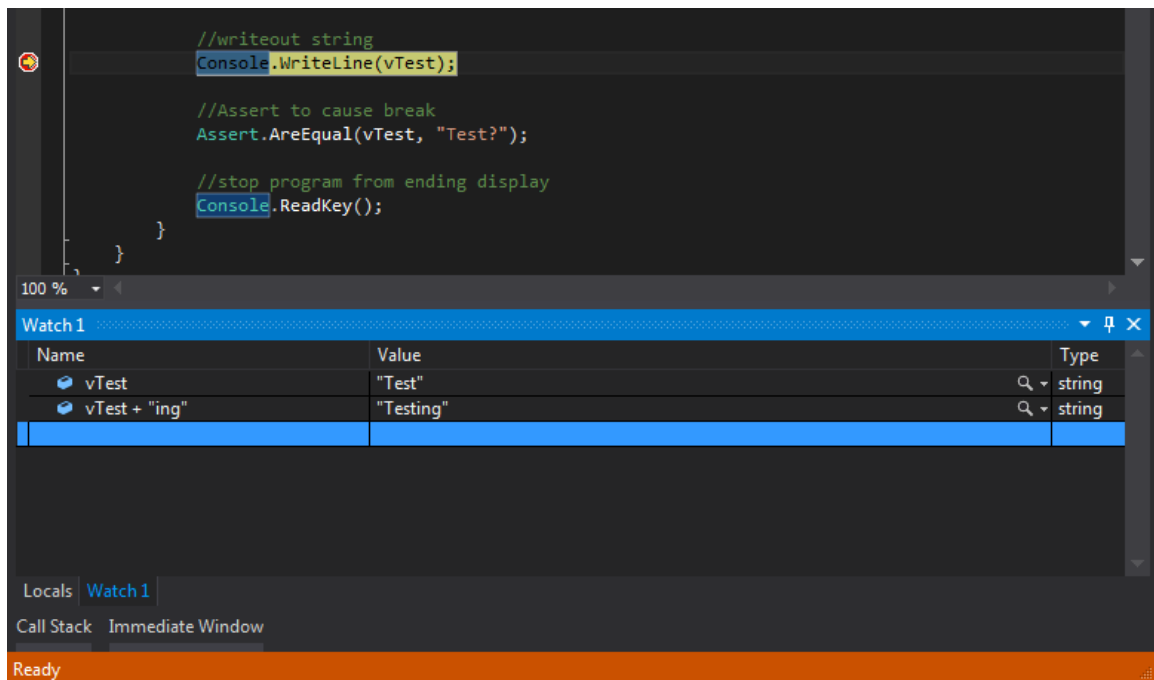


Figure 4.3.6 – C# debugger variable watcher

Assert Classes are another form of breakpoint, but are lines of code that perform a check of some kind and only cause a break if the check returns false. A few examples are simple checks like comparisons or null checks. Figure 4.3.7 shows a break in the example code because of an incorrect comparison [159].

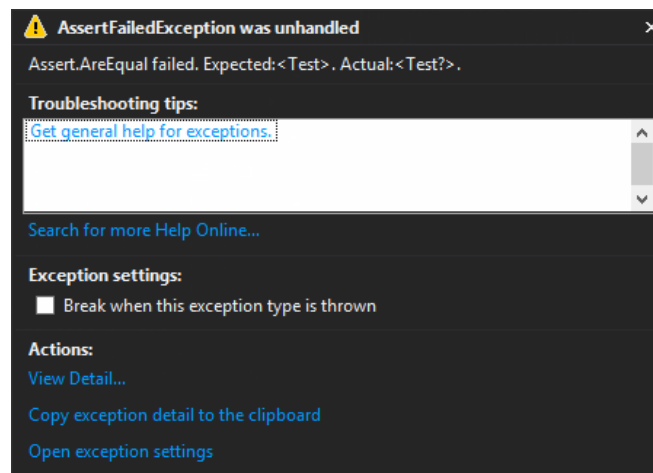


Figure 4.3.7 – C# debugger assert classes

IntelliTrace is a feature only available on Visual Studio Ultimate and essentially allows users to backtrack through code based on break points. The IntelliTrace feature saves the program state at each breakpoint and allows the users to jump from point to point regardless of where the program is stopped [160]. Figure 4.3.8 shows the IntelliTrace window that has recorded the two states of the program at each breakpoint; links can also be seen to bits of relevant information for testers, i.e. the call stack.

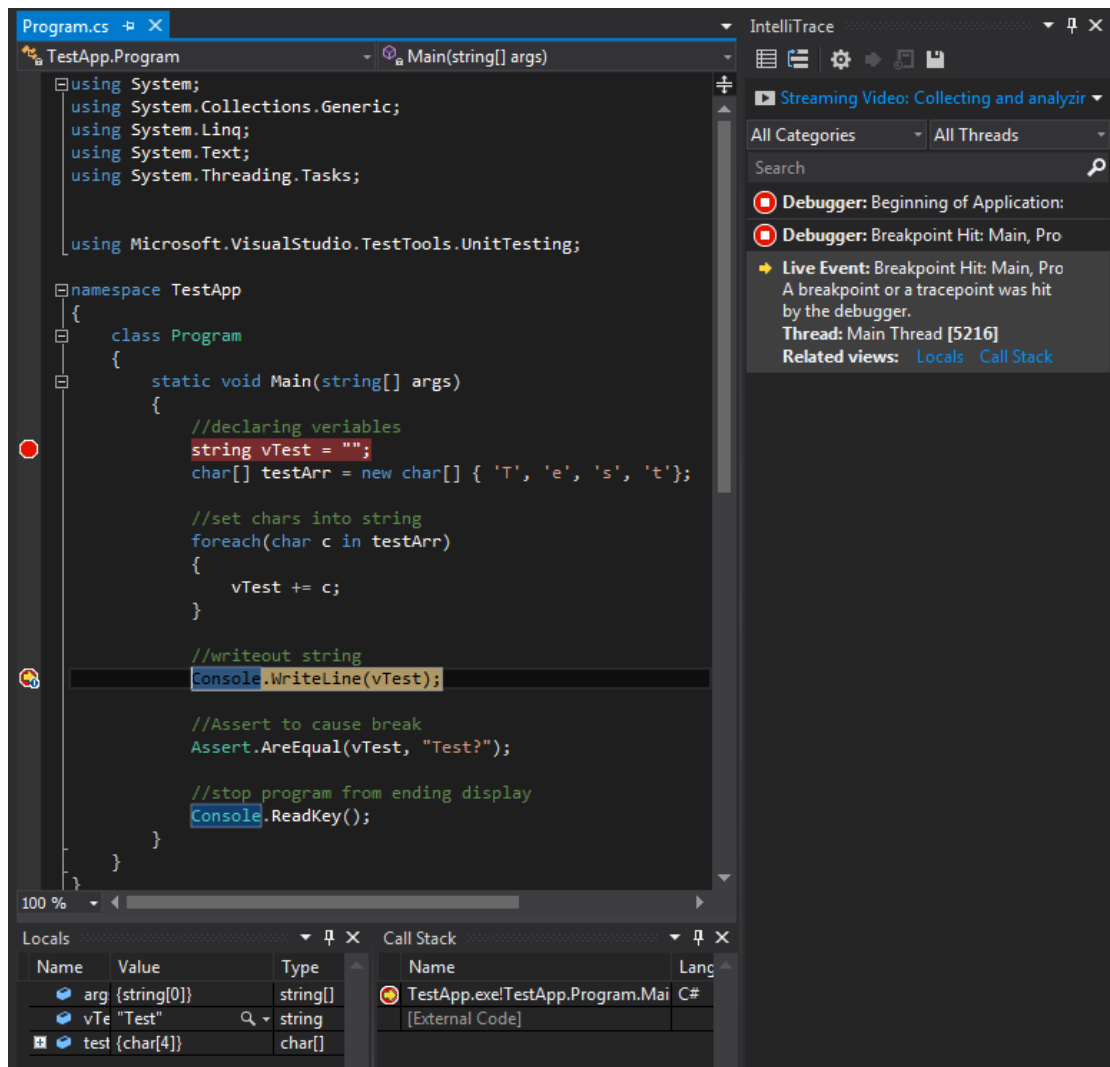


Figure 4.3.8 – C# debugger intelitrace

The debugging features in Visual Studio are the most used features for testing in the package. The debugger works based on the Microsoft Windows Application Programming Interface API [165], and it is important that debugging techniques are highlighted, as they are a very common and powerful set of tools for quality assurance. As this project is aimed at automated quality assurance, of which debugging, as a manual approach to testing, is not part, it will not be further described.

4.3.2.2 Error Correction

Error correction in Visual Studio is performed whilst writing the code and informs the user via underlines of the code. There are 3 types of underline: red, which denotes syntax errors; blue, which are compiler errors; and green, which are compiler warnings. It should be noted, however, that all syntax errors become compiler errors if the program is built [166]. This form of quality assurance is part of Visual Studios IntelliSense [167]. The way in which Visual Studio detects these errors is by constantly compiling the code in the background whilst it is being written. This explains why the majority of errors are compiler errors: the red/blue underline errors are errors that will prevent the program from compiling; this means that if a compiler error is present, no green underlines in that class will be displayed. This is due to the way in which compiler warnings are generated; it can be inferred that warnings can only be ‘discovered’ if the program fully compiles and due to the types of warnings, i.e. unused or unassigned variables [168]. It is expected that these are general compiler optimisations produced using dataflow analysis techniques. A full breakdown of the compiler errors and warnings is given in the extended appendices.

4.3.2.3 Analyser

Visual Studio provides several tools under its analysis header, or profiler, which essentially covers areas that plug-ins did in older versions of Visual Studio. These tools include a performance analyser, code analyser, code metrics, and code clone detection.

There are 4 types of performance analyser: CPU sampling, which monitors CPU-bound applications with low overhead; instrumentation, which measures function call counts and timing; .Net memory allocation, which tracks managed memory allocation (sampling); and resource contention data, which detects threads waiting for other threads. Although the CPU-bound applications analyser is recommended, the instrumentation analyser will probably be the easiest to use because it shows data close to source code, i.e. number of calls to functions and elapsed time with those functions, whereas the CPU-sampling analyser refers to the .dll calls, which can cause confusion.

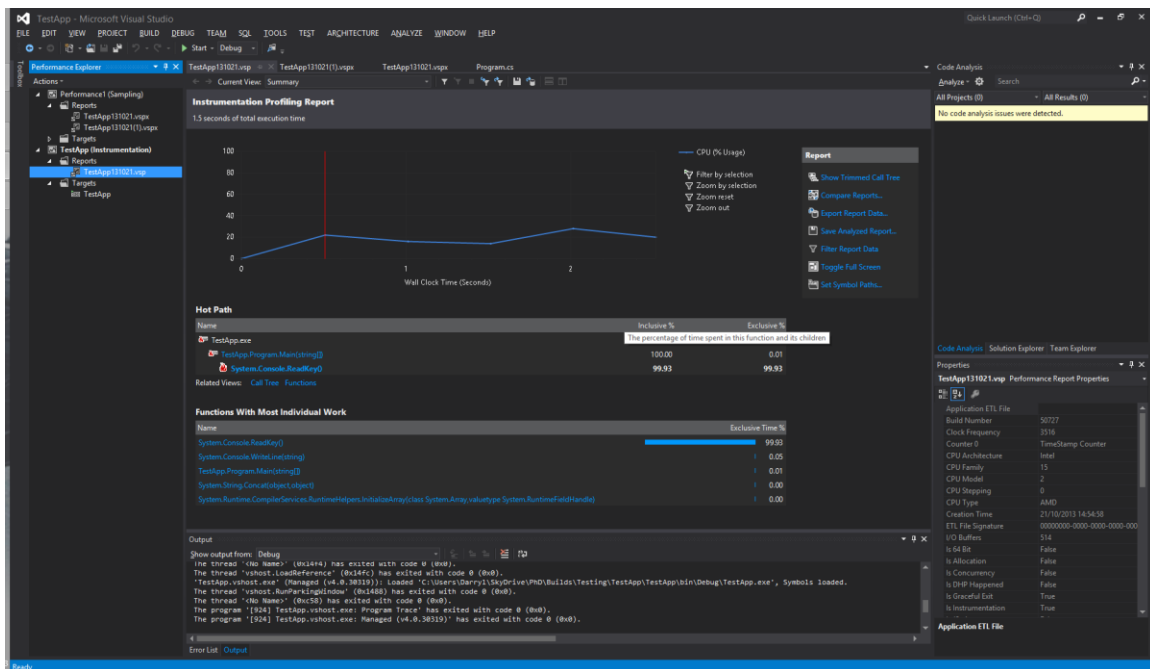


Figure 4.3.9 – Visual Studio 2012 performance analyser

Figure 4.3.9 shows the performance report (instrumentation) for the test application. As can be seen, there is a variety of information; the graph shows CPU usage against wall clock time, and the elapsed time is at the top. The 'hot path' is displayed, which is a highlight of the most 'expensive' code path; however, a breakdown of all the function calls with elapsed times and further function calls can be found through a link. Finally, a top five of the functions, with the most elapsed time, is at the bottom, which could be extremely useful for identifying a major load within the application.

The Visual Studio code analyser statically analyses code, identifying issues. As with IntelliSense, the analyser cannot work if the code has a compiler error, although subsequently it can be used as a more comprehensive form of static analysis [169]. A list of all the checks can be found in the extended appendices. Not only does the analyser allow users to scan for previously established code standards that could improve security, performance, or general practice, but it also allows users to specify custom rules, drawing together rules that third parties have written for different sets of standards. Moreover, users can also write their own rules that could be naming conventions, standard load database connections strings, etc. However, this is not a simple process and can be rather extensive depending on what and how many rules a user wishes to include, especially since it is using a third-party API [170]. The exact manner in which rules are applied to source code is unclear due to a lack of documentation, although, since all rule sets can be applied to all programming languages except F# [171], it can be assumed that either the rule checks are performed upon the Intermediate language or upon a token stream. The token stream is suggested because the code analysis may not be run on F#, as it is a functional programming language.

Visual Studio also allows users to run Code Metrics as a form of analysis. There are five types of metric performed: Maintainability Index, Cyclomatic Complexity, Depth of Inheritance, Class Coupling, and Lines of Code [172]. These are Microsoft ‘versions’ of the standard metrics, but to calculate these metrics, several items need to be identified and counted, e.g. the Maintainability index requires other counting mechanisms. The calculation for the metric is:

$$\text{Maintainability Index} = \text{MAX}(0, (171 - 5.2 * \log(\text{Halstead Volume}) - 0.23 * (\text{Cyclomatic Complexity}) - 16.2 * \log(\text{Lines of Code})) * 100 / 171)$$

Figure 4.3.10 – Maintainability index calculation [173]

In this equation, Halstead’s volume requires the number of distinct operators, the number of distinct operands, the total number of operators, and the total number of operands. These base

counts can be used to perform more varying metrics, which raises the question of why Visual Studio does not possess a feature, similar to the rule set feature, that allows users to create and add their own metrics.

Unfortunately, there is no documentation outlining how the metrics are performed. However, it could be possible to generate metrics for applications in two ways within Visual Studio. The first, and more improbable, is that the source code is analysed directly and keywords, paths, data structures, etc. are counted before calculations are made. The second and more likely way in which Visual Studio performs metrics is upon the Intermediate Language, similarly to NDepend [135], which is a tool for analysing .NET projects. The second way of implementing metrics would explain why there is a limited number of metrics available and also why users are not allowed to add to the list, as not all metrics can be run for every programming language, e.g. NDepend performs many metrics, but some are only available on C# programs [174].

Code clone detection is a more recent implementation into Visual Studio 2013. Essentially, it inspects the source code for matching fragments with varying degrees of similarity, with Type 1 being a code fragment identical to Type 4 in which two or more code fragments perform the same computation but vary only in syntax and simple content, e.g. literals, identifiers, etc. [175].

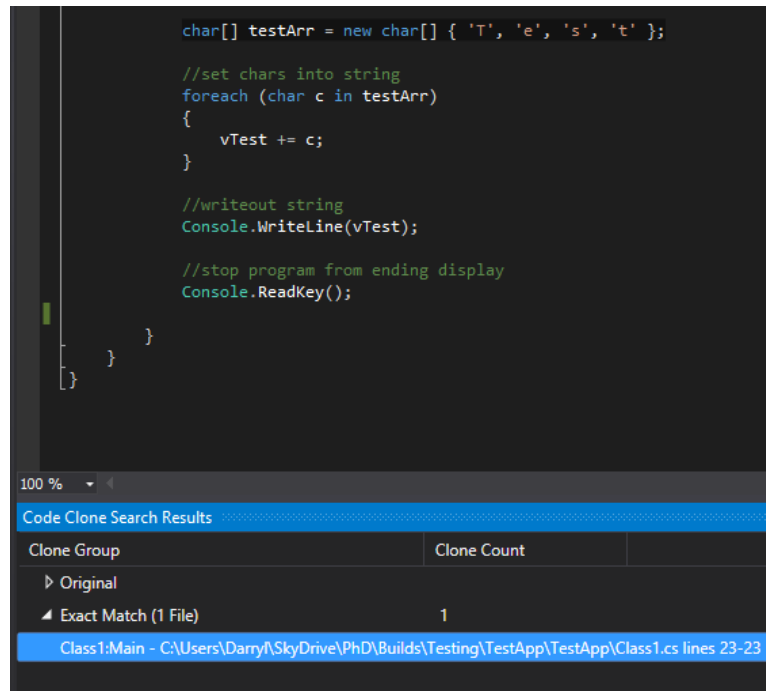


Figure 4.3.11 – Visual Studio 2012 code clone detection

As shown in Figure 4.3.11, a match has been identified as present. This system works by pre-processing source code and will be fragmented. Following this, if the code is not identical, an intermediate representation, which can be assumed as being abstract from syntax, would be used to perform further analysis. Further algorithms can then be used to ascertain the type of match and to filter unlikely fragments [175].

4.3.2.4 *Issues and Limitations*

Though Visual Studio's tools are very extensive, covering many different areas of analysis and quality assurance, there are some limitations. C#, VB, and C++ all have the option to run the above tools and analysis, although F# projects under Visual Studio do not support many of the above features. Like other languages, F# has IntelliSense, allowing compiler errors and warnings, as well as syntax errors, to be displayed in the code pane during development. Also, debugging is included in the F# projects. However, IntelliTrace is supported on an experimental basis [176]. The code performance analysis only gives a limited amount of information, e.g. only the graph for CPU usage over the wall clock. Another limitation for F# is

that the rule sets under code analysis will run but will not allow a user to configure its settings, meaning that no rules are actually run on an F# project. Finally, there is no menu option for metrics to be run or for code clone detection. These differences show that many of the techniques applied by Visual Studio are on a source code level or that, because F# is a functional programming language, they have been disabled due to either not being applicable, not being useful, or their inability to be applied in the first place.

4.3.2.5 *Techniques*

The following tables describe the quality assurance techniques extracted from Visual Studio 2012, broken down into the categories generated by the review of literature. The initial red box contains the name of the tool; following this, the techniques are divided into dynamic and static (see the relevant figure) analysis, following which the categories, in blue circles, further dictate the requirements of implementation until the techniques, in red squares, can be categorised.

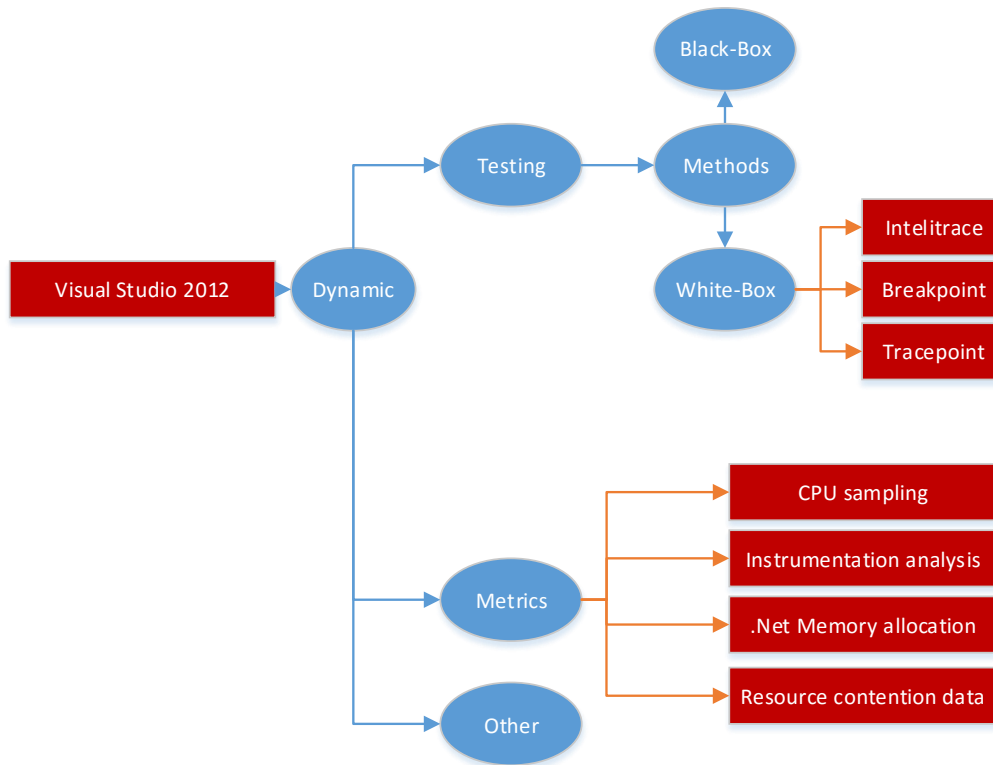


Figure 4.3.12 – Visual Studio 2012 dynamic analysis techniques

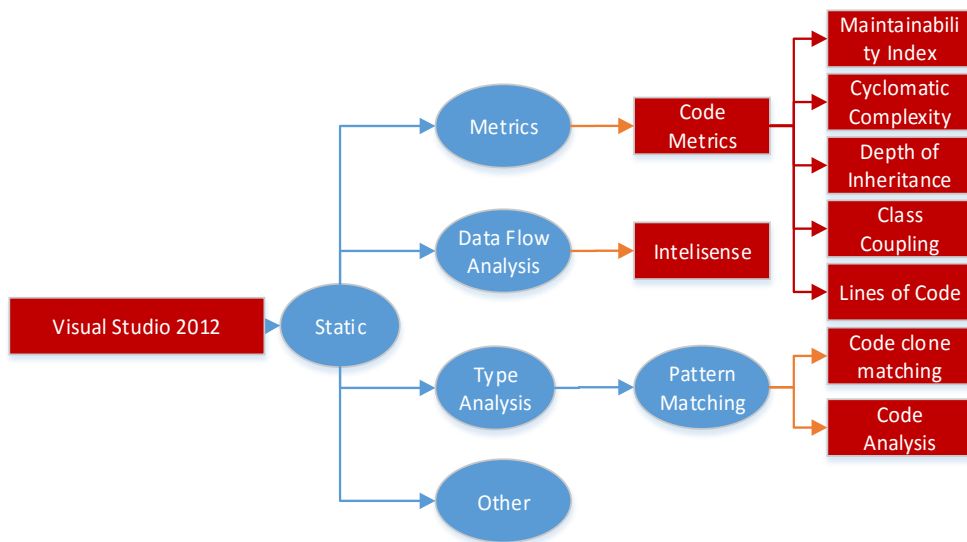


Figure 4.3.13 – Visual Studio 2012 static analysis techniques

4.3.3 Integrated Development Environment (IDE) Comparison

There are several articles that can be cited at this point regarding the differences in quality assurance that the developers of the IDEs have taken. It is important to note the background on which the IDEs are based. NetBeans is an open source application that is developed by the public but is also backed by Oracle [177], whereas Visual Studio is a commercial product that is owned by Microsoft; however, it can be expanded upon with plugins [178]. It can be seen by reviewing the areas covered that NetBeans and Visual Studio cover similar technique categories. Directly covered by both IDEs is the pattern matching with Code Analysis in Visual Studio and Java Hint in NetBeans as well as dynamic analysis testing, which is expected for most IDEs. However, this is where the similarities end, as NetBeans is highly reliant on plugins. Most of the plugins are directly integrated, but are still third-party maintained; on the other hand, Visual Studio has many more features built in as part of the IDE. This difference could be a direct mirroring of the open source vs commercial product approach; however, which is better is left to the individual to decide. For the purpose of this research, the openness of NetBeans has allowed for an improved in-depth analysis; however, Visual Studio's documentation made up for the closed garden that is Visual Studio's source code, which again is probably a reflection of the fact that this is a commercial product.

Another difference that can be drawn from the analysis for NetBeans and Visual Studio is the relation of quality assurance techniques between programming languages. The Visual Studios approach seems to have the same approach for each programming language and in some cases the exact same tools, e.g. performance analyser; however, as discussed in the further analysis, the depth of the similarities is unknown, as there subtle yet important differences. NetBeans, on the other hand, has no linking with each language unless the third-party plugins support multiple programming languages themselves. This makes Visual Studio a more important tool to study further for this research; unfortunately, with the limitation of no access to the source code, this further study cannot happen.

As initially outlined in this section, the main difficulties and differences of the IDEs are based in their backing; nevertheless, their differences permit an interesting and important study.

4.3.4 WinFPT

WinFPT is a quality assurance tool that has been used in various sectors to quality assure Fortran code. As FPT was developed over a decade ago, there are some differences with the way this quality assurance tool works when compared with some more modern tools. The internal representation and how it could be related to LIQA will be discussed, and the techniques that WinFPT uses will be categorised and included in the resulting taxonomy.

4.3.4.1 Internal Representation

WinFPT was developed when memory for programs was a finite and precious resource, with the result that the internal representation was developed to make the most of small amounts of memory. Second to memory, a major concern for WinFPT was speed, and the internal representation had to be streamlined to make it as efficient as possible.

As WinFPT is also designed to apply quality assurance techniques only to Fortran, and therefore could be specialised, the internal representation used is based on tables and the token stream generated from the source code. The tables contain data definitions, statements, symbols, linked lists, and files, and they replace tokens in the token stream with symbolic links to the tables. This internal representation is significantly less memory-intensive than more common modern internal representations such as ASTs. Another reason for this could be that ASTs require tree traversal, which is not a quick process in comparison with WinFPT's table and token stream combination [179].

Due to this internal representation being close to source code, some of the techniques may not be applicable to other programming languages, although converting the internal

representation to work with tables and the GASTM is possible and should be a simple process either during or after conversion to the GASTM, allowing those techniques, depending upon the tables, to be implemented within LIQA.

4.3.4.2 Features

The list of features for WinFPT is very extensive: please see the extended appendices for the full list. The major headings for the quality assurance techniques that WinFPT implements are listed below; these were provided by SimCon (the developers of WinFPT) [25].

- Metrics
- Error checking
- Formatting
- Structural engineering
- Run-time testing
- Optimisation
- Migration
- Security

Several areas will only be summarised and others omitted as out of scope; this is because the focus of the research identifies with several of WinFPT's categories, making them more important to study. These would be the ones aligning themselves with the QA categories that this research is focused on. As all of the metrics listed in the features are statically collected, these will be categorised under 'static' and 'metrics' in the taxonomy. Although formatting is something that LIQA could do, this would be on output following other forms of quality assurance. This also is a static form of quality assurance and will be categorised under 'Other', although, in hindsight, this should be a category in itself. Migration, which has ties with formatting, is, however, a major issue in Fortran code. As Fortran has been in existence since the 1950s, there are many versions, and this can cause issues with compatibility but, because migration is severely language-dependent, it is not going to be sufficiently generic to include within the experiments at this stage of the research and will therefore also be omitted from the taxonomy.

Run-time testing in this case is dynamic analysis, and WinFPT has two main features under this title, coverage analysis and trace data flow. Coverage analysis allows users to view all visited sections of code in a single run of the program, which is achieved by inserting code into areas of the program that cause a break in program flow and having them call an included file for recording. The same method is used for trace data flow, but this can capture variable states and values as well as inputs and outputs to allow for unit testing with various different values without having to reenact the full scenario. The capturing over variables also can be used for data checking against predicted data or for various testing methods.

Optimisation is another area in which WinFPT provides some techniques to improve Fortran programs. There are two provided: expanding routines in-line and unrolling DO loops. Expanding routines in-line is a compiler optimisation that can be activated in some programming languages, although this can be ignored by the compiler and sometimes is done automatically by the compiler. Providing this service externally makes it consistent. The technique is the process of replacing function calls with the full function, which can increase the speed of execution of a program, although it will increase the size of the program [180]. Unrolling DO loops is the second optimisation provided by WinFPT. Unrolling a DO loop takes a loop, checking the statements contained in its body to assess feasibility and then, if feasible, expanding the loop content into individual statements [181]. This technique increases performance but also increases the size of the source code proportionate to the number of statements the loop contains. Both of these optimisations fit into the taxonomy though not under a currently listed category, and therefore a new category, which will be entitled ‘optimisation’, must be added to align with these techniques.

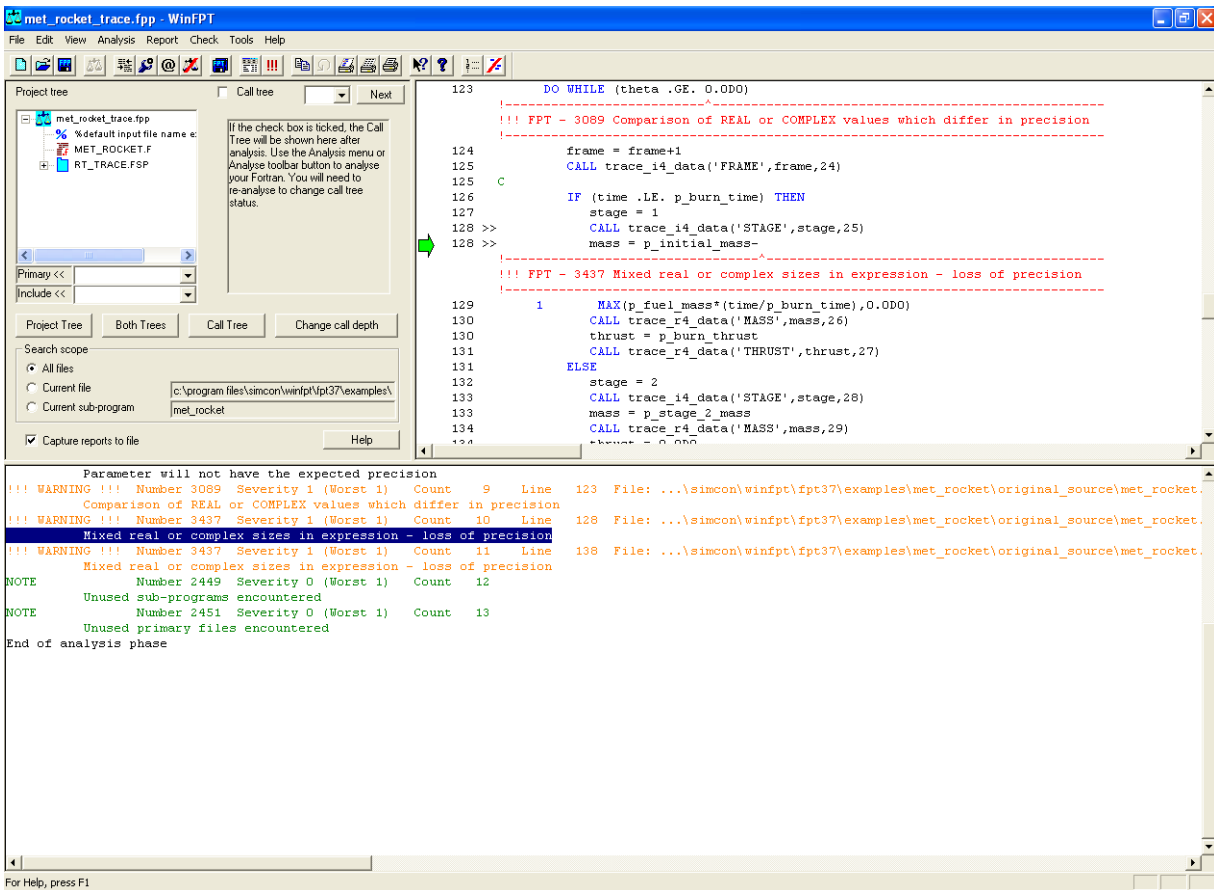


Figure 4.3.14 – Warning identification in winFPT

A further area that WinFPT introduces is ‘Structural engineering’ which, until this point, was not included in other quality assurance tools. The majority of this area is based on the direct manipulation of source code and, because of this, another category was added to the taxonomy under static analysis, thus accommodating these techniques. This category is ‘Code Manipulation’. The three main techniques within this category that WinFPT provides are declaration and names, COMMON Block, and control construct manipulation. Declaration and names have various sub-categories, but the techniques essentially manipulate variables throughout source code, adding declarations, modifying names, and data types, etc. COMMON block modification allows an automated service to extract COMMON blocks into include files and to move all static variables to COMMON, making code simpler to maintain. Control Constructs modify construct based code such as labels and conditional statements to make them more manageable and easier to read, e.g. replacing SELECT CASE with IF ... THEN ... ELSE

chains. Another technique classified under code manipulation, as well as under pattern matching, is the automated correction of some errors. These errors include correcting formats, e.g. commas and delimiters, as well as correcting inconsistent arguments that would result in the re-writing of arguments, or including type conversions to correct these issues. The final feature that WinFPT possesses under structural engineering is the ‘Interactive controller’ that allows users to monitor variables during a simulated run. It could be seen to be similar to a debugger and variable watcher combined with a unit tester and, for this reason, it will be classified under dynamic white-box testing.

The final category used by WinFPT to classify its quality assurance techniques is ‘Error Checking’, the techniques of which utilise either pattern matching or dataflow analysis. WinFPT uses dataflow analysis to search for its dead code and dead variables. Both of these techniques search for code that is essentially unused, be it an uncalled variable or method, or a section of unreachable statements. Pattern matching is used for detecting errors in names, errors in expressions, and mismatched data arguments. These techniques’ name encompasses a wide variety of smaller and more specific techniques, e.g. mismatched arguments that detect wrong data types, wrong data sizes, wrong array bounds, etc. All other expansions of the titles are in the full feature list for WinFPT.

As a commercial product, WinFPT must have key areas to make itself stand out from other products, especially free plug-ins. At this point, the key areas of WinFPT that are not covered by other tools are the structural engineering techniques that allow the user to make dramatic changes to large source code with minimum effort. Additionally, it utilises all other techniques in one package, which requires no modification to the source code.

4.3.4.3 Techniques

The following tables describe the quality assurance techniques extracted from WinFPT broken down into the categories generated by the review of literature. The initial red box contains the name of the tool; following this, the techniques are split into dynamic and static (see the relevant figure) analysis, following which the categories, in blue circles, further dictate the requirements of implementation until the techniques, in red squares, can be categorised.

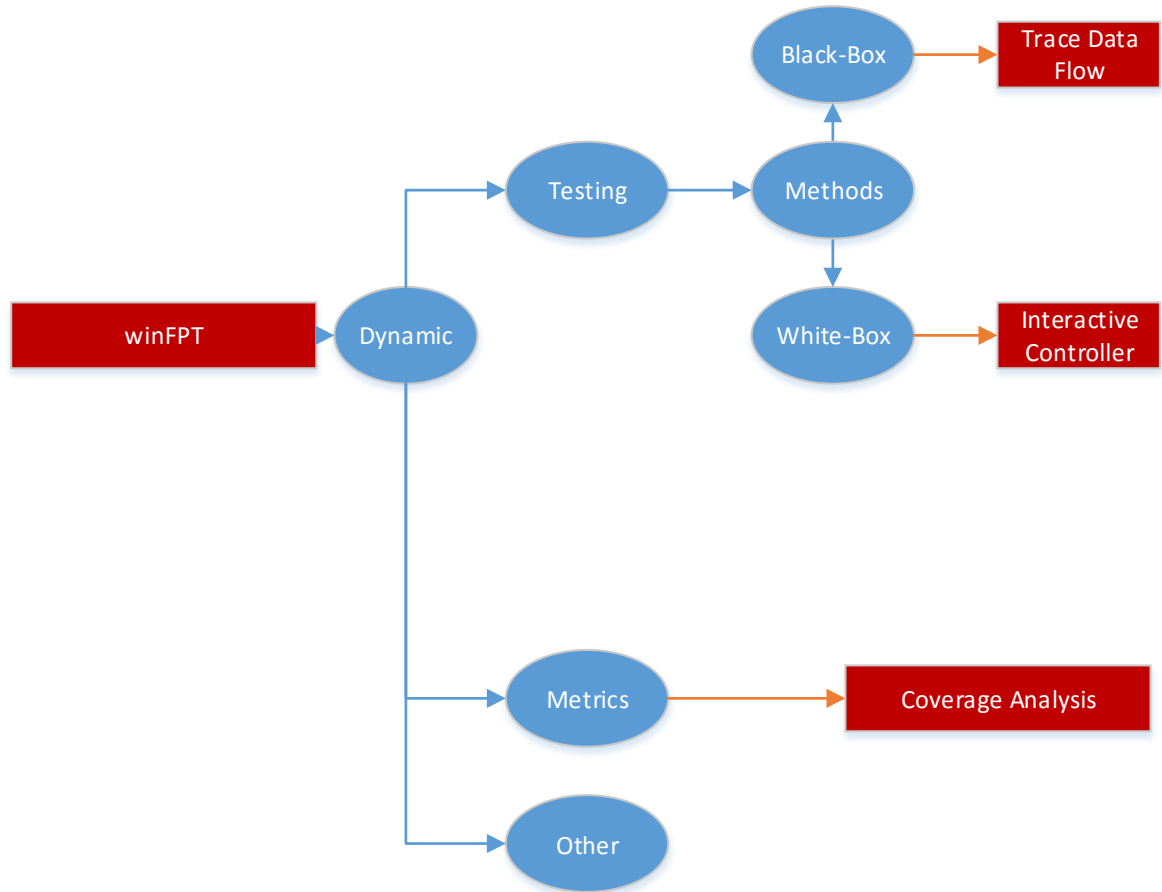


Figure 4.3.15 – Dynamic analysis techniques for winFPT

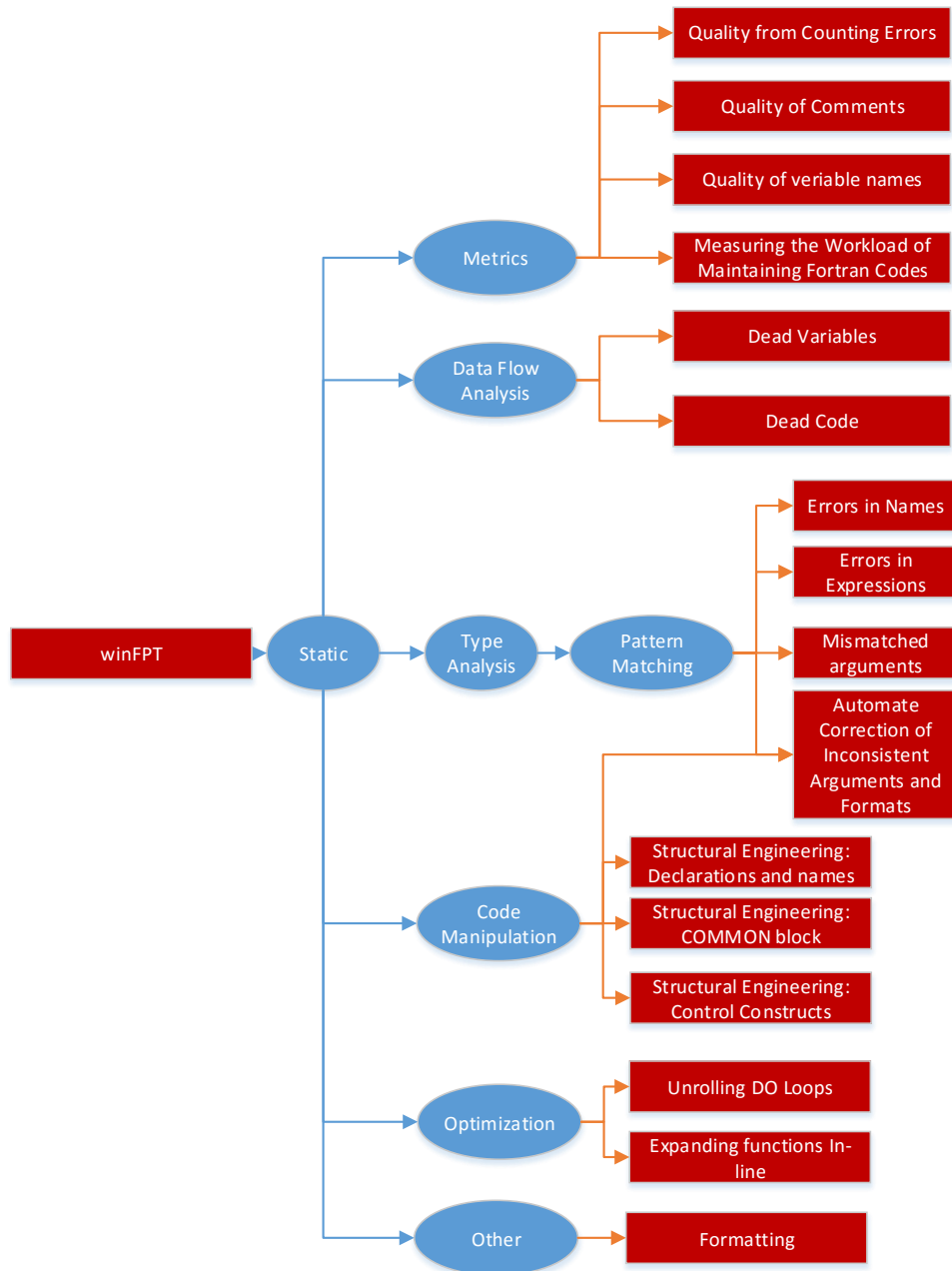


Figure 4.3.16 – Static analysis techniques for winFPT

Automating correction of inconsistent arguments is an exception to the rule of taxonomy, coming under more than one heading. This is because the technique has two stages: the initial finding of the inconsistent arguments would come under the Pattern Matching category, and the automated change of these arguments would come under the Code Manipulation category. This technique could therefore be split into two parts.

4.3.5 Polyspace

Polyspace, developed by MathWorks [58], is part of a wide suite of tools. Indeed, Polyspace is split into two tools itself, ‘bug finder’ and ‘code prover’, as shown in Figure 4.3.17. Though they look the same and provide some overlapping features, the intent of the tools is quite different. Notwithstanding this, they will be discussed as the single entity of Polyspace for simplicity and to avert repetition of features in the taxonomy [182] [183].

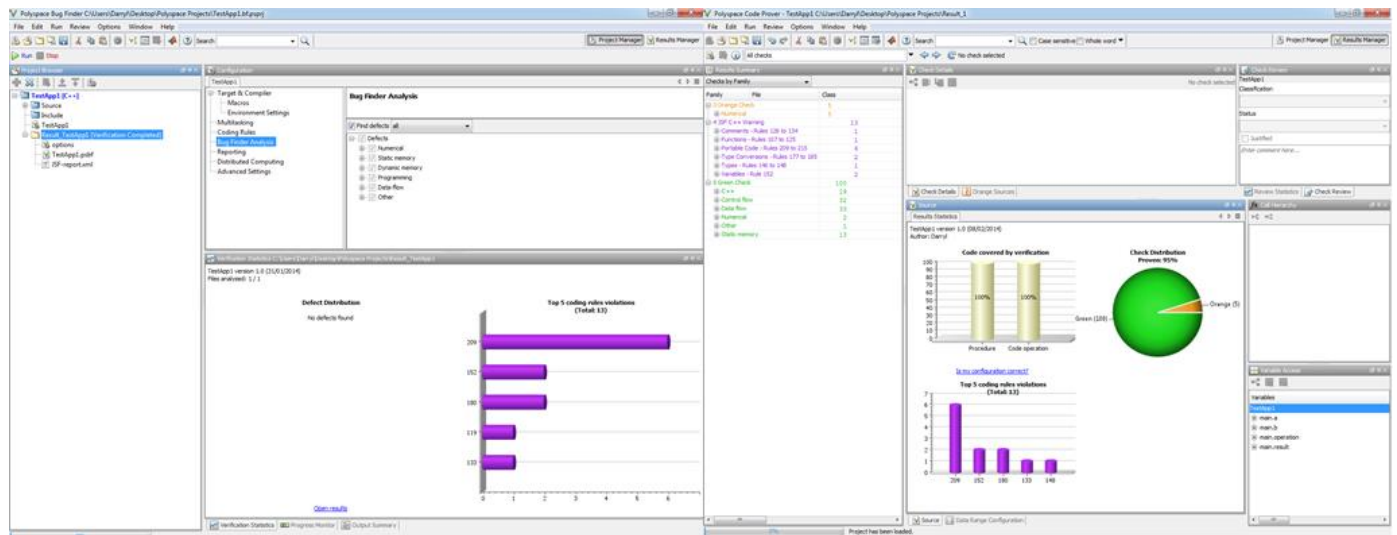


Figure 4.3.17 – Polyspace bug finder & code prover

4.3.5.1 Features

As this is a commercial product and its techniques and internals are not open for analysis, general viewing, or breakdown, the techniques used have to be inferred based on techniques currently used and observed as well as documentation. There are a few features that will not be included in this project due to time constraints and also because the features usually are specific to the toolsets within MathWorks, e.g. tracking issues back to the Simulink model (Simulink being the model-based design software from MathWorks). Polyspace uses formal methods and abstract interpretation to assess C and C++ source code for quality. The tool implements static

analysis to perform run-time error checks, quality metrics, automated code verification, artifact generation, and several other techniques.

As with an IDE, Polyspace detects run-time errors in the form of highlighting source code where issues arise. The system identifies four types of code: run-time error free, faulty under operation, unreachable, and faulty under certain conditions. This technique for identifying run-time errors is used in a wide variety of QA tools as well as IDEs, and the technique usually used to achieve this is data-flow analysis. Polyspace also uses this technique to compute variable values determining a range of possible values through abstract interpretation, which allows users to view these ranges to ensure that no expected border limits are broken.

Polyspace includes software metric retrieval via simple static analysis, with these metrics ranging from Cyclomatic complexity to Hersteller Initiative Software Metrics. See below for a complete list:

- Cyclomatic complexity
- Comment density
- Call levels
- Number of paths
- Number of function parameters
- Hersteller Initiative Software (HIS) metrics

As well as metrics, pattern matching is used to define rules for coding standards. As well as the standards listed below, Polyspace allows users to write their own rules based on specific code violations:

- MISRA-C:2004,
- MISRA-C++:2008,
- MISRA-AC-AGC,
- JSF++,

- Custom naming coding rule violations

Finally, Polyspace has integration with the IEC Certification Kit and the DO Qualification Kit for artefact generation, simplifying the certification processes for all the certifications listed below:

- ISO 26262

‘ISO 26262 is intended to be applied to safety-related systems that include one or more electrical and/or electronic (E/E) systems and that are installed in series production passenger cars with a maximum gross vehicle mass up to 3 500 kg.’ [184]

- IEC 61508

‘The IEC 61508 series are the International Standards for electrical, electronic and programmable electronic safety related systems. It supports the assessment of risks to minimise these failures in all E/E/PE safety-related systems, irrespective of where and how they are used.’ [185]

- EN 50128

Is the European standard EN 50128 ‘Railway applications - Communication, signaling and processing systems - Software for railway control and protection systems’ [186]

- IEC 62304

Is the international standard for medical device software - software life cycle process. [187]

- DO-178C

‘Providing guidance for the production of software for airborne systems and equipment that performs its intended function with a level of confidence in safety that complies with airworthiness requirements.’ [188]

- DO-278A

‘DO-278 provides guidelines for the production of software for ground based avionics systems and equipment that performs its intended function with a level of confidence in safety.’ [189]

Although this does not appear to be a great number of features compared with other quality assurance tools, only features discussed above are within the scope of the present investigation. As a commercial product, additional features link Polyspace with its containing suite, with strong ties to software models, especially Simulink [190]. Other features include automated checks linked in with the build procedure and exporting of reports in HTML, for large software models. The analysis can be performed using distributed computing over a grid. These features are impressive, useful, and therefore worthy of mention; however, they are beyond the scope of this project.

4.3.5.2 Techniques

The following tables describe the quality assurance techniques extracted from winFPT separated into the categories generated by the review of literature. The initial red box contains the name of the tool; following this, the techniques all come under static analysis, as this tool does not provide dynamic analysis, following which the categories, in blue circles, further dictate the requirements of implementation until the techniques, in red squares, can be categorised.

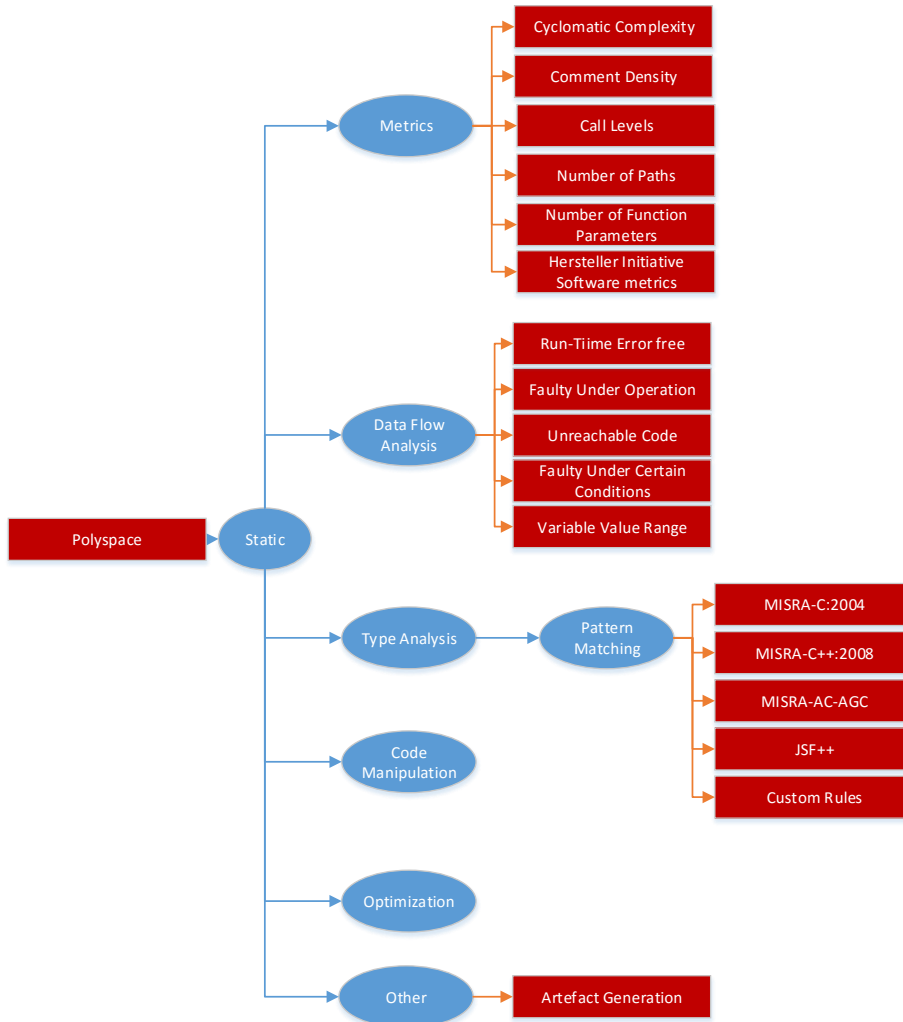


Figure 4.3.18 – Polyspace static analysis techniques

4.3.1 Critical Comparison

There are several major differences in the focuses and approaches each of the tools have taken with regards to automated quality assurance. Polyspace focuses on static analysis; this is because it is part of a suite of tools that, when used together, is more akin to the IDEs than WinFPT. In other words, the suite is used for development from start to finish of a project, whereas WinFPT extends into the realm of dynamic analysis; however, this is not to the extent that the IDEs do (in regards to number of quality assurance techniques). The lack of dynamic analysis of WinFPT could be due to the approach taken by the company that produced it, which is that WinFPT can be customised and added to by SimCon (the holding company) [179], allowing a more custom and less all-encompassing approach that the other tools have taken. Direct code modification and optimisation have not been included in the IDEs or Polyspace, whereas WinFPT does have a variety of features covering these areas. This is with the exception of the IDEs' approach to solving problems highlighted by other forms of analysis, which is useful but should not be considered automated. Polyspace is the only tool here that boasts integration with international standards kits, which may indicate the appropriate target audience of the tool. This is interesting because WinFPT is designed to work with all Fortran applications including those in aerospace and nuclear fields. However, it does not support the international standards that Polyspace does; but again, SimCon could have custom builds for those industries. All things considered, these tools, though very different, have overlapping features and techniques that have identified areas that this research needs to review, including the taxonomy to help refine the categories.

4.4 TECHNIQUES

This section is a summary of the analysis of tools with regards to the techniques and the initial categories of implementation generated from the literature study. The differently coloured boxes represent which tool the technique comes from. Some of the techniques serve the same purpose; in that scenario, the techniques are shown one after the other. Figures 4.4.2 to 4.4.4 represent categories under static analysis, with Figure 4.4.5 being dynamic analysis. At this point, the categories are not finalised and will require modification for several reasons.

Currently, there is no universally agreed definition for each category; also, several techniques or technique groups span over 2 categories, which will not be permitted with the final taxonomy. With few issues, the original categories stand as an effective representation of the implementation requirements for many quality assurance techniques.

4.4.1 Diagram Key



Figure 4.4.1 – Tool key

4.4.2 Static

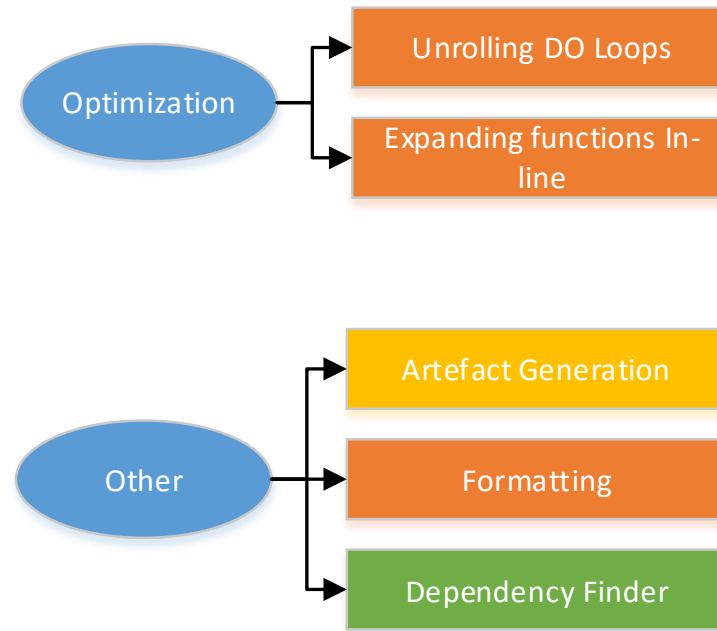


Figure 4.4.2 – Static analysis 1

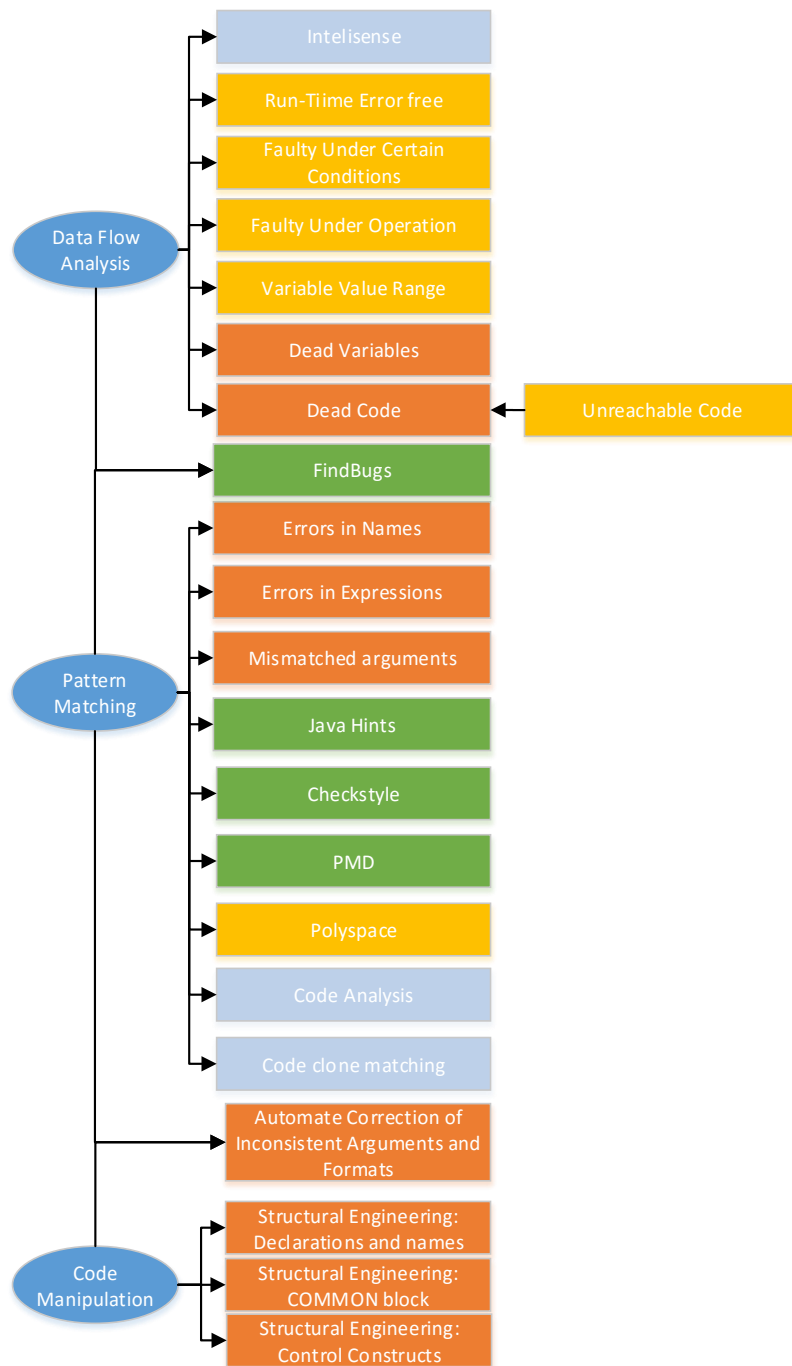


Figure 4.4.3 – Static analysis 2

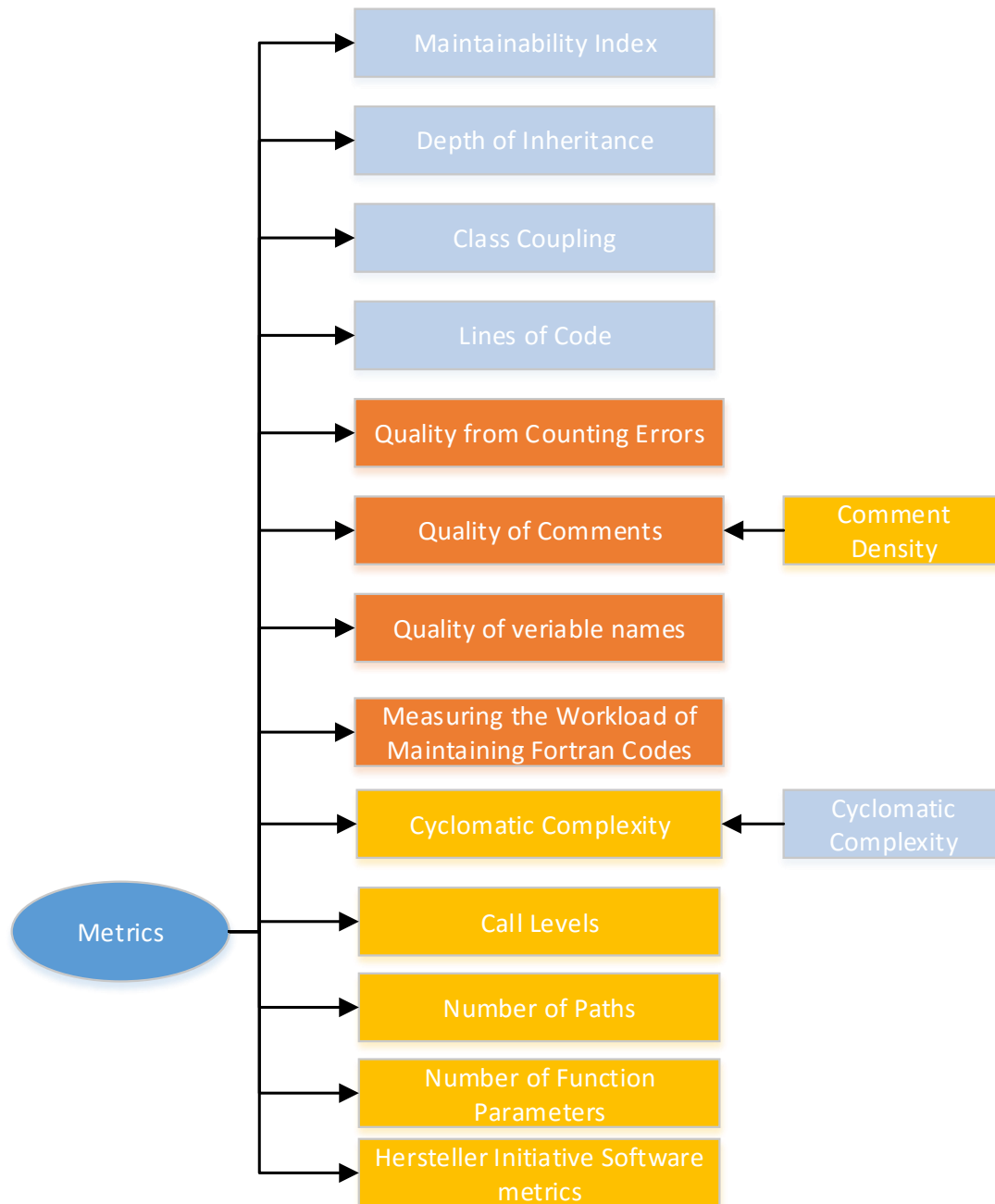


Figure 4.4.4 – Static analysis 3

4.4.3 Dynamic

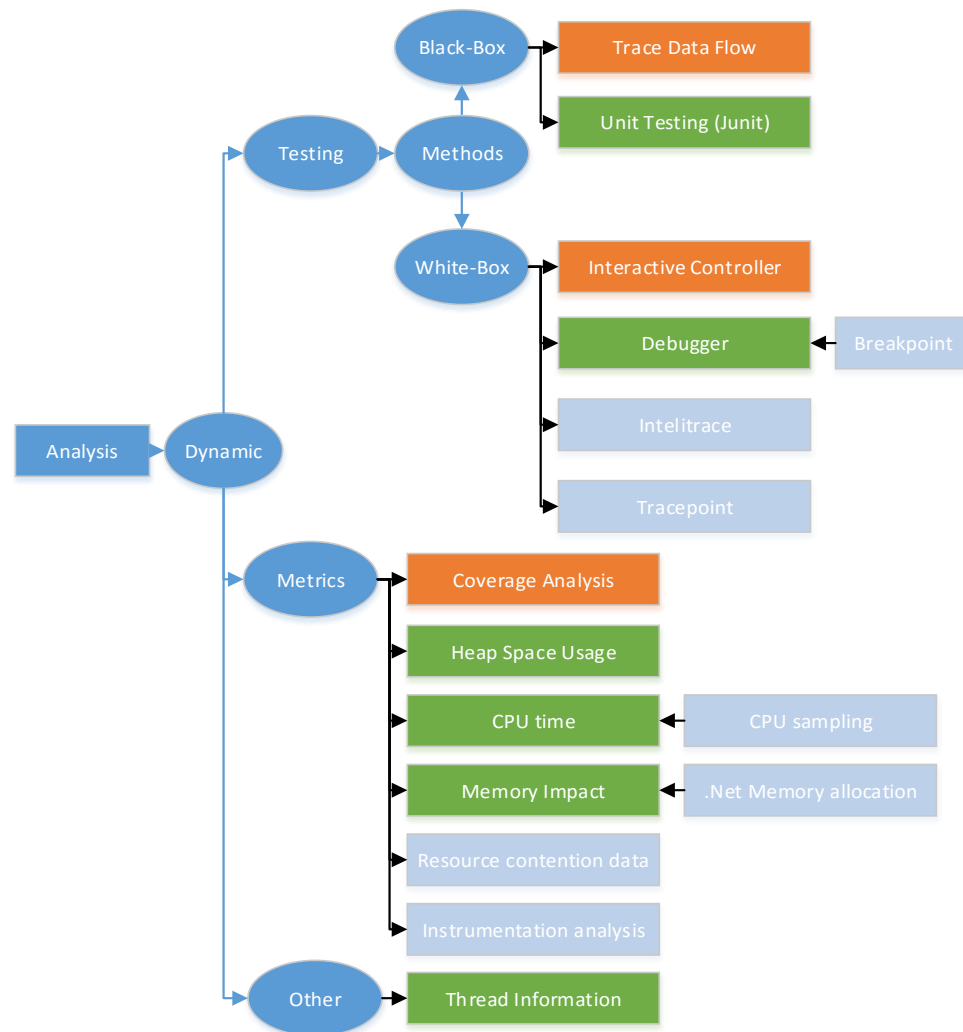


Figure 4.4.5 – Dynamic analysis

4.5 TAXONOMY OF TECHNIQUES

With the completion of detailed analysis, the full taxonomy should be available; however, several areas remain to be addressed first. One such area is the category ‘Other’ that was included in both static and dynamic analysis types. This was intended to collate all of the individual techniques that could not be categorised immediately such as formatting, artefact generation, etc.

The more apparent change to this taxonomy is the inclusion of a different parent node in the form of generation. This was created to include features that utilise other analysis techniques but do not add any new information. It was included as it is an important part of automated quality assurance but cannot be categorised under analysis.

The techniques that were categorised as ‘Other’ were artefact generation, formatting, and dependency finder. Artifact generation, a technique implemented by Polyspace, is the creation of artefacts for standards submission. After considering this, it was found to be a technique in itself. Another overlooked technique that should have been included under the artefact generation category is JavaDoc included in NetBeans, as this auto-generates API documentation for the project. Formatting is a similar technique as it is included in WinFPT, but it should be a category as several other tools, including Netbeans and Visual studio, allow the user to format the code to the standard for the language being written. winFPT’s formatting technique allows for multiple Fortran standards, as this is a more significant issue in the Fortran language. Finally, the dependency finder technique included in NetBeans allows for the visualisation for the class dependencies and library dependencies. There are several other tools that visualise data and representations, but this is usually done without highlighting it is a technique, leading to the decision for the inclusion of the visualisation category. These new categories are added to the final taxonomy, as shown in Figure 4.4.6.

A final discussion point concerns two of the techniques included, ‘Findbugs’ and ‘automated correction of inconsistent arguments and formats’, which have been placed under

two categories. Findbugs was included under two categories because it is not a technique itself, but rather within the tool using both types of technique. However, due to the number of techniques, it was inappropriate to list all of them. With regards to automated correction of inconsistent arguments and formats, this is a feature and includes two techniques, identification and correction.

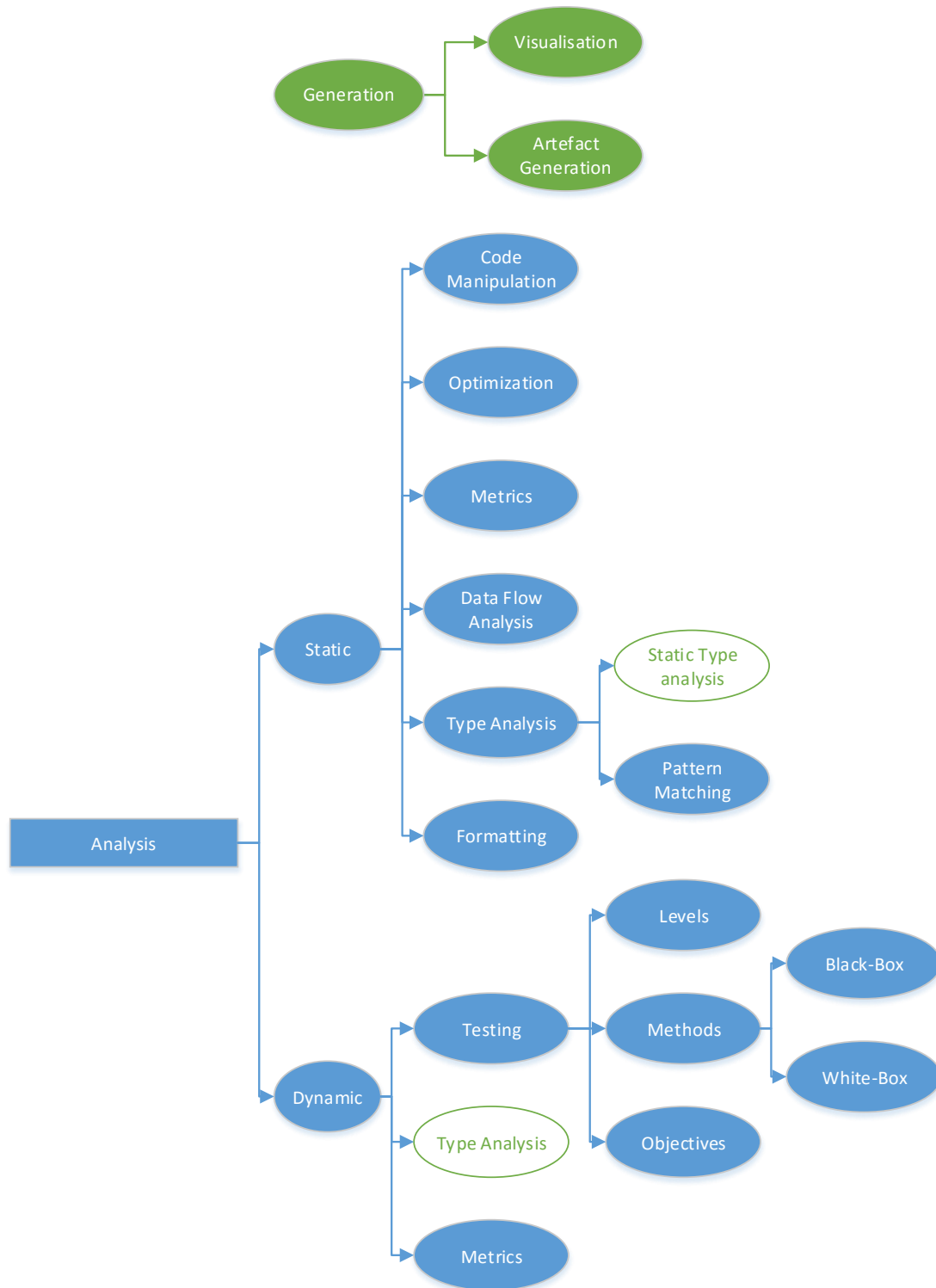


Figure 4.4.6 – Taxonomy of quality assurance techniques

4.6 ADDITIONAL TOOLS

Several tools have been found in addition to those used to create the taxonomy. Some of these tools are nevertheless relevant to this research, and this section will overview these tools and the reasons why they are important to assess in order to identify links to the aims of this research.

Sonarqube [63] boasts that ‘More than 20 programming languages are covered’ and that their platform is designed to manage code quality [63]. The high number of programming languages and the single front end may suggest a generic representation to which the techniques are applied, which would be the same as in this research. However, analysis of the documentation reveals that each language is a different plug-in [191], but on the other hand, Sonarqube allows multiple programming language analysis in a single project [192], which nevertheless suggests some common representation with a single implementation of techniques. Contrary to this, however, Sonarqube is an open project and, on review of the project files, it can be seen that each plug-in has techniques built in at that level [193], meaning that there is insufficient generalisation in the representation used so that techniques can be written independently of the programming language.

The DMS Software Reengineering toolkit was another tool identified as having significant similarities to this research. At first glance, the outlined tool has a passing resemblance to the framework and the aim is also very similar [194], having a tool that can be used on an extensive number of programming languages using only a grammar and preferences for what looks like generic analysis techniques [195]. The tool, however, has several specific differences from the framework proposed by this research. The internal representation used by DMS is based on hypergraphs as its internal representation instead of abstract syntax trees. Similarly to WinFPT, DMS justifies not using abstract syntax trees due to the increase in memory use and processing power required to generate and navigate an abstract syntax tree; this outweighs the benefits. Another reason for using hypergraphs over abstract syntax trees is the inclusion of graph programming languages [196]. The generic techniques that appear to be

implemented within this tool are questionable, due to the inclusion of different available techniques for different programming languages of a similar nature, for example C#, Java, and Visual Basic. Also, the link for the customisable analyser is different for each programming language; some of the links just send users back to the product description, so the extent to which the code analysis is the same for each programming language is also questionable [197].

Both of these tools demonstrate a lack of generic quality assurance. Another factor to consider is the openness of this research. As a standard is being used for the internal representation, anyone can implement a technique upon this or write a parser for a programming language because the specification is available. The generic nature and openness of this research are key ideals allowing development to take place externally from a centralised point.

4.7 CATEGORY SUMMARY

The reason this taxonomy has been created is to categorise the quality assurance techniques via their implementation, allowing for a simple assessment to ascertain whether the framework and further, LIQA, could implement these techniques in a generic form. This taxonomy also adds to the novel content of this research. The taxonomy can be seen to be correct as each category has a vital component that all techniques under it must contain. In order to be implemented effectively, these commonalities are listed below:

Generation – Containing visualisation and artefact generation. These techniques of quality assurance have been included in their own section and highlighted because they are important to automated quality assurance, although they are essentially ways of viewing other techniques' results in a summary or format that is simpler to digest.

Code Manipulation - Needs to be able to modify or replace source code constructs. These must be applied on an individual case, as this is for engineering purposes or for additional functionality.

Optimisation – Needs to be able to modify or replace source code constructs. These can be applied in an automated fashion as they are intended to increase program performance.

Metrics – Utilises counting of constructs within source code, e.g. operands, operators container, etc.

Data Flow Analysis – Stems from the formulation of a control flow graph.

Static Type Analysis – Access to expressions and all variables used to evaluate if a program is type safe.

Pattern Matching – Matches patterns from previous examples against source code, e.g. final method in final class.

Formatting – Modifying whitespace but not source code.

Dynamic Metrics – Counting mechanisms contained within running code.

Dynamic Type Analysis – Ability to generate stores for used variables and to backtrack to assess whether expressions are valid.

Dynamic Testing – The different sub-categories of testing were discussed in detail above but essentially require manual interpretation.

Some of these techniques could be argued as being part of a different category or as being placed in a specific sub-category. However, as this is the first iteration of the taxonomy and not all techniques have been covered, it could be improved with further work. An important part of this work, however, is to evaluate the taxonomy and have a provable structure. This will be presented in the next section.

4.8 EXPLICIT TAXONOMY

As this taxonomy is novel, it needs to be valid and accepted. To achieve this outcome, it has to be provable. In attempting this, the rules that guide the categorisation of the techniques need to be explicit, which will also assist third parties who may wish to use the taxonomy in order to categorise further techniques and expand on the taxonomy.

The rules were created by initially identifying implementation requirements. A simple example is static or dynamic requiring a program to be running or not, which allowed the separation of generation because it does not require access to the program in a static or dynamic state as it uses extracted data (from other analysis techniques). This is the same with regards to the further categories, splitting techniques based on the access requirement, e.g. access to source code level or representations. This would all the techniques under that category to be feasible based on whether that access would be available from the internal representation that this framework provides. Finally, some categories required refinement based on the techniques not being representative, e.g. testing split into levels, methods, and objectives because even if a level could be achieved, this does not mean that a method could be accomplished.

Whilst creating these rules, several changes were made to the taxonomy in order to correct some minor issues. These issues have not affected the techniques in the categories and therefore the rest of the research based on this taxonomy also remains unaffected. However, to allow the taxonomy to stand in its own right, these changes needed to be made.

One such change was the removal of ‘pattern matching’ and ‘static type analysis’ from their supercategory, ‘Type Analysis’. These were separated and ‘static type analysis’ removed. This took place because ‘pattern matching’ did not need the restrictive rules of type analysis; at this point, ‘type analysis’ and ‘static type analysis’ were not disparate, and therefore ‘static type analysis’ was not necessary. The second change was the addition of the category ‘adjust constructs’ under ‘analysis’ → ‘static’, and moving both ‘optimisation’ and ‘code manipulation’ so that they came under this category. This was done as there are particular similarities between these two categories, but there is nevertheless a need to separate them due to the reasoning for using techniques under the specific categories. The resulting taxonomy is shown in Figure 4.4.7.



Figure 4.4.7 – Taxonomy of quality assurance techniques with rules

The rules set for this taxonomy are provided in a top-down order and are as follows:

- Analysis
 - Is a quality assurance technique
 - Generation
 - Uses one or more quality assurance techniques
-

- Dynamic (Analysis)
 - Is performed during runtime
 - Static (Analysis)
 - Is performed on source code
 - Visualisation (Generation)
 - Uses graphics to represent data
 - Artefact Generation (Generation)
 - Creates documents based on techniques
-

- Metrics (Dynamic→Analysis)
 - Collecting numeric data based on a program property
- Type Analysis (Dynamic→Analysis)
 - Recording and comparing type data in expressions
- Testing (Dynamic→Analysis)
 - Using test data to predict and evaluate output
- Formatting (Static→Analysis)
 - Adjusting whitespace, tabs, and newlines
- Type Analysis (Static→Analysis)
 - Evaluation type on expressions
- Data Flow Analysis (Static→Analysis)
 - Requires a control flow graph or equivalent
- Metrics (Static→Analysis)
 - Collecting numeric data based on a program property

- Pattern Matching (Static→Analysis)
 - Searching against a template
 - Adjust Constructs (Static→Analysis)
 - Modifies constructs
-

- Code Manipulation (Adjust Constructs →Static→Analysis)
 - Does not increase performance
 - Optimisation (Adjust Constructs →Static→Analysis)
 - Increases performance
 - Levels (Testing →Dynamic→Analysis)
 - A targeted section of the development
 - Objectives (Testing →Dynamic→Analysis)
 - A type of testing
 - Methods (Testing →Dynamic→Analysis)
 - A way of testing
-

- Black-Box (Methods →Testing →Dynamic→Analysis)
 - Able to see source code during testing
- White-Box (Methods →Testing →Dynamic→Analysis)
 - Based on inputs and outputs only

4.9 SUMMARY

This chapter's purpose was to break down the area of automated software quality assurance into individual techniques, with the aim of creating a taxonomy based on the methods required for implementation. This will allow the framework to be evaluated over a larger scope, by assuming that if one or more techniques from a category can be implemented, all of the techniques in that category could be built on top of the framework. The next stage of this research is to identify which techniques should be implemented into LIQA for evaluation and then proceeding with the implementation. The taxonomy shall act as part of the original contribution to knowledge.

Chapter 5. Framework

In this chapter, the technique development within LIQA upon the GASTM and the testing and evaluation of LIQA will be discussed, as well as beginning with LIQA from the previous development, via initial testing, to ensure smooth progression into technique development. The chapter will continue with a discussion of the techniques and their varying levels success of implementation within LIQA using this framework. This aims to fulfill **A2O2** – Construct a working skeleton framework, specified in the introductory chapter. Additionally, **A2O3** – Assess the framework based on implemented automated SQA techniques, against calculated values and results generated by other automated SQA tools, will be addressed.

5.1 IMPLEMENTATION OF TECHNIQUES

This section describes the techniques that were implemented into LIQA as a proof of concept for the overall framework. The techniques chosen are from the taxonomy of techniques either from literature or the tools analyses. Only a few techniques from each category in the taxonomy have been chosen, as those groups under the same category use the same or very similar processes to facilitate their technique. For example, implementing all of the static metrics category would require 41+ techniques, whereas a sample of 3 would be just as effective in proving the feasibility of that category. These techniques are shown in Figure 5.1.1.

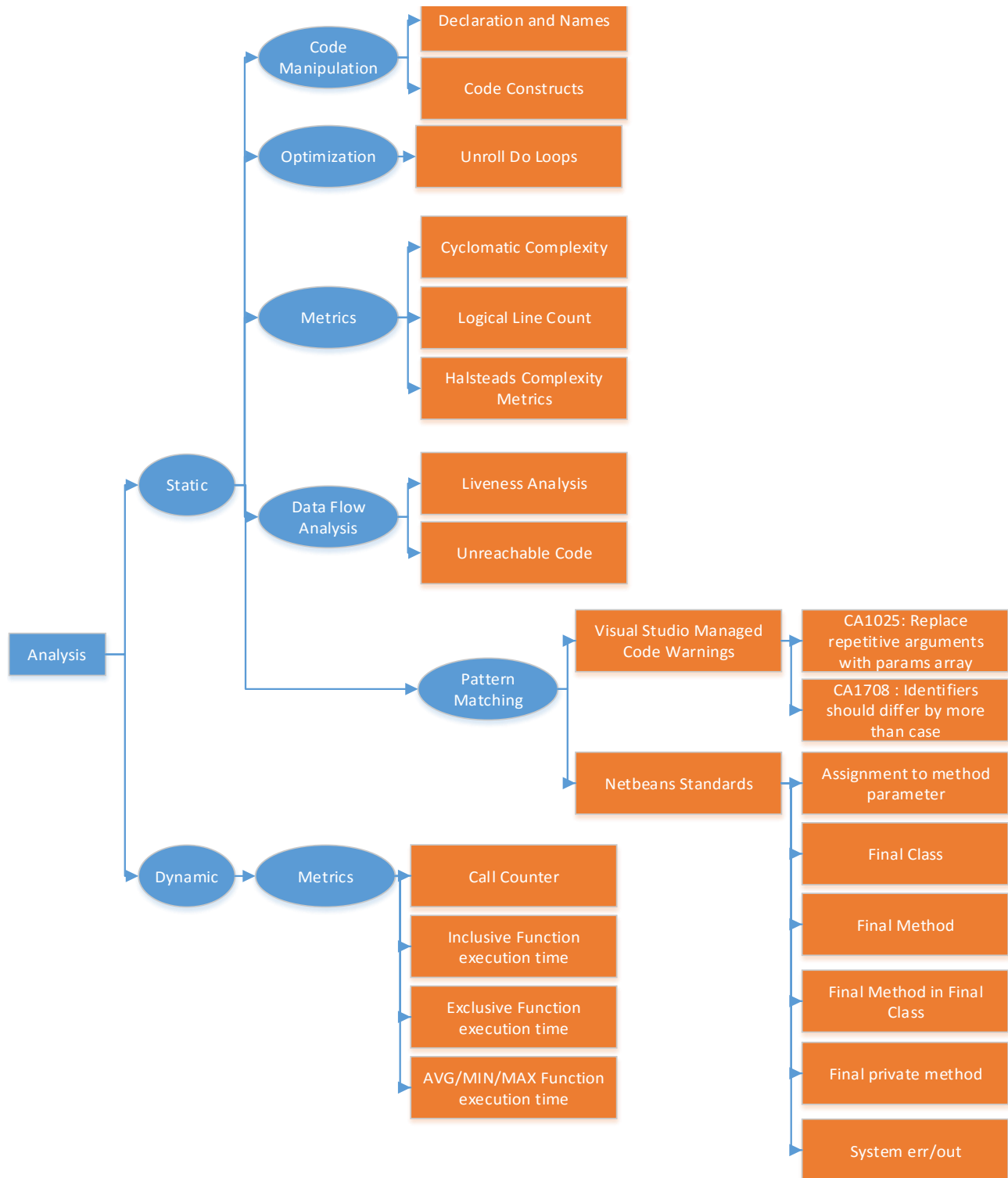


Figure 5.1.1 –Quality assurance techniques being implemented

5.1.1 Static Analysis

This section will cover the static analysis techniques being implemented for demonstration of readability for their respective categories. This will include the categories:

- Code manipulation
 - Declaration and names
 - Code constructs

Of the three code manipulation techniques found during the tools analysis, these two were relevant with regards to Java, whereas the third technique was more language-specific; therefore, these two have been selected as the sample:

- Optimisation
 - Unrolling a loop

Of the two forms of optimisation in the taxonomy, the following one was chosen at random:

- Dataflow analysis
 - Liveness analysis
 - Unreachable code/Dead code

During the development of the representation used for dataflow analysis (the control flow graph), these were key indicators of it being functionally correct; therefore, the sample is based on the critical path of the dataflow analysis implementation:

- Static Metrics
 - Cyclomatic complexity
 - Logical line count
 - Halstead's complexity metrics

These metrics have been chosen as they are a representative sample based on popularity in literature and in the tools analysis:

- Pattern matching
 - Final class
 - Final method
 - Final method in final class
 - Final private method

All of the above represent a set of pattern matching techniques that have a similar goal and therefore can be seen as a single technique:

- `System.out/err`

The above techniques are from NetBeans, whereas the techniques detailed below are allocated from Visual Studio, giving a cross-section for the sample of pattern matching:

- Replacing repetitive arguments with a params array
- Identifiers must differ by more than case

5.1.1.1 Code Manipulation

Very few techniques directly modify code because it is difficult to understand the semantics of a program. Usually, automated techniques inform by highlighting certain code or indirectly inform through reports of various kinds. In this case, code manipulation has been included to demonstrate that IR can be modified without conversion back to a programming language and also to demonstrate the techniques implemented in this category.

Declaration and names

Code manipulation for names is the automation of name changes. This permits changing names for classes, methods, and variables throughout a single project. This requires the IR to be walked so that Classes, methods, and variables can be identified and with their individual scope. Following identification, user selection determining changes will occur, then committing those changes to the declaration of the structure but also the references to that particular declaration.

An example of this is shown in Figure 5.1.2 where ‘i’ has been replaced with ‘count’.

<pre>int i; print(arr); for (i = 0; i < arr.length - 1; i++) { if (arr[i] > arr[i + 1]) { tmp = arr[i]; arr[i] = arr[i + 1]; arr[i + 1] = tmp; } } int c; for (c = i; c > 0; c--) { if (arr[c] < arr[c - 1]) { tmp = arr[c]; arr[c] = arr[c - 1]; arr[c - 1] = tmp; } else { break; } }</pre>	<pre>int count; for (count = 0; count < arr.length - 1; count++) { if (arr[count] > arr[count + 1]) { tmp = arr[count]; arr[count] = arr[count + 1]; arr[count + 1] = tmp; } } int c; for (c = count; c > 0; c--) { if (arr[c] < arr[c - 1]) { tmp = arr[c]; arr[c] = arr[c - 1]; arr[c - 1] = tmp; } else { break; } }</pre>
---	--

Figure 5.1.2 – Variable rename example

To demonstrate a working example of this technique, LIQA will have the variable rename implemented, but not the method or class rename. This is only due to class and method name manipulation being very similar and would not demonstrate any other technique.

To implement the variable rename, the QA.Identifiers package was created and populated with a recording system, pulling out class, method, and variable declarations from the IR during a tree walk. The Classes under Treewalkers.Retrivers.Declarations were implemented to perform the walk of the IR. Finally, the Classes under Treewalkers.Modifiers.CodeManipulation.VariableRename were implemented to traverse the IR and perform the rename.

An additional button under the IR generation GUI had to be implemented to rescan the IR, after changes, to create the XML and graphics for testing.

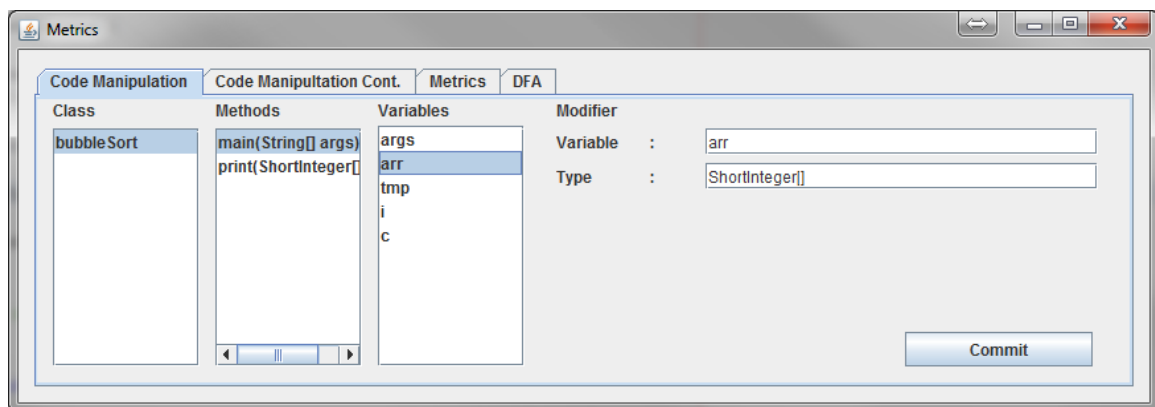


Figure 5.1.3 – Graphical User Interface (GUI) for variable rename

Code constructs

Code constructs are a technique implemented allowing users to modify source code from a construct to a preferred construct. This was also derived from WinFPT, and allows users to convert switch statements to case statements. This ideally is used for adding secondary conditions in extensive condition statements, making it simpler than re-writing the entire statement. It is of note that, in NetBeans, a technique implemented allows if and if else statements that only compare a single String to String literals can be changed into a switch by the IDE using its auto-correction via hints [198]. This technique will not be implemented due to time

constraints, as the techniques are, on a basic level, interchangeable. An example of Switch to if is given below.

<pre>int i; int math; math = read(); i = read(); switch (i) { case 1 : math++; case 2 : math += 2; } write(math);</pre>	<pre>int i; int math; math = read(); i = read(); if (i == 1) { math++; } else if(i == 2) { math += 2; } write(math);</pre>
---	--

Figure 5.1.4 – Switch to If example

5.1.1.2 Optimization

This implementation was derived from WinFPT, allowing users to convert a static loop into a sequence of statements. This is used for optimisation to stop the CPU calculation for each loop, although its use is very specific; an example of this is given in Figure 5.1.5.

<pre>int i; for (i = 0; i < 5; i++) {print(i + "");}</pre>	<pre>int i; print(0 + ""); print(1 + ""); print(2 + ""); print(3 + ""); print(4 + "");</pre>
---	--

Figure 5.1.5 – For unroll example

5.1.1.3 Data Flow Analysis

Data flow analysis is a more specific name for a supposedly wider fundamental set of techniques, including abstract implementation using formal methods and lattice theory. The reason this has been incorporated into the data flow analysis category is that these independent techniques are rarely used outside of data flow analysis, with regards to automated quality assurance. Data flow analysis utilises the control flow graph representation that has been generated from the IR of the framework, as discussed below.

Control Flow Graph

The control flow graph was the basis to facilitate several techniques and therefore was essential to demonstrate. There are two main types of control flow graph, with the difference between the two being the basic nodes. Either there is one statement per basic node or a block of statements with no deviation from the flow, i.e. no condition or iteration. The advantage of using a single statement per block is that it is simple to implement, and from this, the second type of CFG can be generated. The advantage of multiple statements per basic block is the reduction in analysis time for larger programs, which suggests that a combination of the two would be the most effective. However, in this case, only small programs are going to be used in testing, and the fact that the simpler CFG can be converted to use lists of statements means that adding this feature into LIQA would involve a significant addition of time without any significant advantage being gained. Therefore, the simple version of the CFG will be generated from the IR in LIQA [199].

Generating the CFG from the IR in a Java format required the creation of two objects, CFGblock and CFGedge. CFGblock stores the predecessors and successor edges as well as the block ID and statement it represents. The CFGedge stores the link between two blocks as predecessor and successor as well as a label for conditions. An example of how these classes will be used can be found in Figures 5.1.6 and 5.1.7.

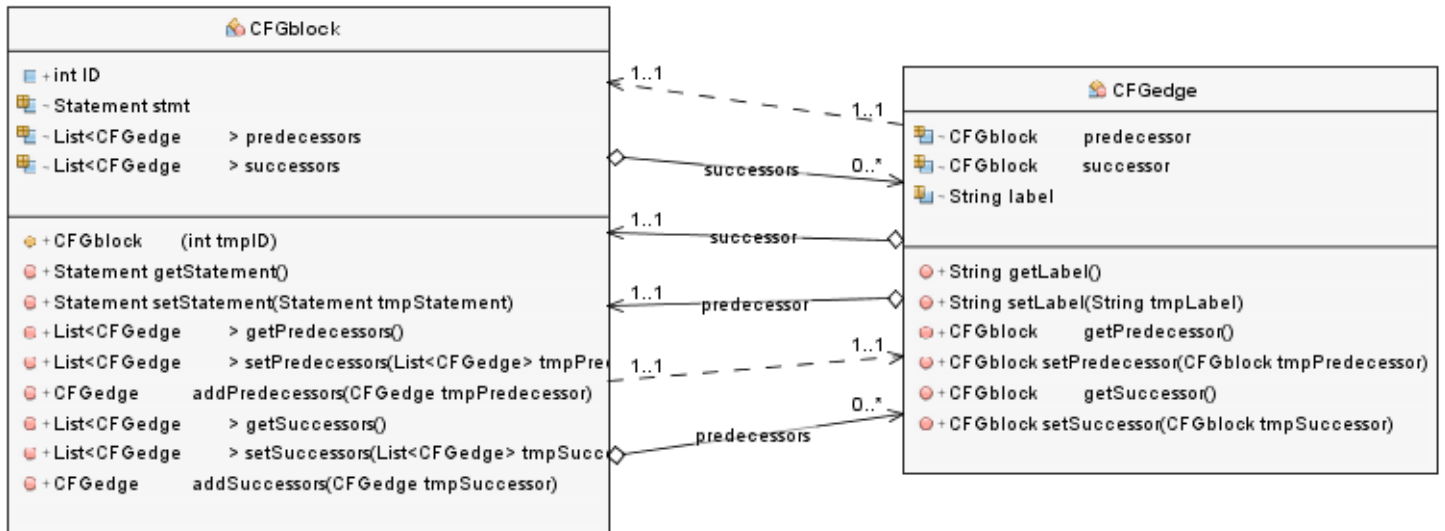


Figure 5.1.6 – CFGObjects class diagram

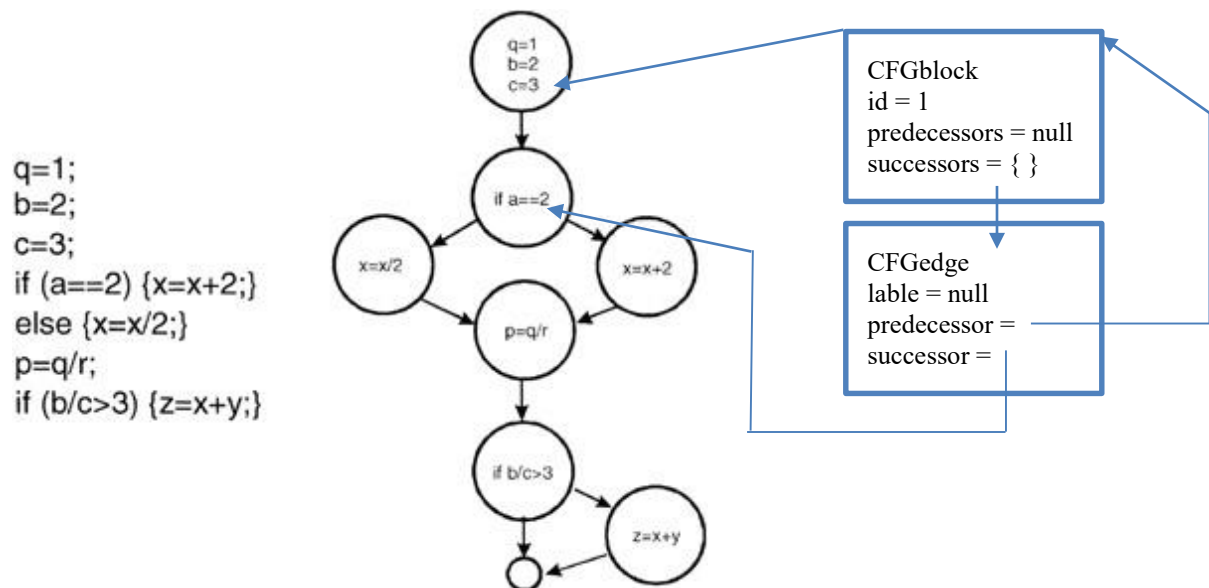


Figure 5.1.7 – CFGObjects example [200]

These blocks are recorded in two ways, through linked lists stored in each block but also a list recoding all nodes. This list of all nodes is used when processing nodes regardless of position.

After the generation of the CFG, several other properties must be computed. These properties form lists of objects, and the lists are altered independently as individual CFGproperties objects that store the GEN, KILL, IN, OUT, and USE values for all nodes in the CFG. The descriptions of these can be found below.

- GEN
 - All variables that are assigned or created in a block
- KILL
 - All variables that are assigned in the same variable name as GEN; also includes GEN
- IN
 - Any variables that enter the block and possible assignment nodes
- OUT
 - $GEN + (IN - KILL)$
- USE
 - All variables used in a block

GEN, KILL, IN, and OUT are all represented by CFGdfi object as each value must contain the variable name and ID of the node in the CFG.

Liveness Analysis

Liveness analysis is a practical implementation utilising the CFG and dataflow analysis for general code improvement. It is a simple technique that computes whether a value assigned to a variable is ever used, a very basic example of which is shown below.

```

1 : int i = 0;
2 : i = read();
3 : switch(i)
4 : ...

```

Figure 5.1.8 – Liveness analysis limitation example

Though the first line declares ‘i’, it also assigns a value of 0 to it, which is then immediately wiped via assigning the user input ‘read()’ on Line 2, meaning that the initial value will never be used. This would be highlighted by liveness analysis. Liveness analysis utilises the control flow graph and GEN, KILL, IN, OUT, and USE to accomplish the technique.

Unreachable Code / Dead Code

Unreachable code is self-explanatory: code that will never be run, regardless of the program inputs. Examples are given in Figure 5.1.9.

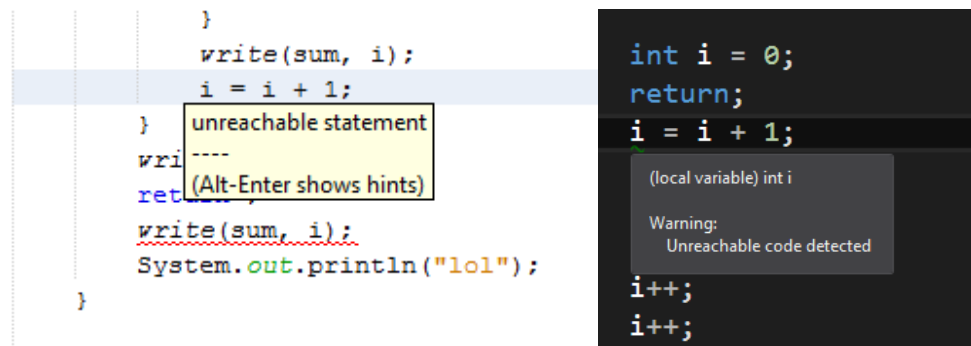


Figure 5.1.9 – Dead code limitation example

Figure 5.2.8 also demonstrates what could be a limitation in the implementation of feedback for unreachable code. As can be seen, in both NetBeans and Visual Studio there are multiple lines of code after the return statement, while only a single line is highlighted as an issue, which could be for one of two reasons. Firstly, it may be because the way in which they both compute the unreachable code is through automated compiler run-through, that is to say the code is simply compiled and generates an error when hitting that line, therefore only displaying

the first line. Alternatively, as discovered through development in LIQA, to display is a very difficult and pointless task, displaying more than one error when the user should pick up on it when a single line is highlighted. As any number of lines could be following the initial line of unreachable code, it is common sense that once a line is picked up, the user will correct all of the unreachable code. However, if this is not the case, each time the user corrects one line, the next one is highlighted. As can be seen from Figure 5.1.10, this does not affect multiple instances of unreachable code in a single application, and therefore the second of the two reasons is more likely.

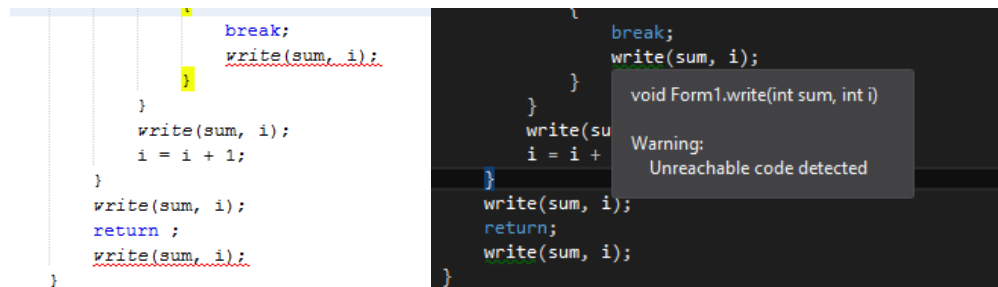


Figure 5.1.10 – Dead code multiple instances

Dead code analysis can take on more than the form of seeking code after a break or return. However, unlike the aforementioned simple implementation, LIQA's shall be a demonstration of another technique utilising the CFG. Dead code analysis examines all nodes in the graph and searches for those that have no predecessors as the algorithm used to compute the graph's successors and predecessors cannot link a return or break with the next line of code.

5.1.1.4 Static Metrics

Cyclomatic complexity

This metric is considered a standard measurement especially since it is a requirement of a variety of quality assurance standards including ISO25010 [201]. Cyclomatic complexity can be described as 'M = E - N + 2P' where:

- E is the number of edges

- N is the number of nodes (N is the block version of the CFG as described above)
- P is the number of exit nodes

[202]

An example of Cyclomatic complexity can be seen below:

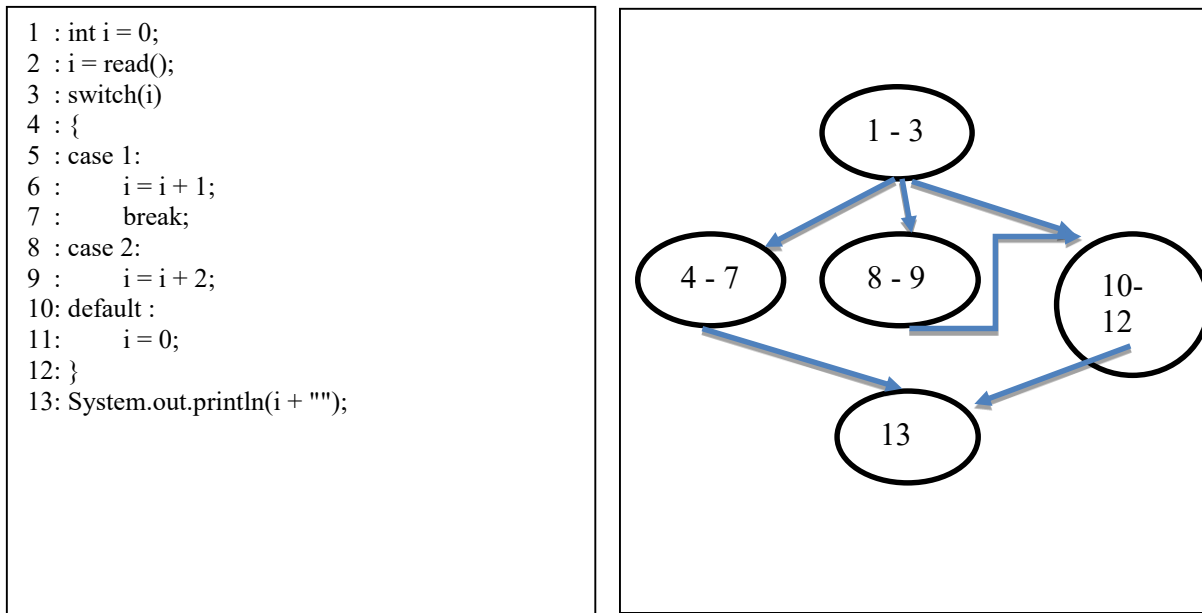


Figure 5.1.11 – Cyclomatic complexity example

The above has a cyclomatic complexity of 4.

$M = E - N + 2P$ where:

- $P = 1$
- $N = 4$: node representing line 13 is not counted as it is a continuation from the previous 3 nodes
- $E = 6$

Though this is the formal formula, there is a simple method of calculating the cyclomatic complexity in perspective, as described by VS [203]. The simpler method for the calculation is to

count the number of binary decisions and add one. Binary decisions are ‘for’, ‘while’, and ‘if’ statements; however, this approach does not consider switch statements in which you can just add the number of cases and default to the total conditions as a modifier, as any of the results should still be correct.

Logical Line Count

Logical line count is a very simple metric, and close to source code count. Source code count is a literal line count of all files regardless of content. Usually, source code line count also informs the user of blank lines. As this IR is abstract from source code, logical line count is the closest metric and overall more informative. Logical line count is a count of any lines that are functional, taking each statement as a single line. In the case of the GASTM, the logical line count can be implemented via a tree walk that counts statement nodes and containing nodes, i.e. functions and classes.

Halstead’s Complexity Metrics

Halstead’s complexity metrics are a means of quantifying the complexity of a piece of code, but, unlike McCabe’s cyclomatic complexity, which works based on binary decisions, Halstead’s metrics are based on operators and operands alone [204]. Though the metrics themselves are almost always the same equation, the operators and operands can differ from source to source [205] [206] [207] [208]. For that reason, a list of tokens has been provided below to outline the operators and operands being used within the implemented metric in LIQA.

- Operators
 - Reserved words
 - Type qualifiers
 - Type
 - Binary and unary operators
- Operands
 - Identifiers

○ Literals

An example of this being calculated can be seen in figure 5.1.12:

```

void sort( int a, int n ) {
int i,j,t;
if( n<2 ) return;
for( i=0 ; i<n-1; i++){
for( j=i+1; j<n ; j++){
if( a[i]>a[j]){
t=a[i];
a[i]=a[j];
a[j]=t;
}
}
}
}
}

```

	Total	Unique
Oppperators	N1 = 50	n1 = 17
Oppperands	N2 = 30	n2 = 7

Oppperators				Oppperands	
3	<	3	{	1	0
5	=	3	}	2	1
1	>	1	+	1	2
1	-	2	++	6	a
2	,	2	for	8	i
9	;	2	if	7	j
4	(1	int	3	n
4)	1	return	3	t
6	[

Figure 5.1.12 – Halsteads complexity example [209]

Several complexity metrics are used. N1 and n1 are operators and unique operators, respectively. N2 and n2 are operands and unique operands, respectively. From these, the various metrics are calculated:

Program Length	:	$N = N1 + N2$
Program Volume	:	$V = N * \log_2(n1 + n2)$
Program Level	:	$L = (2/n1)*(n2/N2)$
Program Difficulty	:	$D = (n1/2) * (N2/n2)$
Program Effort	:	$E = D * V$
Program Time	:	$T = E/18 \text{ seconds}$
Program Content	:	$I = (V / D)$

5.1.1.5 *Pattern Matching*

Pattern matching makes up a large part of automated techniques, especially around the area of standards and guidelines. Although they can vary quite extensively, essentially they identify patterns within code that require modification to adhere to that specific standard. For example, the .NET usability guide states that if a method has more than three parameters of the same type, these should be passed as an array.

The chosen techniques to implement are derived from NetBeans Hints for Java and .NET Managed Code Warnings for C#. Since these patterns pick up standard/guideline issues, the C# patterns can be applied to the Java language. The NetBeans Hints being implemented are:

- Assignment to method parameter
- Final class
- Final method
- Final method in final class
- Final private method
- System out / err

All of the descriptions can be found in the extended appendices [107].

The .NET Managed Code Warnings being implemented are:

- CA1025: Replace repetitive arguments with params array
- CA1708: Identifiers should differ by more than one case

These descriptions can also be found in the extended appendices.

5.1.2 *Dynamic Analysis*

5.1.2.1 *Dynamic Metrics (Profiler)*

A lot of the profiler techniques used by the tools are very low level, which is impractical to perform for a programming language-generic system. Some of the techniques can still be implemented within LIQA and will be done to demonstrate how profiling would work within this

framework. These implemented techniques will be a function call counter, inclusive and exclusive function execution time, as well as average, minimum, and maximum function execution time. This data was retrieved as an example of a type of dynamic analysis and is the basis for call graphs to be generated.

These dynamic metrics are retrieved from the sample program at runtime. This requires using the dynamic section of the framework, which includes inserting nodes into the IR to represent calls to an included class called MonitorDA. This class would have to be written for each programming language when incorporated into the framework. MonitorDA is a small class that has its own limitation in that it uses stack trace to acquire information about the running program, which means that only programming languages that support stack tracing can be dynamically analysed by the implemented skeleton framework.

An information retrieve call will be placed at the beginning and end of each method. This will record whether it is the start or end of a method, the method name, and the current time. This is all the information needed to deduce the previously mentioned dynamic metrics.

5.1.3 Development Discussions

This section has been included to enable discussion of the breakdown of the implementation after its completion. First of all, some metrics have been provided to assess the size and complexity of LIQA.

Total lines of code: 104,765

Total imports: 3,066

Total blank lines: 8,133

Total classes: 263

Total methods: 4,158

Average cyclomatic complexity: 3.72

It is important to consider that the Generic Abstract Syntax Tree Metamodel (GASTM) core components were not included as a library but were decompiled and included directly into the project because modification of specific classes was required and their dependencies had to be included. With this in mind, the development of the skeleton framework for Language-Independent Automated Quality Assurance (known as LIQA) was still extensive, although the metrics provided need to be read with consideration for the modified libraries.

The major components, with the exception of the GASTM, have been broken down into packages related to their purpose, including GUI, IR (with IR.Token), and LIQA. The GUI package (standing for graphical user interface) contains all of the JForms used for LIQA, which was developed from a single initial form into five separate forms to allow for simpler debugging. However, an official build of the framework would also require more than a single form as some techniques, such as variable remaining, require user input and selection. The IR package contained classes and a sub-package called Token, of which Token has a sub-class of Java. The Java packages contain classes that implement the tokenisation of the Java programming language. If other programming languages were implemented, these would be contained in their own package here. The class in the Token package contains constructs to direct tokenisation based on source code and programming language selected. The classes contained in the IR package are builders of two types: a single generic builder, to direct flow, and programming language-specific builders to take tokens and form the internal representation. The final package LIQA is essentially LIQA-dependent code, i.e. initialiser for the program as well as a project information store.

5.2 DISCUSSION OF THEORETICAL TECHNIQUES

The number of software quality assurance techniques collated in the progression of this research has been extensive. Although it would be impractical to implement all these techniques within the scope of this research, it is pertinent that the categories within the taxonomy that do not have techniques implemented within LIQA, be discussed.

5.2.1 Static Analysis

5.2.1.1 *Visualisation*

Visualisation is a large and important area within automated quality assurance. However, it can be acquired through many means to represent the results of other techniques. For example, the Dependency Finder in NetBeans visualises the links between classes including library calls, which is important because it allows users to quickly interpret a large number of links between complex code. Although the implementation of visualisation may be complex in its own nature, it works based on other data that is retrieved either through access with the source/IR or by means of other automated techniques. Concluding that visualisation, though important, is essentially an add-on on top of different techniques and is not essential to demonstrate would, however, be entirely possible to implement within this framework. Visualisation could be implemented in many ways. Two ways would be to extract the data required through other techniques and output them in a format that could be viewed by the user, e.g. SVG or HTML. Another way would be to have the visualisation built directly into LIQA; in this way, modification or animation could be used to improve the user's experience.

5.2.1.2 *Artefact Generation*

Although artefact generation is most commonly referred to as reporting, due to its implications, it could be considered an extremely important factor within software quality assurance. Artefact generation can not only inform the user as to the 'quality' of the program, but is most commonly used for submission for standard certification. The level of detail required for each standard and each artefact within the certification submission is large and will usually require many of the metrics and other techniques to be performed and then summarised. Therefore, though important, much like visualisation (discussed above), it is an additional form of representing data from other techniques and does not have a significant role in this research.

5.2.2 Dynamic Analysis

5.2.2.1 Unit Testing

A major form of dynamic analysis, unit testing, could be achieved within this framework in a variety of ways. Three such ways could work within the framework, though these are very extensive and therefore out of scope. One way of implementing unit testing could be to attach the xUnit framework, which would require source code, and therefore the conversion from IR to source would be required. Another form of unit testing that could be implemented could be similar to WinFPT where a test session of the program is run and all inputs captured. Then a test case is generated where the user can state the number of iterations and modify input values for that test. The third form of unit testing would be to segment the ‘unit’ within the IR and create a test environment for this singular piece of code. Depending on the nature of the test environment, this could be written directly in the IR format and outputted to the selected programming language or, if programming language-dependent features are required, each environment would have to be written in its own programming language.

5.3 TESTING

The framework’s success will be assessed by means of the testing of LIQA. Although this will take place in a classical manner, the testing will encompass the techniques alone and not be evaluating all of the functionality of LIQA, which has been included for ease of use. Testing only the techniques and not the implementation of the IR and GUI will mean that each test has a direct relation to the framework’s functionality, which will yield clear results. These results can then be used to infer if that techniques category, present in the taxonomy, is feasible within this framework.

5.3.1 Testing Plan

Each test will be included in its own directory with any relevant or equivalent files. Each test will be a single technique that will be identified within the test title, and will have a unique ID to refer to it. Some techniques will require multiple tests depending on complexity and

borders, e.g. private methods within a class can either be identified or not, requiring two tests. Each test will have a calculated result that will be arrived at by the tester, as well as a result from a quality assurance tool that utilises that technique. This may require a test program to be written in Java as well as an equivalent programming language for the secondary tool. The tool chosen will be based on the technique and also the similarity of programming language. The equivalent code will be code that embodies exactly the same semantics as the original application and that would generate the same GASTM tree. Some tests will not have a secondary tool for comparison; this is not due to the techniques not being used, but usually because the result is not accessible, e.g. Visual Studio implements Halstead's metrics to calculate the Microsoft maintainability metric [173], but the values for Halstead's metrics are not accessible. Additional notes will be included to instruct on how the test is run via LIQA and also whether anything is required to be taken into account when viewing the results. A list of techniques has been included below for reference:

Code Manipulation

1 Declaration and Names

Replacing a highly used variable with an inappropriate name to use a more appropriate identifier without modifying any other code.

2 Code Constructs

Automated modification of 'switch' to 'if'.

Optimisation

3 Unroll Do Loop

Optimise a section of code by replacing a loop with a set of equivalent statements.

Static Metrics

4 Cyclomatic Complexity 1

McCabe's cyclomatic complexity is used as a check for many standards and

is used to identify overly complex code that may be difficult to maintain.

Test 1 of 3.

5 Logical Line Count 1

Line count is a fundamental metric used to calculate other metrics, usually to establish an idea of the size of the program. Since LIQA is abstract from source code, the logical line count metric, which measures statement number, essentially predicts the lines of source code that would be present on output from the IR. Test 1 of 3.

6 Halsteads Complexity Metrics 1

Halstead's complexity metrics are a number of metrics measuring various aspects of a program: Length, Volume, Level, Difficulty, Effort, Time, and Intelligent Content, which are used by testers for other metrics, in reports for standards application and to ascertain problem areas of code. Test 1 of 3.

7 Cyclomatic Complexity 2

See Scenario 4. Test 2 of 3.

8 Logical Line Count 2

See Scenario 5. Test 2 of 3.

9 Halstead's Complexity Metrics 2

See Scenario 6. Test 2 of 3.

10 Cyclomatic Complexity 3

See Scenario 4. Test 3 of 3.

11 Logical Line Count 3

See Scenario 5. Test 3 of 3.

12 Halstead's Complexity Metrics 3

See Scenario 6. Test 1 of 3.

Data Flow Analysis

13 Liveness Analysis 1

Liveness analysis is very important as any liveness issues have the potential to cause program crashes; accordingly, these must be identified.

14 Liveness Analysis 2

See Scenario 13.

15 Unreachable Code 1

Unreachable code can cause major issues if the code that is unreachable is intended to be run; detecting code placed after break and return is important. However, there are other forms of unreachable code not covered by this technique, e.g. inaccessible if statements.

16 Unreachable Code 2

See Scenario 15.

Pattern Matching

17 Replacing repetitive arguments with params array 1

Having more than three of the same type of parameter is in violation of the .NET managed code guidelines and therefore needs to be identified.

18 Replacing repetitive arguments with params array 2

See Scenario 17.

19 Replacing repetitive arguments with params array 3

See Scenario 17.

20 Identifiers should differ by more than just case 1

Having variables' names differ only by one case is in violation of the .NET managed code guidelines and therefore needs to be identified.

21 Identifiers should differ by more than just case 2

See Scenario 20.

22 Assignment to method parameter 1

As Java only has parameters passed by value, not by reference assignment to them, is a violation of NetBeans code guidelines and needs to be identified.

23 Assignment to method parameter 2

See Scenario 22.

24 Final Class 1

Having a final class declared is against the NetBeans standard. The 'TestApp' class has been declared as final.

25 Final Class 2

Identical to Scenario 24 except that the final has been removed; no final class should be identified.

26 Final Method 1

Having a final method declared is against the NetBeans standard. The 'fact' method has been declared as final.

27 Final Method 2

Identical to scenario 26 except that the final has been removed; no final methods should be identified.

28 Final Method in Final Class 1

When a final class is declared, all methods within that class are final without declaration; adding the final keyword is redundant and a violation of the NetBeans code standard.

29 Final Method in Final Class 2

Identical to Scenario 28 except that the final has been removed; no final methods should be identified as class is not final.

30 Final Private Method 1

A method declared as private and final is redundant and therefore against NetBeans quality code checks. The fact method in this scenario has been declared as such.

31 Final Private Method 2

Identical to Scenario 30, except that the final has been removed; no final methods should be identified.

32 System err/out

Debugging lines left in code can reduce the speed of code execution, which can have a significant impact depending on the situation of the output. Leaving debugging outputs is another NetBeans check for code quality.

Dynamic Metrics

33 Dynamic Metrics

Dynamic metrics, or profiling, are used to identify problem areas in code, much like static metrics. As dynamic metrics involves the collection of data whilst the program is being executed, code has to be inserted into the program at specific points to retrieve the required data.

5.3.2 Testing Notes

Test ID 2: A value of 0.5 is assigned as a result of the output working correctly. However, the syntax was not as expected.

Test ID 3: minor syntax issue

Test ID 6: counting issues, still possible

Test ID 9: incorrect counting

Test ID 12: null pointer exception

Test ID 33: issues with time calculations, unknown cause

5.4 ANALYSIS OF RESULTS

The analysis of the test results will be completed objectively by examining overall results and then categorising the results into technique categories as defined by the taxonomy. Finally, each test that did not completely pass will be examined and a discussion of each of these will be included.

A percentage pass grade has been given via Equation 1 where ‘R’ represents the number of results in the test and ‘C’ represents the correct results in the test. This equation is essentially a reverse percentage.

$$1 - \frac{(R - C)}{R}$$

Figure 5.4.1 – Equation 1

Although this will give a value of the pass rate for each individual test, further consideration is required as some tests are performed multiple times for boundary testing and retesting more complex code with the same techniques. The number of additional tests can signify that categories may have a high pass percentage due to retests mirroring original tests; therefore, Equation 2 must be used as a comparison. These are known as grouped tests. Equation 2 was formed using the initial reverse percentage, shown in Equation 1. This needed to be

repeated for all sets using summation [210], and this then needed to be divided by the number of tests in the group represented by X.

$$\frac{\sum_{i=1}^x 1 - \frac{(R_i - C_i)}{R_i}}{x}$$

Figure 5.4.2 - Equation 2

‘R’ represents the number of results in the test and ‘C’ represents the correct results in the test, while ‘X’ represents the number of repeated tests that will give a single percentage for each technique that can be used for comparison against individual test results. Firstly, the overall results can be placed in three categories, pass (100%), partial pass (50% - 99%), and fail (0% - 49%), as shown in the pie charts shown in Figure 5.4.3.

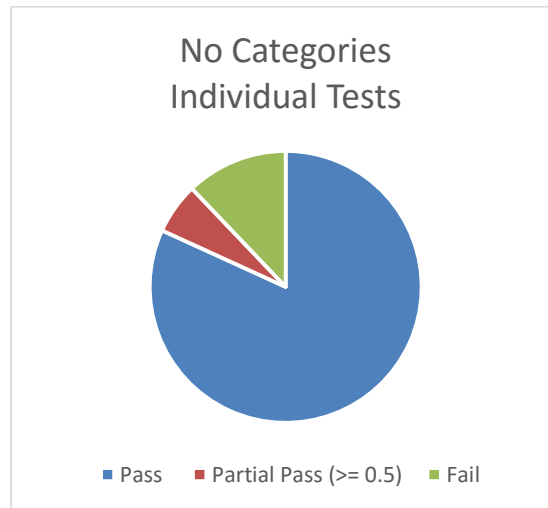


Figure 5.4.3 – Pie chart showing individual test results

Considering the results individually (Figure 5.4.3), it can be seen that LIQA techniques have a high pass rate at 27 out of 33. However, this result does not take into account multiple tests targeted at the same techniques. Figure 5.4.4 groups the tests using Equation 2 from Figure 5.4.2. As can be seen, the results are skewed, highlighting a similar failure rate at 11.8% over the previous 12.1% value. In addition, this graph shows the higher increased partial pass overtaking the number of failures and reducing the full pass rate. Overall, these results show an 85%+ pass and partial pass rate, which is exceptionally high and highlights the overall value of this framework. Before the analysis can end, a more detailed examination of categories and individual failures is required to uncover the flaws in this framework.

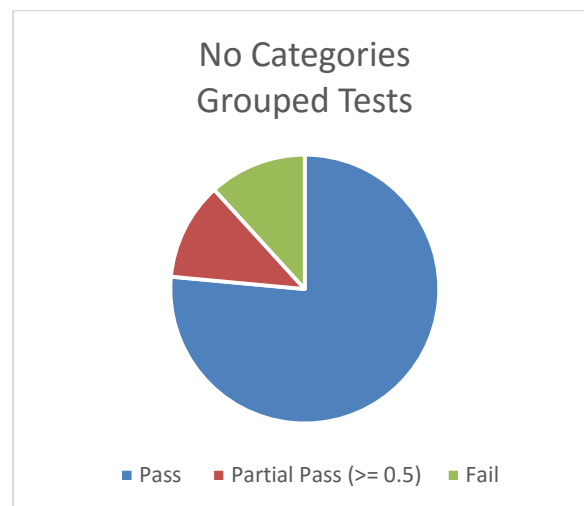


Figure 5.4.4 – Pie chart showing grouped test tests results

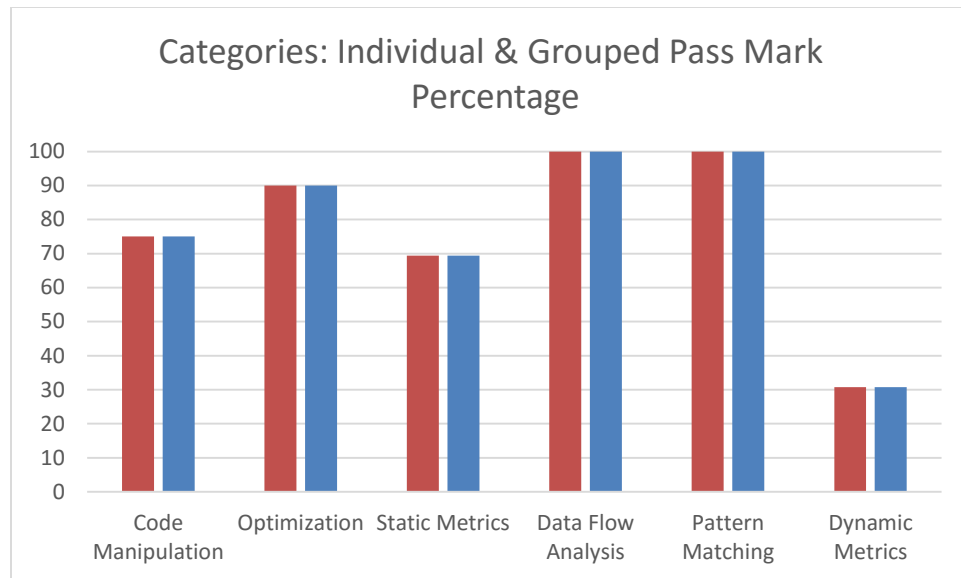


Figure 5.4.5 – Bar graph showing categorised test results

The Figure 5.4.5 shows the pass rate of all tests within their respective categories, as defined in the taxonomy. The coloured bars display both grouped and individual tests that are identified as the same. This is in contrast with the previous graphs and therefore proves that the categorised test presents all differences independent of each other. The levels of this graph also help to identify issues with regard to techniques; it can be seen that most issues are located in dynamic analysis and that all other categories have a high pass rate, although, considering the graph in Figure 5.4.6, a slight question arises as to the accuracy of identified issues. For example, optimisation has a 90% pass rate although all of these passes are only partial. Code manipulation is similar in this manner, having 50% pass and 50% partial pass.

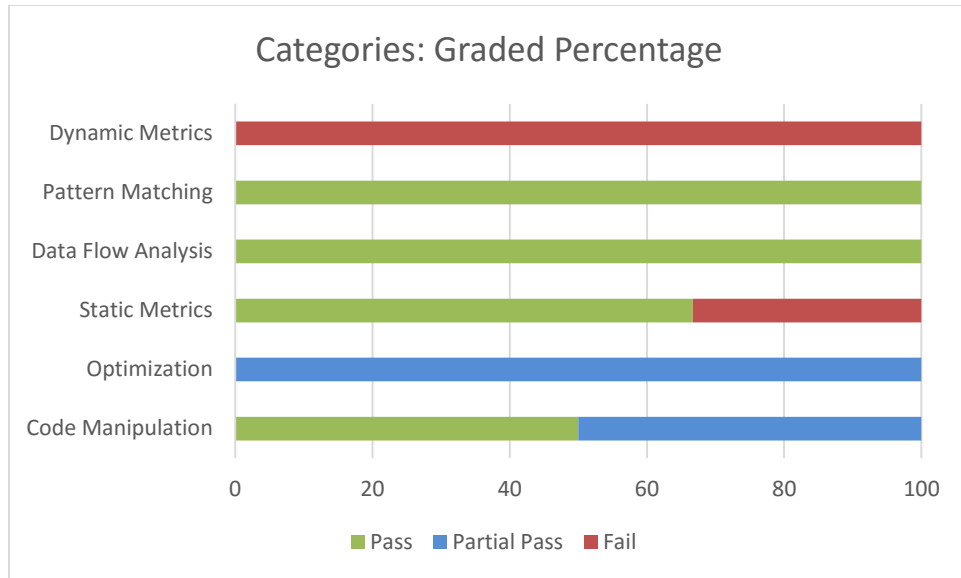


Figure 5.4.6 – Bar graph showing categorised pass results

Issues that have arisen in the categories are identified in this bar graph; listed in order of severity: Dynamic Analysis, Static Metrics, Optimisation, and Code manipulation. In order to accurately assess the framework, each failed or partially passed test must be reviewed individually to identify the extent to which the framework’s design, or its implementation or technique, are issues.

Looking in detail at the partial passes, specifically Tests 2 and 3, they contain syntax issues; however, they would otherwise be full passes. Regarding Test 2’s identified issue, the result of the technique would work; however, the issue related to the ‘IF - ELSE IF’ printed in an embedded format rather than in sequence is due to the GASTM representing ‘IF – ELSE IF’ as embedded trees. However, this could be counteracted in the Java output to create the sequence effect. Test 3 also had a minor syntax issue in which any code un-rolled from its loop was contained in a block statement, which arose from a decision made when the technique was implemented, as replacing the loop by a single block statement was easier to implement within the IR, although this could be modified to allow for multi-statement replacement.

Test 33 is the dynamic analysis technique and is identified as the category with the largest number of fails. Four of 13 tests passed, which were the method counters, while the failures were all related to measuring the execution time of methods. The idea for dynamic analysis, inserting prewritten methods to identify different running metrics, is sound, but the problem for the method timing issue seems to be in the prewritten code and does not reflect the validity of the technique. What is meant by this is that though the figures were incorrect, it is believed that the issue may have been related to the test or the MonitorDA timing calculations. The issue may arise from the methods execution time being too small, and this may have caused some issues. A re-test with delays written into the methods may be a better test for these techniques, an example of which is shown in Figure 5.4.7:

Method	Count	IN (MIN)	IN (MAX)	IN (AVG)	EX (MIN)	EX (MAX)	EX (AVG)
main	1	41	41	41	11	11	11
m1	1	19	19	19	4	4	4
m2	2	9	9	9	6	6	6
m3	4	2	3	2	2	3	2

	Count	Exclusiv e	Inclusive
Main	1	A	A+B+2C+4D
M1	1	B	B+C+2D
M2	2	C	C+D
M3	4	D	D

Figure 5.4.7 - Additional Profiling Test with original calculation

The code for which can be seen in figure 5.4.8:

```

public static void main(String[] args)
{
    m1();
    m2();
    m3();
    int i;
    for(i = 0; i <2500000000; i++){
    }
    public static void m1(){
        m2();
        m3();
        int i;
        for(i = 0; i <5000000000; i++){
        }
    }
    public static void m2(){
        m3();
        int i;
        for(i = 0; i <10000000000; i++){
        }
    }
    public static void m3(){
        int i;
        for(i = 0; i <20000000000; i++){
        }
    }
}

```

Figure 5.4.8 Additional profiling test code

The final three tests that failed, Tests 6, 9, and 12, were all implementing the same technique, Halstead's Metrics. The test was evaluated on the four base numbers that are retrieved from the source, which are then used in various formulas to assess different aspects of the program, such as length and complexity, etc. Tests 6 and 9 had only a counting issue, which essentially was due to either incorrect counting or some aspects of the program not included in counting, e.g. any expression within array retrieval (`arr[?]`), which is an implementation issue rather than an issue with the technique being used upon the IR. Test 12 was another implementation issue as a null pointer exception was caused within the counting class, identifying a problem with the LIQA implementation of Halstead's Metric.

When developing the tests, a significant limitation was taken into account in that the generation of the CFG would only work if, after each loop, at least one statement was included, e.g. `System.out.println`. This was due to the discoverer for the CFG. Though this is a significant limitation, it could be counteracted. In this case, the CFG discoverer was not modified due to the significant amount of time it would take to re-write which, considering that the CFG generated

properly with a non-significant statement placed after the loops, it was deemed acceptable for this level of testing.

5.5 FRAMEWORK CONCLUSION

This section of the research brings to a close the development and testing of the framework that has been implemented in the form of LIQA in order to determine its success in terms of the implementation as well as the theoretical framework. This is the programming paradigm-specific framework, in this case specific to the procedural paradigm, which is due to the internal representation used (GASTM). The GASTM has its limitations with it being a subset of programming languages. However, because it is a subset for most procedural programming languages, the support for many programming languages outweighs this limitation, especially since the additional part of a programming language can still be modelled using the ASTM, making the interoperability between programming languages possible. Although it would have been ideal to implement a parser for another programming language besides Java, there is enough evidence in the ASTM documentation to support the assumption that this is possible.

The techniques themselves, as seen by the results analysis, can be implemented using the GASTM as an IR for programming languages. Although there are some issues, these can be identified as general implementation faults rather than theoretical implementation issues. This is to say that the techniques can be implemented correctly in theory, but the actual implementation in LIQA has faults.

A factor to consider, however, is the appropriateness of dynamic analysis within this framework. Though a viable method of implementing some forms of dynamic analysis is inserting code to track various aspects of a running program, there are other ways of applying these techniques that could be considered better, e.g. tracking through a virtual machine. Though inserting code is not the most appropriate solution, many details of a running program can still be acquired in this way, for example coverage analysis to identify high-use areas of code, variable

monitoring for unit testing, etc. These aspects of a running program are valuable to testers and can still be implemented upon the IR. As for additional dynamic analysis, this must be implemented in another form. A solution to cover both the issue of not being able to implement all types of dynamic analysis and to reduce overhead of inserting code, is the option to create a virtual machine (VM) that is capable of compiling and running the IR, which would allow monitoring to be carried out at the VM level. This would also allow for the implementation of a debugger and variable watcher. Though a viable solution, introducing a VM creates its own issue in that each compiler can create different outputs depending on the programming language and the compiler version and creator. If a program is targeted at a specific programming language and compiler, performing dynamic analysis upon the LIQA-based compiled code could have discrepancies, whereas inserting code would not. Considering this path as an option, inserting code for dynamic analysis is a more appropriate approach and, if further dynamic analysis needs to be performed, this would have to be done externally from this framework.

As a result of this research, it can therefore be confidently concluded that a framework for automated quality assurance upon generic procedural programming languages is possible.

5.6 SUMMARY

This chapter aimed at and succeeded in specifying the automated quality assurance techniques, taken from the taxonomy, forward to implementation within LIQA the application designed around the framework. This not only covers Aim 2, Objectives 2 and 3 but is also one of the original contributions of this thesis. Following this chapter will be a theoretical examination of the expansion of the ideas that build the foundation of the framework and overlap of these principles to achieve a wider coverage in the space of programming languages and paradigms.

Chapter 6. Theoretical Discussion

This chapter is not only a future work area. The chapter takes the work done on the Procedural Language-Independent Automated Quality Assurance Framework and theorises expansion as well as suggesting future expansions. Such expansions include paradigm, grammar addition, direct expansion, etc.

6.1 PARADIGMS

6.1.1 Paradigm Discussion

Paradigms in programming are generally considered a ‘way of thinking’ or a way of approaching a problem/programming task [211]. There are many programming paradigms and combinations of programming paradigms, but only a few are considered the ‘main’ approaches to software development. These approaches are imperative/procedural (which can be considered the same, although, by definition, they are slightly different), Functional, Logical, and Object-Oriented [9]. There are several other programming paradigms that could be considered important but, from a construct assignment approach, these are the main ones upon which this research will focus; other paradigms such as ‘Event driven’, which could be considered main paradigms, utilise similar constructs. There is also no international standard definition for each programming paradigm, nor is there a clear divide amongst community definitions of some of the programming paradigms. Each of these programming paradigms needs to be considered when a generic framework is discussed because each programming paradigm has different constructs to represent and assist in the implementation of that approach. In a world where a programming language was purely based on a single programming paradigm, programming languages based in different paradigms would be disparate; however, this is not the case. Individual paradigms are important in order to assess how generic this framework can be. In addition, there are programming languages that breach different paradigms and so the constructs included in these programming languages must be included in the framework to allow full assessment of those programming languages. Each paradigm’s constructs must be absorbed into the internal representation only if that paradigm meets a certain set of criteria.

The criterion for the feasibility for programming paradigm compatibility is that the paradigm requires quality assurance in at least one of the same ways also required by a previous paradigm, i.e. both paradigms must have some quality assurance techniques in common, or must have programming languages that include constructs of both paradigms; e.g., Java can be used as a procedural language or as an object-oriented language.

A final note before analysis is that a paradigm could be implemented in a programming language that does not support the ‘pure’ constructs because a paradigm is a way of approaching a problem. Programming languages that are designed around other paradigms can be adopted and a program’s implementation could be considered of a paradigm that the programming language does not completely support.

6.1.2 Paradigm Analysis

In this section, several programming paradigms will be broken down to allow the theorised inclusion of their constructs within the internal representation of this framework. An initial scope must be included due to the number of paradigms available, and a distinct difference between each paradigm must be identified to assess its impact. Though paradigms are considered as a way of approaching a problem, the implementation of these paradigms could be considered based in one of two areas, imperative or declarative. The Imperative Paradigm, by definition, follows its path based on computer architecture in which instruction sets are used to instruct the moving of data from memory for computation to produce results. The initial programming language for the Imperative Paradigm is assembly [212]. Higher-level imperative programming languages permit the use of more than the simple constructs provided in assembly, but were still dictated by the imperative paradigm. The Declarative Paradigm, in contrast to the imperative paradigm, is not about writing how a program should achieve something, but rather what the program will achieve [213]. This is sometimes referred to as declarative semantics [214] where a statement’s meaning is determined independently of how it will be implemented. Figure 6.1.1 is a categorisation of programming paradigms that clearly shows declarative and imperative being distinct areas, but also identifies the other paradigms that will be examined in detail later. These

are, from left to right in the diagram, logical (pink), functional (yellow), and purple (procedural). Another key paradigm, though contained within the purple section, is object-oriented and that, as it is a popular paradigm, will also be analysed.

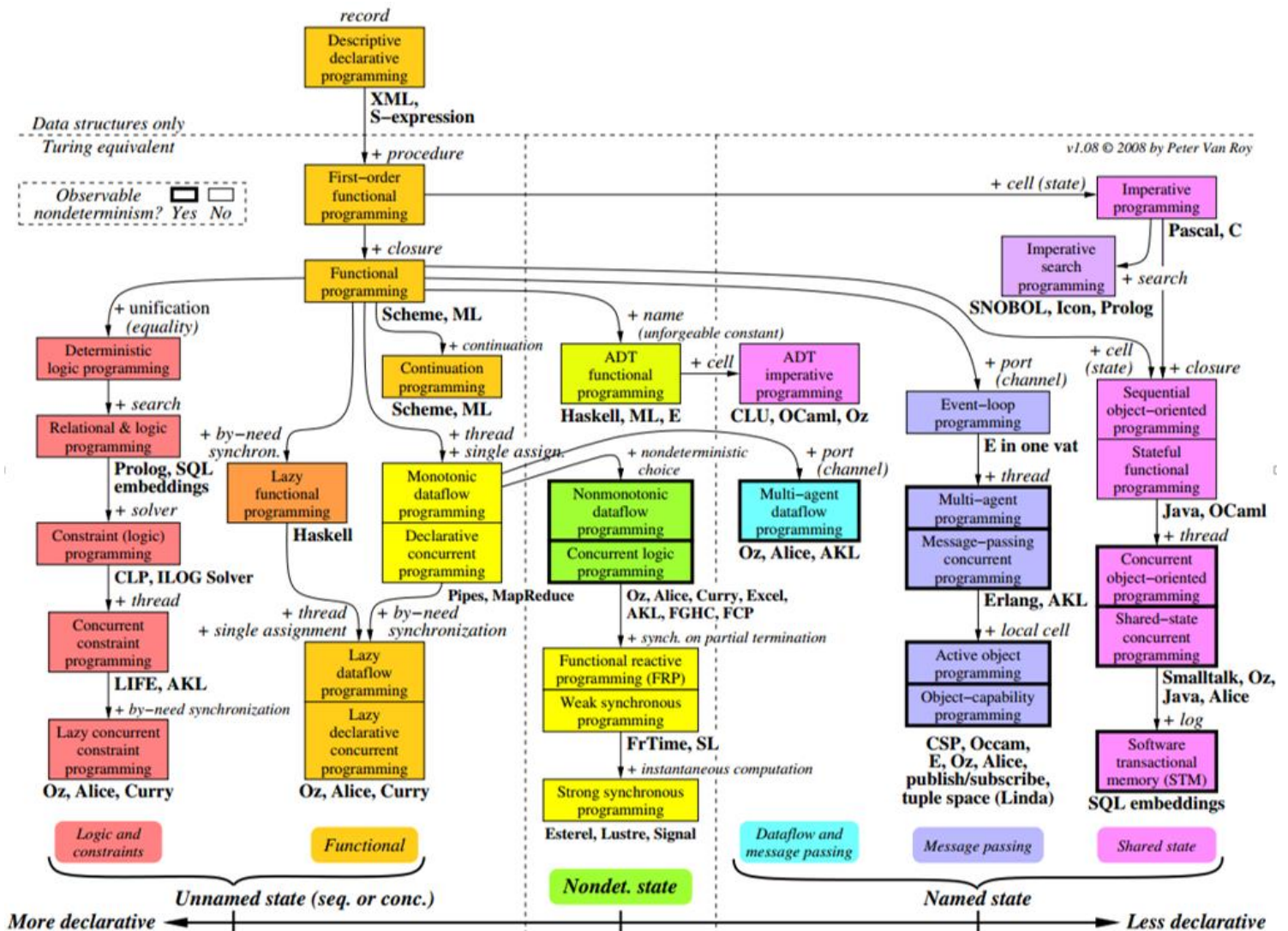


Figure 6.1.1 – Paradigm Breakdown [215]

While Van Roy discusses the different concepts used to build up a paradigm [215], this research is interested in the constructs that are not included in a paradigm that has already been

theorised to be feasible within the internal representation of the framework. Essentially, the constructs required for adding a programming paradigm need to be identified so that they can be introduced to the internal representation. This breakdown of programming paradigms shows how extensive paradigm development has been, displaying only a small number of key paradigms to try to represent the entire domain.

6.1.2.1 *Procedural*

It is first essential to outline the procedural programming paradigm. However, within this research, several paradigms are being assumed under this single title: Imperative, Structured, and Procedural.

The Structured Paradigm arose from the further development of programming languages when a level of abstraction developed between a high-level programming language and low-level imperative programming languages. This allowed the adoption of more constructs such as blocks and subroutines, which formed a more considered approach to programming where sequence, sections, and iteration took the forefront and were considered the building blocks for all programs built using the structured paradigm [216].

The Procedural Paradigm focuses on modularity. It could be considered that a common practice today is the utilisation of object-oriented languages and constructs to develop programs written in a procedural manner [217]. The procedural paradigm concepts lie in re-usability, where modules are created to collate code into similar groups, e.g. a database handle module may contain all the code required to access the database in a generic format. But the ideals behind the procedural paradigm are verb-fixated, meaning that the idea behind procedural programming is action-oriented, where a problem is broken down into functions and the purpose of data is to help the function complete its task [218]. This is why procedural programming is considered programming by side effect, where the return value of a function may not be as

important as the data manipulated by the function [219]. This definition is being phrased in a way that will help differentiate the procedural paradigms from other paradigms.

Key constructs

The key constructs have been separated into three sections representing the development of the paradigm through its ‘predecessor’ paradigms. These are very loose separations, and each could be argued as being present in another, which is why these constructs are being bundled together as a single paradigm.

- Variables
 - Assignments
 - Repetition
 - Expressions
 - Conditional branching
-
- Subroutines
 - Block
 - Return
 - Break
-
- Procedures/methods
 - Function
 - Record/Class
 - Modules

Initial Paradigm

The procedural paradigm was chosen as the initial paradigm for several reasons. One such reason relates to the domain at which this research was aimed, which is scientific software. Procedural programming languages are some of the oldest programming languages and though they have their drawbacks, they are used in all industries, especially the scientific domain. Another reason that the procedural paradigm was the initial focus for this framework was the large number of known issues regarding this paradigm. Due to the inherent nature of this paradigm, there are many user-generated issues, optimisation opportunities, and standards to take into account, and therefore the procedural paradigm provided an extensive quality assurance technique base to start with.

6.1.2.2 *Object-Oriented Paradigm*

The object-oriented paradigm is considered one of the most recent paradigms [220]. It is a paradigm that lies within the imperative paradigm, but its ideals are based more in modelling the real world via utilisation of instructions rather than instructions used to complete a task [221]. The way the object-oriented paradigm works can be considered a data-first approach. This is in contrast with functional (a function-first) and procedural (instruction-first) approaches, where data is represented as objects in which objects are modelled on real-world entities [222]; e.g. an instance of a class ‘Car’, which stores properties, methods, and functions related to the description and operation of a car, would be an object of type ‘Car’. Object-oriented thinking is completely different from the procedural approach, but it does have the same base paradigm of imperative. This is because the object-oriented paradigm is a concept paradigm where there have been a few specialised constructs made to help the development of object-oriented programs while, in reality, any semi-high-level procedural language can be implemented in an object-oriented way. For this reason, this paradigm receives this special mention and is at this time considered fully compatible with the framework to the same extent as the procedural paradigm.

6.1.2.3 *Functional*

The functional paradigm is an interesting area, particularly at present. There are several reasons for this interest, and these will be mentioned after an initial discussion of the paradigm's ideals.

The basic principle behind functional programming is to emulate and incorporate mathematics using data constructs originally implemented in imperative programming, e.g. lists. Basic values are built up using functions in an expressive format in which the program is not a list of instructions but an evaluation of expressions [223]. The major theme that runs through the functional paradigm is that a function does not cause side effects, as it is based on mathematic functions, i.e. a function's purpose is its return value [224]. This theme is more prominent when it is considered that all values in a functional programming language are immutable, that is to say, they cannot be modified [225].

Skipping the history lesson, functional programming is an important paradigm as it mingles more closely with popular paradigms such as object-oriented, of which the .NET platform is an example [213]. The .NET platform uses the Common Language Runtime (CLR) as a base for all of its supported programming languages, allowing for libraries to be built for multiple programming languages. One of these is F#, which originally was functional but is considered of multiple paradigms due to its adoption of objects. Even Java in version 8 has direct links with the functional paradigm by including lambda expressions.

It is a significant step having declarative programming language constructs available in predominantly imperative programming languages, and vice versa [226], and there are differing opinions as to whether this is an effective method of developing programming languages [227] [228]. Bearing in mind that these constructs have been implemented, whether it be to accommodate new and different users or just to expand the programming languages, is irrelevant, but it is nevertheless a considerable issue with regards to this framework. Until this point, all constructs have been imperative; contradictory to this, the declarative statements do not

necessarily dictate computational instruction but describe what is required. As a result, this may lead to difficulty including them into the internal representation.

Constructs

In order to assess which constructs are important for functional programming languages, several programming languages that support the functional paradigm can be considered. The four programming languages used as a basis for construct analysis are: Scheme, as it is a purely functional language; F#, as a functional programming language that adopts other paradigm constructs; Java and C#, to represent other programming paradigms that have, with varying degrees of success, adopted the functional constructs to incorporate the functional approach.

There are four main items that must be included to allow programmers to use the programming language in a basic functional manner. The term ‘items’ has been used because three are constructs and one is basic functional support for manipulating data. The three constructs are lambda functions, method references, and lists, with the support being included for basic functional operations on lists. All of these are supported in Java, C#, F#, and Scheme in different forms.

First to be considered is the basic data construct required for functional operation: lists. There are the usual primitive types to be found in functional programming languages, but the initial functional programming languages contain this data in their initial composite data type, lists [229]. As Scheme is a functional programming language, lists are supported by default and are a main construct [230]. This is similar in F# [231] but, contrasting this in Java and C#, there are various forms of list where, for example, the `List<T>` [232] in Java is not as suitable as the `Iterable<T>` [233] for functional programming. This is because some of the functionality that the functional paradigm requires is not natively present in the `List<T>` collection, e.g. Scheme `cdr`, `cons`, and `car` [234]. Though more popularly used constructs such as maps and dictionaries are

now used in both functional and procedural paradigms, it is important that the list construct is supported due to its heavy links with the functional paradigm.

Functional operation must then be included within the internal representation. Obviously in F# and Scheme, these are supported as in-built functions [235] [236] [237] [238], while in Java and C#, they had to be included under the functional libraries. The three main functions that are native to the functional paradigm are ‘map’, ‘reduce’, and ‘filter’, which are included in Java under the stream construct as the methods ‘map’, ‘reduce’, and ‘filter’, respectively [239]. This seems relatively simple considering that under C# the methods are found under the Enumerable type with the following names: ‘select’, ‘aggregate’, and ‘where’ [240] [241]. Though not important for the purposes of this research, it is an interesting point when considering a programming language that may want to adopt a functional approach.

Lambda functions are essentially anonymous functions in programming languages like Scheme and F#. As the functional paradigm emulates lambda calculus, and considering it is a key component to what makes functional language so expressive, it would seem controversial not to include this construct. Scheme represents this construct with the keyword ‘lambda’ [242]. F#, with similar syntax to Scheme, uses the keyword ‘fun’ as a replacement [243], while C# and Java replace the keyword approach with a symbol identifier placed between the parameter definition and expression itself, C#’s being ‘=>’ [244] and Java using ‘->’ [245].

Like lambda expressions, the identification of functions as first class is an important factor in defining functional compatibility in a programming language. This can also be referred to as method reference passing, but all it means is that a function can be passed in the same way that a parameter is passed. Functions are treated like data, which is how this works in Scheme [246] and, as such, has been adopted into F# in the same manner [247]. As this is not the case in imperative-based programming languages, an adoption in a different manner occurred in which, mainly due to strong typing, a type had to be generated so that these functions could be passed and so that the receiving method or function does not throw errors. In Java, the type is based in

'java.util.function' [248] and has various sub-classes for particular function instances, e.g. 'predicate', which is a Boolean function of one argument [249]. C# uses a similar system where the passed value is of type 'Delegate' or a sub-class of 'Delegate' [250]. As well as these types, a way of identifying the function that was passed in had to be created, which was achieved in Java by using the '::' operator to separate the class, object, or instance and the method name [251]. This was not done in C# as the identifier was sufficient.

It is important to note that although these are not all the constructs used within functional programming, they are key constructs that are needed to adopt a functional approach to tasks. As with the imperative constructs, only these key ones will be theorised within the internal representation and, if successful, this paradigm will be classed as supported. However, only further evaluation of the paradigms' constructs and individual programming language constructs would make an implementation of this framework fully generic.

Quality Assurance

Functional constructs are appearing in popular programming languages that can be, without these functional constructs, adopted into the framework. This is an important statement because non-adoption of these constructs would severely limit the impact of the framework. However, it would be key to point out here that this will not be an exhaustive study of quality assurance within the scope of the functional paradigm. This is to establish the feasibility of quality assurance within functional programming language as well as links with more generic quality assurance techniques that span multiple programming paradigms.

When considering the quality assurance of functional programming languages, certain key indicators can be identified, some of which are based in functional constructs in other programming languages, while other indicators are found in functional-based language. It is important to point out that quality assurance techniques could be considered useful for already

supported programming languages with functional constructs, while other techniques are new and based entirely in functional constructs.

Two of the most interesting techniques that are being highlighted here are in the multi-paradigm programming languages, Java and F#. In F#, there are performance analysis tools including several other common dynamic analysis techniques such as breakpoints, etc., which were shown in the Visual Studio analysis in a previous chapter. This is interesting for two reasons: functional programming is not usually considered in a performance-critical environment, and also these performance tools are almost the same as those used for C# analysis. Java 8u25 and NetBeans 8.0.1 [252] have included several ‘hints’ to modify procedural code into a functional format utilising functional constructs [253]. This is interesting as, although these tips can be turned off, NetBeans, by default, is promoting the use of the functional features over procedural code.

Something of note but not as interesting as cross-paradigm quality assurance techniques is the implementation of specific functional quality assurance tools. Haskell [254] has several of these including HLint, designed for increasing the readability of programs [255]. There are also QuickCheck [256], which is essentially a unit tester, and style-scanner [257], which is a formatting tool. There are also examples of quality assurance tools for Scala [258] that include Wartremover [259], similar to HLint, and Scalastyle [260], similar to style-scanner.

Though these tools indicate that quality assurance exists in the world of functional programming, they do not show two major points, i.e. what quality assurance techniques are being implemented and how these relate to the other programming paradigms. Literature can identify some key areas of functional program analysis, and therefore relationships can be established between quality assurance techniques in different programming paradigms.

One technique that is predominant in the identification of issues within code and assessing code with various constraints is *metrics*. The absence of metrics, be they similar to the procedural metrics or different, would be considered a surprise; however, this is not the case. There are several publications identifying different metrics that are used on functional programming languages [261] [262]. One paper discusses a tool-set based on a previous procedural tool-set stating that some of the metrics are applicable [262], demonstrating a direct link to quality assurance techniques used in different programming paradigms. The second paper discusses the possibility of metrics that would be useful for functional programming and ways in which they could be implemented [261].

Another technique that lies in the functional paradigm and in other paradigms is clone identification. Used to minimise code size and maximise code re-use, clone identification is a common technique discussed in literature. This implementation of clone detection in a functional paradigm uses an intermediate representation called a Function Control Tree [263], which may relate to the internal representation of this research.

A final quality assurance technique that shows some similarity between the imperative and functional programming paradigm quality assurance techniques, is refactoring. Refactoring is essentially a way of making code more readable, succinct, and easier to maintain [264]. This has been used in multiple paradigms and is an important form of quality assurance for the functional paradigm due to its expressive nature [265].

Though limited, these three techniques demonstrate a link between the quality assurance of functional programming languages and programming languages that are based in the imperative paradigm [266]. That, and considering the inclusion of functional constructs within imperative-based languages, further supports the need for including these constructs within the internal representation of the framework.

Adoption of constructs

The adoption of the previously mentioned functional constructs will be accomplished using the Abstract Syntax Tree Metamodel in the form of a description of the new classes and their location with regards to inheritance. Their use will also be described within an instance of an application in order to fully understand how these constructs are to be adopted.

Firstly, in Lists, there is an option to identify different types of list, as in .NET there are several types and therefore identifying a functionally specific list is advised. *'FuncListType'* will be added as a subclass of *'CollectionType'*. *'FuncListType'* will be a generic list for programming languages that support functional lists.

Method references can be implemented without adding any new objects to the abstract syntax tree meta-model. When defining parameters using the *'FormalParameterDefinition'* object, this object has *'Name'* and *'Type'* properties, where *'Type'* can be a *'FunctionType'* and *'Name'* could be the identifier. When passing the function, there is *'TypeQualifiedIdentifierReference'*, which has *'IdentifierReference'*, which can be the function name, and *'TypeReference'*, which can be an *'UnnamedTypeReference'*, which has *'Type'*, which can be *'FunctionType'*. Declaring first class functions could not be simpler, as the *'FunctionDefinition'* has a property *'FunctionMemberAttribute'*, which has a property *'isThisConst'*, a Boolean depicting a constant function that is synonymous to a first class function.

When considering the way in which a functional program is built up, though classes exist, they do not have to be used. The GASTM model does not support this directly, as the progress from file to aggregate type is fairly intuitive, where an aggregate type can be one of the following sub-classes: *StructureType*, *UnionType*, *ClassType*, or *AnnotationType*, none of which describe a container-less list of functions and constants. A simple solution to this would be

to add a new subclass to AggregateType that should have a name like ‘Containerless’ or ‘FirstClass’.

The last construct to include is the lambda expression, which can be described in the format of the abstract syntax tree as:

Expression => LambdaExpression

LambdaExpression ->

➔ Expression : Body

➔ Expression: ActualParams*

where LambdaExpression is a sub-class of Expression, contains an Expression type for its body, and contains a list of parameters as well as an optional assignment reference, to allow for imperative implementations of lambda that are strongly typed. Type information with regards to the return could be stored; however, this information is usually inferred with functional language and therefore has been left out, but could be stored as ‘*Type : ReturnType ?*’.

When considering the applicability of this paradigm within the scope of this research, only the constructs and the quality assurance techniques have to be considered; however, it must be stated how well these can be incorporated into the framework. Based on the observations made above, specifically those made when looking into the adoption of functional constructs within current object-oriented and procedural programming language, it has been seen that the paradigm in its entirety (both constructs and quality assurance techniques) can be adopted by the framework.

6.1.2.4 Logical

Like the functional paradigm, the logical paradigm is considered part of the overarching declarative paradigm [215]. Two programming languages will be considered, as well as the overall paradigm, when identifying constructs, feasibility, and whether the paradigm requires

inclusion within this research. The two programming languages being evaluated are Datalog [267], which is considered a purely logical language, and Prolog, which is considered one of the original logical programming languages. Prolog through its development has taken other paradigms into account including imperative constructs, e.g. statements [268], and additionally, modifications to the Prolog language have yielded object-oriented constructs [269].

Although the logical paradigm, at this point, appears to have the same ties to the imperative paradigm as the functional paradigm, this is not the case. Though there have been implementations of logical programming languages within imperative language, e.g. Prolog is accessible within Java, C#, C++, etc. [270] [271], these are implementations of the programming language via library and are made accessible via queries that are written in string literals in the host language [270]. However, there are some bi-directional implementations of Prolog and other programming languages such as Java [272] [273], though these are limited. This is different from the functional paradigm, as imperative languages have implemented functional constructs directly into the programming language, although there have been proposals to amalgamate functional and logical programming languages [274].

The logical paradigm is considered to be, in both implementation and thought process, drastically different from the other paradigms discussed [275]. The basis of this paradigm is to create facts with relations, and then the aim is to query the data with a problem, where the user is then provided with the answer. Because of this, the logical paradigm can be described as goal-oriented due to the focus being on what needs to be achieved and not on describing how to accomplish it [276].

The main constructs for this paradigm are facts, terms, and rules. In both Datalog and Prolog, facts are tables populated with terms where these are variables that are stored as literal constants. Rules are used to describe relationships amongst facts [277] [278]. Unlike functional and imperative paradigms, these constructs are completely different and the way computation is handled, where queries are performed after facts and relationships have been outlined, highlights

this difference. Prolog has been added to other programming languages via libraries, e.g. Java [270], C#, C++, VB, Python, etc. [271], and in a bi-directional format, e.g. Java [272] [273]. Similarly, Prolog users have tried to incorporate other paradigms, e.g. with types in Visual Prolog [279] and objects with logtalk [269], although these are very much unidirectional extensions as imperative programming languages and do not incorporate logical constructs.

Quality Assurance

As with any programming paradigm, some quality assurance techniques have been developed. At this point, based on the interaction with the other paradigms, a strong overlap needs to be found if the paradigm is going to be considered for adoption into this framework.

Static analysis of logical programs, specifically Prolog, as this is considered the most popular logical language, is present and widely used for a number of tasks. Type analysis is performed to compute the correctness of a program as Prolog is not a strongly typed programming language and inferred type can create compilation errors [280]. Another form of static analysis used is optimisation, but this usually occurs on post-compiled code to increase performance, which would be technically imperative code [281]. Other forms of static analysis, such as mode analysis [282] and termination checking [283], are logical programming language-specific.

Dynamic analysis contains one particular technique that is recognisable in other paradigms, which is the profiler. However, the metrics recorded by the profiler are not organised in a familiar manner, as execution time is still recorded as well as call counters. However, these are based on predicates rather than on functions [284].

Adoption of constructs

The adoption of the logical paradigm, and therefore logical programming language, will be considered out of scope for this project for several reasons.

- The constructs are too disparate, which would make them very difficult to include in the Abstract Syntax Tree Metamodel.
- The purpose of logical programs is different, as the logical paradigm is used in completely different domains that have little to no overlap with imperative or functional paradigms.
- Inter-paradigm inclusion is one way. There is work that has been done to incorporate inter-paradigm communication, but it is driven by the logical paradigm towards other paradigms, with other paradigms adopting the logical paradigm.
- Quality assurance techniques are desperate because the logical paradigm is so dissimilar that quality can be defined in a completely different way, with its own priorities.

6.1.2.5 Other Paradigms

In this section, several points will be considered and discussed concerning the ideals of paradigms and specific paradigms that are interesting or require highlighting as they have an impact on the framework in terms of usability, adoption, or exception. This includes multi-paradigm programming languages and how they will be accommodated within the framework, domain-specific, and other paradigms, as well as a few other specific paradigms and how they affect not just the framework but the area of automated quality assurance in general.

First, it is important to discuss multi-paradigm programming languages as this covers most programming languages available and in use today [285]. Java, which has been the programming language used for testing the skeleton framework named LIQA (Language Independent Quality Assurance), is considered multi-paradigm. Though it was initially intended to represent the ideals of an object-oriented language [286], it can be used for procedural coding

as well as for adopting functional constructs, making it span 3 of the 4 base paradigms discussed above. Programming languages like these are common and adopt enough constructs to enable each paradigm to adapt to scenario and programmer preference [287]. The way in which these multi-paradigm programming languages interact with the framework is essentially the same as normal, so long as the constructs for that programming language have been implemented in an abstract syntax tree meta-model format. For example, though Java has been used to test LIQA, not all of the constructs present in Java have been adopted into the abstract syntax tree meta-model. It is, however, feasible to do so, as these constructs lie in the three core paradigms that this framework could theoretically support, i.e. object-oriented, imperative, and functional. This does leave a small limitation, which is that if a programming language has constructs based in the logical paradigm, only programs that do not contain these constructs could be adopted into the framework. It is advisable that these languages not be included as supported programming languages anyway, because they would not be supported completely.

Another area that requires discussion is domain-specific or ‘more focused’ paradigms, that is to say the other paradigms, e.g. Symbolic [288], Event-Driven [211], Flow-Driven [289], Aspect-Oriented [290], etc. These paradigms have a multitude of uses and areas that result in them being considered the best form of programming, e.g. Symbolic programming was created to represent ‘learning’ in artificial intelligence [288], and Event-Driven programming is where a program’s flow is dictated by user interaction, which, in many ways, describes most office software. Though these paradigms have been created for specific purposes, the programming languages and programming constructs are derived from the core programming paradigms, i.e. imperative, object-oriented, functional, and logical; for example, Symbolic programming can be accomplished through LISP or Prolog, which lie in the functional and logical paradigms, respectively. Aspect-Oriented programming is achieved through the use of object-oriented and imperative programming languages [290]. These paradigms, therefore, do not need to be considered for the internal representation of this framework, as the constructs are the same as the core programming paradigms, and these are simply considered ways of approaching a problem or task.

A final consideration for the framework is a few specific paradigms that require an additional discussion to cover the field comprehensively, and these are *esoteric programming languages* and *constructive programming languages*. The esoteric paradigm is considered either an art or a joke [291], and is not intended for use in any manner where quality assurance would be applicable. Esoteric programming languages are usually very small and difficult to read (because they have been designed that way) and/or replace common keywords with something else e.g. ‘ArnoldC’ is an esoteric programming language in which keywords are quotes from different Arnold Schwarzenegger movies [292]. It is obvious at this point that the esoteric paradigm will not be considered for adoption within this framework. The other paradigm being discussed here is the constructive paradigm, which, like all paradigms, does not have an agreed definition. In fact, two completely different definitions can be found [293] [294]. As the visual definition for the constructive paradigm is another approach that utilises some of the core paradigms, it is not of particular interest, although the Scalor language [295], which is based on the other definition, is of interest. From the information given, the constructive paradigm, and therefore Scalor [296], allow the user to develop their own syntax and essentially build their own programming language [297] whilst including constructs from other paradigms [296]. This is interesting because the platform boasts no need for quality assurance as it is based on pure maths and logic. Although the platform may in fact require no quality assurance to ensure safe, reliable code, maintainability and user-based quality is at question as logic and maths cannot prevent a user from writing an entire program in an unmaintainable manner. The constructive paradigm, because of its ‘user-dependent’ nature, allows easier and probably high-quality programming based on safety and performance. Bringing this back to the framework in this research, it is clear due to the closed platform that Scalor is, with internal quality assessed by the platform it would be unnecessary to include this paradigm in the framework. The constructs used by this constructive paradigm again mimic the core paradigms and therefore theoretically could be included in the framework if logical constructs were discarded.

6.1.3 Generic Quality Assurance

This section has been included to discuss the broad idea of generic quality assurance and how the paradigms and quality assurance techniques interact, assessing possible issues and reasoning behind them being included or excluded, and how this may be affected by factors in each field.

Each paradigm has its place as a way to approach a problem, and each paradigm has evolved over time to cover many bases of development; this results in overlap and returns choice back to the developers. It is because of this that this research is so interesting; the overlapping areas provide a focal point to underpin the whole premise of interoperability between programming languages and thus permit crossover in specialised fields such as automated quality assurance. The breakdown of each paradigm, presented in chapter 5, shows that procedural and object-oriented paradigms have similar constructs, making them easy to view them as a single entity in this research. This is not a surprise based on the object-oriented paradigm being essentially developed from the procedural paradigm. When we compare these two to another paradigm, e.g. logical, which was developed in isolation and the constructs of which do not in any way match with procedural or object-oriented paradigms, there starts to be difficulties in identifying if, how, or why these paradigms should be combined. This is, as we have discussed previously, why the logical paradigm was deemed inappropriate to include in this framework; having no similar quality assurance techniques was the proverbial nail in the coffin for logical programming in the scope of generic automated quality assurance. The final paradigm to consider was functional, which interestingly has reasoning for both inclusion and exclusion when it comes to this framework. Functional programming languages are very different in their approach to solving problems, allowing the programmer to be more declarative and less explicit when writing code. That, combined with the fundamental difference of state vs no state, means that these paradigms look very different. If we consider quality assurance techniques at this point we can see some, but not much, overlap in the techniques implemented for each paradigm. Very basic techniques such as metrics in both dynamic and static senses are available for both paradigms; other techniques include unit testers and formatting tools that branch the gap between functional and procedural. However, each paradigm still holds its specialised quality assurance

techniques, such as refactoring for functional and various forms of code standard pattern matching used in the procedural paradigm. As can be seen, there is both argument for and against the functional paradigm being included with the framework.

A major consideration for any paradigm is the current state of play in the development world, that being how programming languages are developing, which makes an interesting case for both logical and functional paradigms in the light of this research. Programming languages are being developed to cross over paradigms and share features to allow developers the most control over how they use their selected language. Both Java and .NET are good examples of this, allowing functional constructs directly embedded into the language making procedural, functional, and object-oriented paradigms all available as native code. This forces a reaction out of this research to open the scope to include functional and even reconsider logical when developments such as LINQ [298] show an adoption of the logical paradigm by the .NET framework.

The final factor in this discussion should be the difficulty of achieving such an ambitious goal as generic automated software quality assurance, and therefore the practicality of this aim. There are many difficulties, especially when combining programming paradigms into a single representation due to the constructs being so different in cases like logical and procedural. There is a temptation to try to include everything that drives issues like level of abstraction and possibility of returning to the original language. One solution is to do everything at the instruction level; however, the number of quality assurance techniques starts to dwindle, as many were designed for high-level languages. The other option is to abstract even further, but then the possibility of returning to a high-level language is unlikely after any techniques have been implemented. Therefore, practicality dictates that a balance must be found, explaining the decisions made in the previous section, although it must be made clear that this is not the only method and that success could be found using an alternative representation or even a different approach to the problem as a whole.

6.2 COMPLETE FRAMEWORK / FUTURE WORK

As this research has outlined the feasibility of a programming language-independent (partially paradigm-independent) automated quality assurance framework, it is important to express how this could be taken from its current skeleton state in LIQA to a useful and practical product. There are several key areas that require assessing and discussing with regards to expansion and inclusion of the framework. These include direct expansion, optional additions, integration with tools, and theorised utility expansion.

6.2.1 Direct Expansion

A few direct and obvious expansions for the skeleton implementation of this framework (LIQA) are required before it would be a viable and usable product. This essentially encompasses programming language and techniques.

Firstly, a single full programming language, most likely Java, should be represented with all of its constructs. Keeping in mind that the constructs could relate to other programming languages, e.g. lists, vectors, dictionaries, etc., these should be designed using the abstract syntax tree meta-model specification and kept as generic as possible. Interaction with these objects can be utilised with other programming languages to achieve generic quality assurance. Techniques, therefore, do not have to be re-written, and the constructs do not have to be re-written for each programming language. Extending beyond the core generic abstract syntax tree metamodel, representation should be implemented with careful consideration and in practice should take place in collaboration with experts from differing fields with experience in a wide variety of programming languages. Following this, the obvious expansion is for other programming languages to have parsers written allowing them to be integrated into an implementation of this framework. Again, this should be done within the guidelines of the abstract syntax tree meta-model, and all new constructs should go through an adding procedure, which should be a collaboration between domain and programming language experts in order to ensure that the constructs can be applied generically to other programming languages where possible.

Another area of direct expansion is automated quality assurance techniques. Although there are many generic techniques, as shown in this research, it is anticipated that this internal representation is to be widely used, because of its inclusion of open source and standard specification, and therefore it would be an important addition to include non-generic automated quality assurance techniques as well as generic ones. This would allow a more complete framework, assessing programming languages for their specific issues as domain-specific standards could be adopted. This framework, combined with standards across the board of programming languages, could make this framework a standard itself allowing programming language developers, IDE developers, and software developers to have a standard tool for assessing quality and improving the quality of all programs. This, in turn, could then be used by standard agencies to generate configuration files to assess software for that specific standard. This is of course speculation and would require a large-scale adoption of the framework in various industries including programming language development, quality assurance, standards agencies, and software development.

In the same vein as the aforementioned addition of automated quality assurance techniques, because computation and programming is a continuously developing field, used in various industries in a variety of ways, it is important that new quality assurance techniques, developed to assist in advancing forms of programming, be considered. As the rate of increase in processing slows [299], a focus is being drawn to multi-processor computation [300]. This has had a major impact on software development in several domains that deal with programs that can be run concurrently (an application that can split its computation into parallel running processes or threads). As this study's initial focus is on programming for scientific development, there is research that aims to create optimisations to software, allowing it to become parallel processor-focused in a more extreme way than mainstream applications already are [301]. Optimisations like these can be implemented in a framework such as this, as the optimisations discussed are implemented within a beta version (at this time) of WinFPT that has been analysed and assessed in this research, and by the developers of WinFPT, concluding that all techniques implemented in WinFPT can be adopted into this framework, given that the pre-processor used in WinFPT is

used before parsing into the internal representation. This has been addressed because a standard-based framework must be adaptable to growing concerns within its field.

6.2.2 Grammar Prefix

A grammar prefix to the framework would be a large undertaking but it may improve programming language incorporation, allowing the framework to be added to in a quicker and simpler manner than writing a parser for each individual programming language. The grammar prefix would generate a parser for a programming language where a programming language specification, or grammar, has been written that describes the programming language's constructs and syntax. This process is described in Figure 6.2.1:

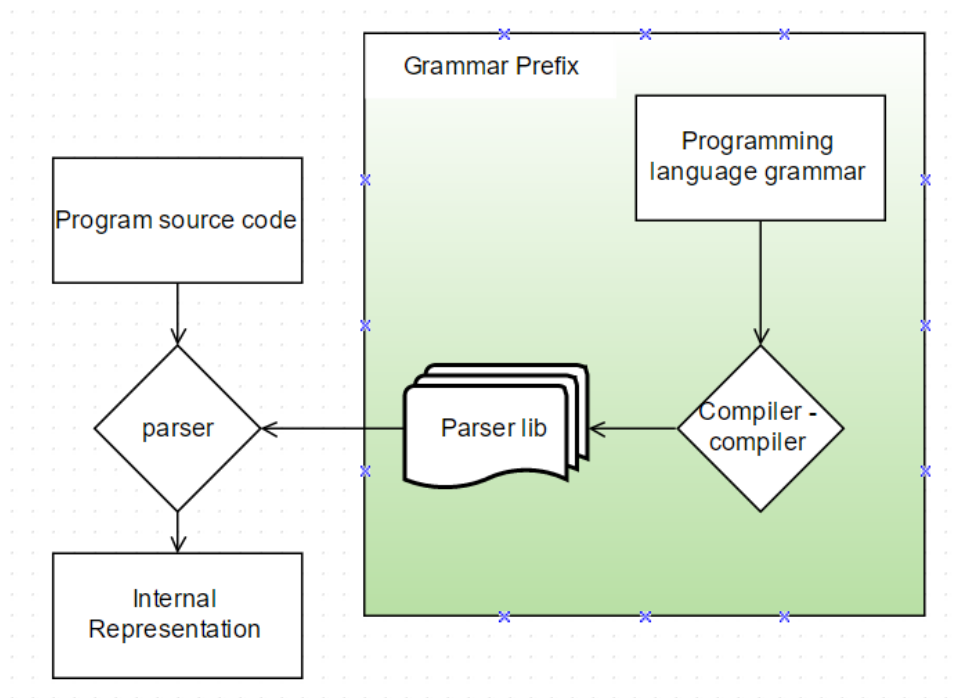


Figure 6.2.1 – Language Independent Quality Assurance (LIQA) grammar prefix

This grammar should itself be a standard, aligning with the aim of the research, but also with the ability to cover a multitude of programming languages, e.g. Backus–Naur Form (BNF) is a standard for writing grammars [302], but Extended Backus–Naur Form (EBNF) has been

developed to cover more constructs and properly describe more complex programming languages [303]. This ‘description’ of a programming language would be inputted and from this information a parser would be generated to translate programs written in that programming language into the abstract syntax tree meta-model.

There are several issues or limitations regarding this method of including programming languages, one of which is that adding a grammar prefix itself would be no small feat and would have to be extensively checked, as it would essentially be a compiler generating a compiler, similar to YACC [304] or JavaCC [98]. Another issue with this is that these meta-programming languages are known as context-free grammars that cannot represent context-sensitive programming languages such as Fortran [305], although these programming languages usually have a context-free specification and the context sensitivity is dealt with using special conventions [303].

Though there are limitations, the amount by which this grammar prefix would improve the framework’s adoption of new programming languages, means that there is still an argument for making this an extremely important part of future work arising from this research.

6.2.3 Query Addition

The most common quality assurance technique, by pure volume, is pattern matching, as different coding standards evaluate different aspects of the program that do not meet with their specification. As pattern matching is a major part of quality assurance techniques, it seems like writing classes for each individual pattern would take an extraordinary amount of time and effort. It is therefore proposed that a query system could be made to minimise this task whereby a pattern could be expressed with a combination of fixed nodes and wildcard nodes to allow the addition of patterns more quickly, easily, and simply.

Two approaches could be taken to create a system for queries to be performed on the internal representation used in this framework. First of all, a custom query system could be developed, the advantages being that it can be optimised for the abstract syntax tree meta-model and can be manipulated to adapt to user requirements, e.g. new query features for bespoke pattern matching. This solution would have the most versatility when compared with the second solution, which is to have an XML tree outputted and allow the queries to be written in a language such as XQuery [306] or XQL [307]. This approach is simple to accomplish as the abstract syntax tree meta-model can be outputted in an XML format simply, and the query languages are well known, meaning that developers would not be required to learn a custom query system. However, there may be unforeseen limitations in terms of requirements of pattern matching quality assurance techniques.

The recommended approach for a query system would be outputting in XML and using an XML query language as this would not only be the simplest to implement but also simpler for adoption, which is an important factor when considering the framework being released for general purpose use.

6.2.4 Linking with IDE

A further improvement for this framework would be to link the implementation with an Integrated Development Environment (IDE). In an ideal world, when the framework has been implemented, IDE developers would integrate the framework directly into the IDE, allowing a seamless user experience that is consistent regardless of programming language or chosen development environment. However, this will not be the case until the framework is recognised and implemented on a large scale. For integration with IDEs in the short term, targeting popular IDEs that support a range of programming languages would be ideal, such as NetBeans and Eclipse. The framework should be implemented in a way that allows updates to be rolled out without having to update the plug-ins individually, which could be accomplished by making the implementation of the framework a command-based implementation with the plug-ins as just graphical front ends integrated into the IDEs.

6.2.5 Dynamic Analysis

Other than dynamic metrics collected through a profiler-like system, dynamic analysis has been more or less disregarded in this research for automated quality assurance. This is due to the complications involved with implementing a structured and reliable form of dynamic analysis that is sufficiently generic to be consistent across all programming languages.

The basis of dynamic analysis, after and including profiling and unit testing, is a virtual machine capable of running the programs and allowing breakpointing and stepping through code. With this tool, it could be said that all forms of dynamic analysis, for each particular programming language, would be possible. Therefore, although it would be an interesting topic for discussion with regards to implementing full dynamic analysis within the current framework, it is important to consider that this was judged beyond the scope of this research due to the not insignificant complexity and sheer size of the task.

There are three options with regards to dynamic analysis that this framework could adopt:

One is to use the virtual machines or compilers intended for each programming language, e.g. the Java Virtual Machine for Java and the .Net Runtime for .Net language, etc., then to utilise their own features to implement the desired dynamic analysis techniques. This has the advantage of pre-existing virtual machines being available and also that they are guaranteed to work with their respective programming language. However, there are several disadvantages. Compilers pose their own problems, and different compilers and virtual machines have their own quirks. For example, the order in which parameters are executed (if functional calls are used as parameters in a function call) is different in the iFort compiler and the gFortran compiler. Another disadvantage is that individual compilers and virtual machines implement different features, allowing some of the more advanced dynamic analysis techniques to be available for those programming languages that support it, making it essentially non-generic. A final disadvantage, and effectively the ‘nail in the coffin’ for this route, is that the maintenance and

workload required to adopt these virtual machines and compilers would be tremendous, vastly reducing any positive impact.

The second option would be to create a virtual machine that essentially compiled and ran the internal representation. This has many advantages; for example, all of the techniques implemented within this virtual machine would be available to all of the programming languages supported by the framework at any time. Although there would be some issues related to library calls, systems like mono use method mapping to combat this, but the extent to which programming language libraries would be compatible is uncertain at the start. Two major disadvantages make this approach questionable, one being that by creating this virtual machine and compiler, more issues could be introduced, and the complexity of creating a compiler that would support all these programming languages could create many conflicts. The second disadvantage is that this would probably involve more work than the initial route, which itself is on a huge scale, and therefore this path is effectively not feasible.

The final route, and this seems like an excuse, is not to include dynamic analysis. Initially, it seems like a means of reducing the workload because the prospective work seems too great, but several factors can be considered in favour of this route. The framework is intended to be added to IDEs as a way of automatically checking for code quality via techniques provided through various forms of research. Dynamic analysis automation is limited mainly to dynamic metrics and automated testing, both of which have either been implemented within the IDEs themselves or have frameworks dedicated to them. Therefore, adding dynamic analysis to this framework would essentially be redundant. The final fact to consider if automated dynamic analysis does develop and belong in the framework in a more impactful way, is that by being present as a plug-in to the IDEs, a link to the IDEs' virtual machine or compiler may prove a better form of including automated dynamic analysis into this framework.

6.3 SUMMARY

This chapter was targeted at completing the theoretical implications of additional paradigms being included under the scope of this research. These targets were laid out in Aim 3 and all of its objectives. This chapter showed that it would be feasible to include the functional programming paradigm; however, this may not be worthwhile. This chapter also reasoned that including the logical paradigm within the internal representation used in this framework is not possible and has little point. As well as focusing on paradigms, the chapter outlines future work that could be taken on to improve the inclusion of this research in the body of current automated software quality assurance, outlining ideas to improve adoption in this space.

Chapter 7. Conclusion

The conclusion of this research has several purposes. Initially, the project's ideals and scope need to be outlined as well as any research questions generated and considered by this study.

‘A generic framework to support programming languages of disparate paradigms can be designed to facilitate automated software quality assurance’ was the initial hypothesis for this research, in which ‘paradigms’ relates to programming paradigms and the programming languages that represent these. Automated quality assurance within the context of this research is any technique designed to increase the quality of a program, that can be performed without intervention by the user. This includes code modification and identification of aspects of code, e.g. values representing code or highlighting problems in code. Previous work examined considered performing automated quality assurance techniques in a limited fashion with regards to programming languages and programming paradigms, although the automated quality assurance techniques implemented are similar across the board. Based on this statement and the context, several research questions arose:

- Which quality assurance activities are appropriate?
- If the programming language paradigms change, does the internal representation change?
- Do quality assurance activities change as the paradigm changes?

The structure of this chapter will therefore be as follows: Empirical Findings will discuss the research's conclusions with regard to the research questions; Theoretical Implication will consider the original contribution to knowledge and how this research can influence the field it is based in; Recommendation for Future Research will consider possible ways forward for this framework; Limitations of the Study will cover limitations in the implementation and choices made within this research project; and finally, Conclusion of the conclusion, which will contain final statements and round off the research as a finished piece of work. The chapter will end with

a Personal Note that will contain comments about the experience as well non-research-focused comments that seem relevant to this work.

7.1 EMPIRICAL FINDINGS

It is important to outline the findings of the research, not only to justify the project but also to understand why the project was attempted in the way it has been. Firstly, the most important findings to highlight are those that relate to the research questions and the initial statement.

The research started with an outline of the areas of interest. These areas included programming language, programming paradigm, program quality, and automated quality assurance. The research then moved on to the basic proposal for the framework with, in the first instance, a focus on the internal representation, as this was at the center of the framework. The focus of the internal representation was, at first, on the imperative/procedural paradigm, as this was considered the programming paradigm closest to the research. From there, quality assurance techniques within the imperative/procedural, as well as certain object-oriented paradigms, were analysed. This was done with the use of a literature study followed by an analysis of automated quality assurance tools. This analysis, and techniques found in literature, formed the basis of a taxonomy of quality assurance techniques, the motivation of which was the implementation of the techniques. This allowed for broader statements to be made about individual categories and their feasibility within the scope of the framework of the research. After the formation of the taxonomies, a skeleton implementation of the framework was developed. This implementation was used in experiments to assess the feasibility of the framework as it stood, i.e. a paradigm-specific but quality assurance technique-generic framework. After the experiment and framework had been deemed successful, the focus of the research shifted towards disparate programming paradigms of which a formal analysis was performed in order to ascertain the feasibility of their inclusion within the framework. A segment was included, following the discussion of programming paradigm, that highlighted the range of this research and how it

could be extended in many forms as well, as an ideal aim for the overall fully implemented framework.

From this flow of research, key questions arose to produce an evaluation of the hypothesis ‘A generic framework to support programming languages of disparate paradigms can be designed to facilitate automated software quality assurance’. These included:

Which quality assurance activities are appropriate?

This was initially answered with the taxonomy of quality assurance techniques, as the analysis of literature and quality assurance tools showed that there are quality assurance techniques that are programming language-generic and span across multiple programming languages within a single paradigm. However, this question spawned the further query:

Do quality assurance activities change as the paradigm changes?

The analysis of programming paradigms did show that there were quality assurance techniques developed specifically for that paradigm that it would not make sense to use outside of that paradigm. However, an overlap between the imperative, object-oriented, and functional paradigms was seen. So, the direct answer to this question is ‘sometimes’, which may not seem helpful. However, considering multi-paradigm programming languages as well as single-paradigm languages that are adopting constructs from other paradigms, this overlap is important to consider when creating a quality assurance tool for those programming languages. The final question posed by this research was:

If the programming language paradigms change, does the internal representation change?

This question has two answers, as having only one would be over-simplifying. If the paradigms have remotely similar approaches or constructs, then the internal representation does not have to change. This is the case for imperative, object-oriented, and procedural programming paradigms; though these paradigms are disparate, they also have similar constructs and approaches that allow them to relate better at an abstract level, which is where an internal representation should lie. However, if an attempt is made to converge into one internal representation, the logical programming paradigm, and any of the previously mentioned

paradigms, then the answer to the question would be in the affirmative. This is due to the approach taken, as while the logical paradigm attempts to build relationships between data and solve queries on the data, a logical program does not describe the process, whereas the other programming paradigms describe, more or less, how to achieve a goal.

This leaves the outcome of this research when considering the hypothesis ‘A generic framework to support programming languages of disparate paradigms can be designed to facilitate automated software quality assurance’. The outcome has overall been positive, although the hypothesis is not completely correct. As shown during testing and analysis, a procedural and object-oriented based framework is feasible, and the paradigm discussion highlighted that the functional paradigm and any paradigm built of a combination of these three paradigms can be implemented into a single internal representation within the framework. However, the addition of any logical paradigm component would be unrealistic and contradicts the hypothesis.

7.2 THEORETICAL IMPLICATION

This section of the conclusion seeks to outline the original contribution to knowledge made by this research as well as to illustrate where this project is situated with regard to the field and current knowledge. The section will also outline the plausible impact or implications this research might have as an influence on current practice.

As part of this research, a taxonomy of quality assurance techniques was created from the novel viewpoint of implementation requirements and methods. Although this stands alone in its own right, in the scope of this research, it served as a tool for simplifying the process of validating the framework across automated quality assurance techniques. This taxonomy could serve as a tool for further evaluation of the framework, to allow others to categorise additional quality assurance techniques for other research, or to assess feasibility of implementation under the framework discussed in this research.

As a stepping stone in this work, an imperative/procedural programming paradigm was the initial starting point, enabling automated quality assurance techniques to be generalised across a 'single' paradigm. This was deemed successful, resulting in a significant original contribution to knowledge being put forward. Moreover, extending this led to the programming paradigm-generic framework and, though there are limitations, this novel idea has been shown to be feasible. The implications of this work have the potential to be tremendous, with the adoption of an open source and standardised approach, as taken in this work. The framework could be used to create a layer between programming languages and automated quality assurance so that programming language developers and users have a full choice between programming languages without having to consider quality as an issue. This, on the flip side, allows any new quality assurance technique to be applied onto multiple programming languages, therefore extending to multiple platforms simply. With regard to the main focus group of this research, the importance of automated quality assurance within the scientific domain has already been clarified and, with the adoption of this framework, the ability to roll out standards for quality would be simple and would shift the focus from making the program generate the output required to the purpose of the program, e.g. within a climate control model, such as WRF, all of the code could be analysed by a single tool, regardless of programming language (C and Fortran in the case of WRF). This would remove concerns concerning whether a program is functional because its implementation is incorrect, as this tool could become 'trusted' and therefore the framework could itself become a standard.

7.3 RECOMMENDATIONS FOR FUTURE RESEARCH

Several expansions for the framework and suggestions for future study in line with this research have already been outlined in the previous chapter. This discussion was extensive due to the many and varied directions that expansion of this research could take. It is therefore important that a priority for future research should be to make this framework adapt and expand in a manageable and controlled manner, ensuring the best use of time and input to this framework.

It can be argued that each of the following areas are suitable candidates for future research:

- Programming language addition
- Quality assurance techniques addition
- Grammar prefix
- Internal representation query system
- IDE linking
- Dynamic analysis system

However, logic dictates in this matter that though this framework has been deemed feasible, a more through proof needs to be obtained before expansion can occur. Priority, therefore, should be given to implementing a full programming language into the framework. If the proof obtained is positive, then further research beyond this should enable the framework to develop at a faster pace. Therefore, developing the querying system on the internal representation, as well as the inclusion of the grammar prefix, should be considered next in line for additional content of the framework.

7.4 LIMITATIONS OF THE STUDY

There are several limiting factors to this research that should be clearly identified and explained. Firstly, the research assumes that the previous work done with the abstract syntax tree meta-model is applicable to the range of programming languages described in its specification. This assumption allowed, initially at least, the removal of cross programming language compatibility, and reduced the scope significantly enough to make the research feasible within the allocated time. This also highlights that the research has neither implemented nor run tests on additional programming languages of a similar paradigm, nor has it implemented programming languages stated to be compatible in later stages of this research, e.g. functional paradigm. Though this is a crucial limitation, such procedures cannot be expected in this research, and therefore such a large-scale implementation should be considered for future research. The final limitation to note is the implementation and testing results. Some the results did not match predictions, not because the theorised results were incorrect, but because the implementation of

specific techniques was done incorrectly. It might therefore be argued that this could have an effect on the outcome of the framework.

Though these issues have been raised, future work in this area, as identified above, would not only fill the gaps but would also strengthen the framework and enable development to incorporate methods of expanding it beyond its identified limitations. It should be noted that these limitations are not critical, as the assumptions made in this research are grounded in previous research in which a large number of organisations took part, and though there were issues with the implementation, the results are nevertheless positive and the errors can be explained so as to conform to the predictions, without losing validity.

7.5 PERSONAL REFLECTION

Overall, I believe this research has been successful in both identifying a major area of advancement for quality assurance of software as well as providing an adequate solution. It is, however, evident to me that a substantial level of development is still required before any impact could be made on the current state of industry and academia surrounding this research.

One insight I have had whilst active in this research project is that the balance between planning and implementation is a very fine line when learning is happening throughout the process. If this balance is not proper and more planning is performed, the output would be less able to adapt to any new information that may present itself over the large amount of time that the research is carried out. Alternatively, if less planning is done, the project has the risk of becoming too large or even spinning out of control, making the outputs' aim less impactful and too broad without much penetration into a specific point.

To reflect on this experience, I would have to conclude that the framework theory is sound; however, the tool LIQA itself could have been better planned, which would have provided simpler expansion and addition in later periods of research, increasing productivity;

however, like I mentioned before, it is a fine balance and difficult to get right unless you have been through the process before. I believe that I have had many valuable lessons throughout this project and believe that the success of this thesis will prove a massive feat in my personal and professional development.

Overall, I believe this experience has left me a better researcher. One factor showing this was my ambition to change the way we look at programming language development. Although it is good to have such passion and motivation, the expectation should be realistic; otherwise, tangents and fuzzy scope will overcome even the best researcher. Learning more about the research process and how to manage such a wide area and vast depth of knowledge whilst still challenging and driving forward change has made me humble and has changed my view on good research practice. Finally, one of the most significant lessons that I have learned and believe those in my position learn is to identify your transition from student to educator when discussing and explaining your own field and how this change affects the way in which you approach people differently. It has been an enjoyable and sometimes terrible path to walk, but one worth taking.

7.6 CONCLUSION OF THE CONCLUSION

To conclude this work, it is important to re-engage with the aims. ‘Outline a framework based on Procedural and Object-Oriented Paradigms’, and ‘Develop a Skeleton Framework’ were both completed successfully and a framework was developed that could be used in the area of automated software quality assurance. ‘Expand the footprint of the framework by discussing the theoretical inclusion of other programming paradigms’, however, has identified that a universal framework, encompassing all programming paradigms, as proven in this research, is not plausible using this internal representation. It is important to understand that, of the four main paradigms discussed in this research, only one is not compatible with the framework and the importance of this paradigm, within the context of the research and its impact, is small. Overall, this research should be considered a success, and it can be concluded that the argument

for this framework is sufficiently strong to attempt the task of generating this into a standard for quality assurance across the paradigms of functional, procedural, and object-oriented paradigms.

Chapter 8. Bibliography

- University of Southampton, "What is your paradigm?," University of Southampton, [Online]. Available: http://www.erm.ecs.soton.ac.uk/theme2/what_is_your_paradigm.html. [Accessed 2015].
- 1]
- M. Hammond and J. Wellington, *Research Methods : Key Concepts*, Oxon: Routledge, 2013.
- 2]
- C. W. Dawson, "Research," in *Projects in Computing and Information Systems: A Students Guide*, Second Edition ed., Harlow, Essex: Peasons Educations Limited, 2009, pp. 15-37.
- 3]
- S. W. Ambler, "Evolutionary Software Development: How Data Activities Fit In," [Online]. Available: <http://www.agiledata.org/essays/evolutionaryDevelopment.html>. [Accessed 23 November 2012].
- 4]
- T. Greening, *Computer Science Education in the 21st Century*, Springer Science & Business Media, 2012.
- 5]
- M. C. Daconta, L. J. Obrst and K. T. Smith, "Chapter 7: Understanding Taxonomies," in *The Semantic Web : A Guide to the Future of XML, Web Services, and Knowledge Management*, Indianapolis, Wiley, 2003, pp. 145-155.
- 6]
- D. Owens and M. Anderson, "A Generic Framework for Automated Quality Assurance of Software Models: Supporting Languages of Multiple Paradigms," *Journal of Software*, vol. 8, no. 9, 13-14 April 2013.
- 7]
- EHU, "Edge Hill Schools of Business and Computing Ethics Policy for Undergraduate and All Postgraduate Research," 2012.
- 8]
- A. Laird, "The Four Major Programming Paradigms Topic Paper #17," *Computer Science*, 3 April 2009.
- 9]
- S. S. Brilliant and T. R. Wiseman, "The first programming paradigm and language

- 10] dilemma," *SIGCSE '96 Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education*, vol. 28, no. 1, pp. 338-342, March 1996.
- A. Aaby, "Functional Programming," 1996. [Online]. Available:
11] <https://student.brighton.ac.uk/burks/pcinfo/progdocs/plbook/function.htm>. [Accessed 22 11 2012].
- J. Liu, X. Jin and K. C. Tsui, *Autonomy Oriented Computing: From Problem Solving to*
12] *Complex Systems Modeling*, Springer Science & Business Media, 2006.
- W. Schreiner, "Functional and Logic Programming," in *COMPUTER SCIENCE AND*
13] *ENGINEERING*.
- D. Clark, *Beginning C# Object-Oriented Programming*, Apress, 2011.
14]
- L. Day, "Programming Paradigms," *Computerphile*, 30 August 2013. [Online]. Available:
15] <https://www.youtube.com/watch?v=sqV3pL5x8PI>. [Accessed 21 09 2015].
- I. o. Microelectronics, "4.2 Evolution of Programming Paradigms," [Online]. Available:
16] <http://www.iue.tuwien.ac.at/phd/heinzl/node32.html>. [Accessed 11 12 2012].
- D. D. Spinellis, "Programming Paradigms as Object CLasses," 1993. [Online]. Available:
17] <http://www.dmst.aueb.gr/dds/pubs/thesis/PhD/html/thesis.pdf>. [Accessed 1 12 2012].
- S. Dmitriev, "Language Oriented Programming: The Next Programming Paradigm,"
18] [Online]. Available: <http://www.onboard.jetbrains.com/articles/04/10/lop/>. [Accessed 11 12 2012].
- P. V. Roy, "Programming Paradigms for Dummies: What Every Programmer Should
19] Know," [Online]. Available: <http://www.info.ucl.ac.be/~pvr/VanRoyChapter.pdf>. [Accessed 22 Novemeber 2012].
- DedaSys, "Programming Language Popularity," 13 April 2011. [Online]. Available:
20] langpop.com. [Accessed 22 November 2012].

- 21] C. Artho, A. Biere, P. Eugster, M. Baur and B. Zweimüller, "JNuke: Efficient Dynamic Analysis for Java," *Proc. CAV '04*, 2004.
- 22] Q. Systems, "Cantata - The Unit Testing Tool for C/C++," [Online]. Available: <http://www.qa-systems.com/cantata.html> . [Accessed 09 11 2012].
- 23] K. Harrison, "Static Code Analysis on the C-130J Hercules Safety-Critical Software," *UK International Systems Safety Conference*, 1999.
- 24] MathWorks, "Static Analysis with Polyspace Products," 1994. [Online]. Available: <http://www.mathworks.co.uk/products/polyspace/>. [Accessed 09 11 2012].
- 25] SimCon, "SimCon - Fortran Analysis, Engineering & Migration," 1995. [Online]. Available: <http://www.simconglobal.com/>. [Accessed 09 11 2012].
- 26] MSDN, "Compiling to MSIL," [Online]. Available: [http://msdn.microsoft.com/en-us/library/c5tkafs1\(v=vs.71\).aspx](http://msdn.microsoft.com/en-us/library/c5tkafs1(v=vs.71).aspx). [Accessed 21 November 2012].
- 27] P. Newcomb, "Abstract Syntax Tree Metamodel Standard ASTM Tutorial 1.0," October 2005. [Online]. Available: http://www.omg.org/news/meetings/workshops/ADM_2005_Proceedings_FINAL/T-3_Newcomb.pdf. [Accessed 5 2 2013].
- 28] A. Tripp, "Manual Tree Walking Is Better Than Tree Grammars," 22 February 2006. [Online]. Available: <http://wwwantlr2.org/article/1170602723163/treewalkers.html>. [Accessed 5 2 2013].
- 29] J. W. Backus, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Hettick, R. A. Nelson, D. Sayre, P. B. Sheridan, H. Stern, I. Ziller, R. A. Hughes and R. Nutt, "The FORTRAN Automatic Coding System," *western joint computer conference: Techniques for reliability, ACM*, pp. 188-198, Fall 1957.
- 30] T. Mikkonen and A. Taivalsaari, "Using JavaScript as a real programming language," *Technical Report. Sun Microsystems, Inc.*, p. 17, 2007.

- 31] J. Misek and F. Zavoral, "Mapping of Dynamic Language Constructs into Static Abstract Syntax Trees," *ICIS '10 Proceedings of the 2010 IEEE/ACIS 9th International Conference on Computer and Information Science*, pp. 625-630 , 2010.
- 32] T. Parr, "ANTLR," [Online]. Available: <http://www.antlr.org>. [Accessed 4 2 2013].
- 33] "Grammar List," [Online]. Available: <http://www.antlr3.org/grammar/list.html>. [Accessed 5 2 2013].
- 34] M. Van Den Brand, P. Moreau and J. Vinju, "A generator of efficient strongly typed abstract syntax trees in Java," *EE Proceedings - Software Engineering* 152, 2 (2005) 70--87, pp. 70-87, 2005.
- 35] G. Fischer, J. Lusiardi and J. Gudenberg, "Abstract Syntax Trees – and their Role in Model Driven Software Development," *Software Engineering Advances, 2007. ICSEA 2007. International Conference on*, p. 38, 25-31 August 2007.
- 36] Y. Ichisugi, "Source code translating method, recording medium containing source code translator program, and source code translator device". United States Patent 6516461, 4 February 2003.
- 37] ASM, "Model Driven Modernization," 2012. [Online]. Available: <http://www.automatedsoftwaremodernization.com/component/content/article/3.html> . [Accessed 5 2 2013].
- 38] P. Tu and J. Liu, "Research on technology of transforming Abstract Syntax Tree of JAVA Language to Implementation Layer of Procedure Blueprint," *Information Science and Engineering (ICISE), 2010 2nd International Conference on*, pp. 1-5, 4-6 December 2010.
- 39] R. Akers, I. Baxter, M. Mehlich, B. Ellis and K. Luecke, "Re-engineering C++ Component Models Via Automatic Program Transformation," *Reverse Engineering, 12th Working Conference on*, 7-11 November 2005.
- G. Pierce, Unity IOS game Development, Birmingham: PACKT Publishing, 2012.

40]

OMG, "Catalog Of Omg Modernization Specifications," 19 July 2012. [Online]. Available:

41] http://www.omg.org/technology/documents/modernization_spec_catalog.htm . [Accessed 5 2 2013].

J. Tian, Software Quality Engineering : Testing Quality Assurance and Quantifiable

42] Improvment, D. F. Shafer, Ed., Hoboken, New Jersey: IEEE Computer Society, 2005.

R. Yin and X. M. Ding, "How to improve the quality of software testing," *Systems and*

43] *Informatics (ICSAI), 2012 International Conference on*, pp. 2533-2536, 19 May 2012.

L. Rosenberg, "Software quality assurance engineering at NASA," *Aerospace Conference*

44] *Proceedings, 2002. IEEE*, vol. 5, pp. 5-2569 - 5-2575, 2002.

D. P. Wesenberg and K. Vansaun , "A system approach for software quality assurance,"

45] *Aerospace and Electronics Conference, 1991. NAECON 1991., Proceedings of the IEEE 1991 National*, pp. 771-776, 20 - 24 May 1991.

R. L. Krutz, R. D. Vines and G. Brunette, Cloud Security : A Comprehensive Guide to

46] Secure Cloud Computing, Hoboken, New Jersey: Wiley, 2010.

F. Losavio, L. Chirinos, N. Lévy and A. Ramdane-Cherif, "Quality Characteristics for

47] Software Architecture," *Journal of Object Technology*, vol. 2, no. 2, pp. 133-150, March 2003.

BITS, "Software Assurance Framework," January 2012. [Online]. Available:

48] <http://www.bits.org/publications/security/BITSSoftwareAssurance0112.pdf>. [Accessed 23 October 2012].

G. U. Maheswari and V. V. R. Prasad, "Optimized Software Quality Assurance Model for

49] Testing Scientific Software," *International Journal of Computer Applications*, vol. 36, no. 7, December 2011.

A. Austin and L. Williams, "One Technique is Not Enough: A Comparison of Vulnerability

- 50] Discovery Techniques," *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*, pp. 97-106, 22-23 September 2011.
- D. Bell and P. G. Brat, "Automated Software Verification & Validation: An Emerging
51] Approach for Ground Operations," *Aerospace Conference, 2008 IEEE*, pp. 1 - 8, 1-8 March 2008.
- B. Chess and C. Wysopal, "Software Quality Assurance for the Masses," *Security &*
52] *Privacy, IEEE*, vol. 10, no. 3, pp. 14 - 15, May 2012.
- N. Truong, P. Roe and P. Bancroft, "Static Analysis of Students' Java Programs," *ACE '04*
53] *Proceedings of the Sixth Australasian Conference on Computing*, vol. 30, pp. 317-325, 2004.
- G. Naumovich, G. Avrunin, L. Clarke and L. Osterweil, "Applying Static analysis to
54] software architectures," in *Software Engineering - ESEC/FSE '97*, M. Jazayeri and H. Schauer, Eds., Springer Berlin / Heidelberg, 1997, pp. 77-93.
- T. Reps, T. Lev-Ami, M. Sagiv and R. Wilhelm, "Putting Static Analysis to Work for
55] Verification:A Case Study," *ISSTA '00 Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pp. 26-38, 2000.
- N. Ward, "Code Verification with the aid of MALPAS," *High Integrity Ada, IEE*
56] *Colloquium on*, pp. 3/1-3/3, 8 1 1991.
- D. Balzarotti, M. Cova, V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel and G. Vigna,
57] "Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications," *Security and Privacy, 2008. SP 2008. IEEE Symposium on*, pp. 387- 401 , 2008.
- Parasoft, "C/C++test," 24 09 2012. [Online]. Available:
58] <http://www.parasoft.com/jsp/products/cpptest.jsp?itamlid=47>. [Accessed 09 11 2012].
- R. Fairley, "Tutorial: Static Analysis and Dynamic Testing of Computer Software,"
59] *Computer*, vol. 11, no. 4, pp. 14-23, April 1978.

- 60] A. Biere and C. Artho, "Combined Static and Dynamic Analysis," *In Proc. Intl. Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL '05)*, 2005.
- 61] M. Salah, S. Mancoridis, G. Antoniol and M. Di Penta, "Scenario-driven dynamic analysis for comprehending large software systems," *Software Maintenance and Reengineering, 2006. CSMR 2006. Proceedings of the 10th European Conference on*, pp. 80 - 90, 22-24 March 2006.
- 62] W. E. Wong, "An Integrated Solution for Creating Dependable Software," *Computer Software and Applications Conference*, pp. 269 - 270, 2000.
- 63] SonarSource S.A, "Sonarqube," 31 July 2014. [Online]. Available: <http://www.sonarqube.org/>.
- 64] M. Glinz, "Software Product Qualityc," 2014. [Online]. Available: SonarSource S.A.
- 65] G. Developers, "Running Dead Code Analysis," Google , 27 May 2015. [Online]. Available: • <https://developers.google.com/java-dev-tools/codepro/html/tasks/maintopic?hl=en> .
- 66] Infosys, "Infosys Functional Test Case Generator," Infosys, [Online]. Available: • <https://www.infosys.com/IT-services/independent-validation-testing-services/service-offerings/Pages/functional-test-case-generator.aspx> .
- 67] P. Fowler and S. Rifkin , "Software Engineering Process Group Guide," CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1990.
- 68] N. Iqbal and J. . M. R. Qureshi, "Improvement of Key Problems of Software Testing in Quality Assurance," *Science International-Lahore*, vol. 21, no. 1, pp. 25-28, March 2009.
- 69] M. Visconti and L. Guzman, "A Measurement-Based Approach for Implanting SQA and SCM Practices," *SCCC '00 Proceedings of the XX International Conference of the Chilean Computer Science Society*, p. 126, 2000.
- J. Whittaker, "What Is Software Testing? And Why Is It So Hard?," *IEEE Software*, vol. 17,

- 70] no. 1, pp. 70-79, January 2000.
- A. Bertolino, "Software Testing Research: Achievements, Challenges, Dreams," *Future of Software Engineering*, 2007. FOSE '07, pp. 85 - 103 , 23-25 May 2007.
- 71]
- K. Karhu, T. Repo, O. Taipale and K. Smolander, "Empirical Observations on Software Testing Automation," *Software Testing Verification and Validation*, 2009. ICST '09. International Conference on, pp. 201 - 209, 1-4 April 2009.
- 72]
- X. Dianxiang, W. Xu, M. Kent, L. Thomas and L. Wang, "An Automated Test Generation Technique for Software Quality Assurance," *Reliability, IEEE Transactions on*, vol. 64, no. 1, pp. 247-268, 2015.
- 73]
- M. Dimjašević and D. Giannakopoulou, "Test-case generation for runtime analysis and vice versa: verification of aircraft separation assurance," *In Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pp. 282-292, 2015.
- 74]
- L. Yu, J. Zhou, . Y. Yi, . P. Li and . Q. Wang, "Ontology Model-based Static Analysis on Java Programs," *Computer Software and Applications*, 2008. COMPSAC '08. 32nd Annual IEEE International, pp. 92-99, 28 July 2008.
- 75]
- L. You, "A markup language for java bytecode," *Systems and Informatics (ICSAI)*, 2012 International Conference on, pp. 2420 - 2424, 19-20 May 2012.
- 76]
- A. Kolosov, "Using Static Analysis in Program Development," 31 January 2008. [Online]. Available: <http://www.viva64.com/en/a/0017/>. [Accessed 5 2 2013].
- 77]
- CERT, "ROSE Overview," 2008. [Online]. Available: <https://www.securecoding.cert.org/confluence/download/attachments/3524/Rose+1+Overview+v3.pdf>. [Accessed 5 2 2013].
- 78]
- C. Christopher, "Evaluating Static Analysis Frameworks," *Carnegie Mellon University Analysis of Software Artifacts* , 10 May 2006.
- 79]
- T. Ball, "The Concept of Dynamic Analysis," *ESEC/FSE-7 Proceedings of the 7th*

- 80] *European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pp. 216-234 , 6 November 1999.
- OW2, "ASM," 21 October 2012. [Online]. Available: <http://asm.ow2.org> . [Accessed 5 2
81] 2013].
- I. Sommerville, *Software Engineering*, vol. 9, Boston, Massachusetts: Pearson Education,
82] 2010.
- R. Jeffries and L. Lindstrom, "Extreme Programming and Agile Software Development
83] Methodologies," *Information Systems Management*, vol. 21, no. 3, pp. 41-52, 2004.
- S. W. Ambler, "Evolutionary Software Development," [Online]. Available:
84] <http://www.agiledata.org/essays/evolutionaryDevelopment.html>.
- E. L. May and B. A. Zimmer, "The Evolutionary Development Model for," *The
85] Evolutionary Development Model for*, vol. 47, pp. 39-41, 13 August 1996.
- Apple, "OSX," [Online]. Available: <http://www.apple.com/uk/osx/>.
86]
- Linux users, "Linux in the UK," [Online]. Available: <http://linux.co.uk/>.
87]
- w3schools, "OS Platform Statistics," 2013. [Online]. Available:
88] http://www.w3schools.com/browsers/browsers_os.asp.
- M. Gallagher, "Options for porting Objective-C/Cocoa apps to Windows," 15 April 2010.
89] [Online]. Available: <http://www.cocoawithlove.com/2010/04/options-for-porting-objective-ccocoa.html>.
- Xamarin, "Mono," [Online]. Available: <http://www.mono-project.com/>.
90]

- Oracle, "Java," [Online]. Available: <http://www.java.com/>.
- 91]
- Oracle, "Jaca SE Development Kit 7 Downloads," 2013. [Online]. Available:
 92] <http://www.oracle.com/technetwork/java/javase/downloads/jdk7-downloads-1880260.html>.
- Oracle, "Netbeans IDE," 2013. [Online]. Available: <https://netbeans.org/>.
- 93]
- Modisco, "Modisco," [Online]. Available: <http://www.eclipse.org/MoDisco/>.
- 94]
- Apache, "The Apache™ Batik Project," [Online]. Available:
 95] <http://xmlgraphics.apache.org/batik/>.
- V. Slavětínský and J. Kosek, "XsdVi," 22 March 2013. [Online]. Available:
 96] <http://sourceforge.net/projects/xsdvi/>.
- Modisco, "Modisco User Guide > Infastructure > GASTM > Overview," [Online].
 97] Available:
<http://help.eclipse.org/indigo/index.jsp?topic=%2Forg.eclipse.gmt.modisco.infra.doc%2Fdoc%2FMoDisco%2FComponents%2FGASTM%2FGASTM.html>.
- JavaCC, "Java Compiler Compiler tm (JavaCC tm) - The Java Parser Generator," [Online].
 98] Available: <http://javacc.java.net/>.
- OMG, "OMG Architecture-driven modernization: Abstract Syntax Tree metamodel
 99] (ASTM)," January 2011. [Online]. Available: <http://www.omg.org/spec/ASTM/1.0/PDF/>.
 [Accessed 5 2 2013].
- D. Owens and M. Anderson, "A Generic Framework for Automated Quality Assurance of
 100] Software Models - Implimentation of an Abstract Syntax Tree," *International Journal of
 Advanced Computer Science and Applications (IJACSA)*, vol. 5, no. 1, 2014.
- B. Farrimond and J. Collins, "Dimensional Interference Using Symbol Lives," *International*

- 101] *Conference: Software Engineering Theory and Practice*, 2007.
- G. Chatzieftheriou and P. Katsaros, "Test-Driving Static Analysis Tools in Search of C
102] Code Vulnerabilities," *COMPSAW '11 Proceedings of the 2011 IEEE 35th Annual Computer Software and Applications Conference Workshops*, pp. 96-103, 2011.
- B. Wichmann, A. Canning, D. Clutterbuck, L. Winsborrow, N. Ward and D. Marsh,
103] "Industrial perspective on static analysis," *Software Engineering Journal*, vol. 10, no. 2, pp. 69-75, 3 1995.
- A. Anywhere, "TestingAnywhere," [Online]. Available:
104] <http://www.automationanywhere.com/Testing/>. [Accessed 09 11 2012].
- S. Goldin, T. Luengwitayakorn and S. Supadarattanawong, "Test-driven development for
105] graphical UIs: A multi-platform toolset," *TENCON 2010 - 2010 IEEE Region 10 Conference*, pp. 2429 - 2433, 21-24 11 2010.
- NetBeans, "What's the Difference between NetBeans Platform and Eclipse RCP?," [Online].
106] Available: <https://netbeans.org/features/platform/compare.html>.
- Netbeans, "Java Hints," 21 March 2014. [Online]. Available:
107] http://wiki.netbeans.org/Java_Hints.
- University of Maryland, "<http://findbugs.sourceforge.net/>," 07 July 2014 . [Online].
108] Available: <http://findbugs.sourceforge.net/>.
- P. Mohan, "JUnit Testing in Netbeans," 28 February 2013. [Online]. Available:
109] <http://oopbook.com/junit-testing/junit-testing-in-NetBeans/>.
- Oracle, "Profiler," [Online]. Available: <https://profiler.netbeans.org/>.
110]
- The Eclipse Foundation, "Search : Programming Languages," [Online]. Available:
111] [http://marketplace.eclipse.org/search/site?f\[0\]=im_taxonomy_vocabulary_1%3A1966#search](http://marketplace.eclipse.org/search/site?f[0]=im_taxonomy_vocabulary_1%3A1966#search).

- Android, "Android Developer Tools," [Online]. Available:
112] <http://developer.android.com/tools/help/adt.html>.
- The Eclipse Foundation, "PHP Development Tools," 11 June 2014. [Online]. Available:
113] <http://www.eclipse.org/pdt/>.
- The Eclipse Foundation, "Quick Fix," [Online]. Available:
114] <http://help.eclipse.org/luna/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Freference%2Fpreferences%2Fjava%2Feditor%2Fref-preferences-content-assist.htm>.
- The Eclipse Foundation, "Eclipse Test & Performance Tools Platform Project," 25 February
115] 2011. [Online]. Available: <http://www.eclipse.org/tptp/>.
- Mountainminds GmbH & Co, "Java Code Coverage for Eclipse," 27 September 2012.
116] [Online]. Available: <http://www.eclemma.org/>.
- C. Walton and L. Walton, "EclipseMetrics," [Online]. Available:
117] <http://www.stateofflow.com/projects/16/eclipsemetrics>.
- A. Loskutov, "JDepend plugin for Eclipse: JDepend4Eclipse," [Online]. Available:
118] <http://andrei.gmxhome.de/jdepend4eclipse/>.
- Sourceforge, "PMD," 31 August 2014. [Online]. Available: <http://pmd.sourceforge.net/>.
119]
- Microsoft, "Visual Studio," [Online]. Available: <http://msdn.microsoft.com/en-us/vstudio/aa718325.aspx>.
120]
- Citizendium, "List of languages using the .NET Framework," 4 July 2014. [Online].
121] Available: http://en.citizendium.org/wiki/List_of_languages_using_the_.NET_Framework.
- Microsoft, "Visual Studio Languages," [Online]. Available: [http://msdn.microsoft.com/en-us/library/vstudio/ee822860\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/vstudio/ee822860(v=vs.100).aspx).
122]
- T. Thai and H. Lam, "A.2 Third-Party Languages for .NET," in *.NET Framework*

- 123] *Essentials*, O'Reilly Media, Inc., 2003.
- MSDN, "Programming Languages," 2003. [Online]. Available:
124] [http://www.msdn.microsoft.com/en-us/library/aa292164\(v=vs.71\).aspx](http://www.msdn.microsoft.com/en-us/library/aa292164(v=vs.71).aspx).
- MSDN, "MSDN," 2013. [Online]. Available: <http://msdn.microsoft.com/en-US/>.
125]
- MSDN, "Analyzing Application Quality by Using Code Analysis Tools," 2012. [Online].
126] Available: <http://msdn.microsoft.com/en-us/library/dd264897.aspx>.
- MSDN, "Finding Duplicate Code by using Code Clone Detection," 2012. [Online].
127] Available: <http://msdn.microsoft.com/en-us/library/hh205279.aspx>.
- MSDN, "Analyzing Managed Code Quality by Using Code Analysis," 2012. [Online].
128] Available: <http://msdn.microsoft.com/en-us/library/dd264939.aspx>.
- MSDN, "Measuring Complexity and Maintainability of Managed Code," 2012. [Online].
129] Available: <http://msdn.microsoft.com/en-us/library/bb385910.aspx>.
- MSDN, "Debugging with IntelliTrace," 2010. [Online]. Available:
130] [http://msdn.microsoft.com/en-us/library/dd264915\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(v=vs.100).aspx).
- MSDN, "Running Unit Tests with Test Explorer," 2012. [Online]. Available:
131] <http://msdn.microsoft.com/en-us/library/hh270865.aspx>.
- MSDN, "Analyzing Application Performance by Using Profiling Tools," 2012. [Online].
132] Available: <http://msdn.microsoft.com/en-us/library/z9z62c29.aspx>..
- DevExpress, "CodeRush for Visual Studio," [Online]. Available:
133] <http://www.devexpress.com/Products/CodeRush/>.
- Parasoft, "dotTEST," 19 June 2013. [Online]. Available:
134] <http://www.parasoft.com/jsp/products/dottest.jsp>.
- P. Smacchia, "NDepend," [Online]. Available: <http://www.ndepend.com/>.

135]

N. Nagappan and T. Ball, "Static analysis tools as early indicators of pre-release defect density," in *In Proceedings of the 27th international conference on Software engineering (ICSE '05)*, New York, 2005.

The Eclipse Foundation, "Platform," [Online]. Available: <http://wiki.eclipse.org/Platform>.
137]

MSDN, "Overview of the .NET Framework," [Online]. Available:
138] <http://msdn.microsoft.com/en-us/library/zw4w595w.aspx>.

Microsoft Research, "Phoenix Compiler and Shared Source Common Language Infrastructure," [Online]. Available: <http://research.microsoft.com/en-us/collaboration/focus/cs/phoenix.aspx>.

Oracle, "NetBeans Java Hint Module Tutorial," [Online]. Available:
140] <https://platform.netbeans.org/tutorials/nbm-java-hint.html>.

Oracle, "Debugger and Profiler," [Online]. Available:
141] <https://NetBeans.org/features/java/debugger.html>.

Oracle, "Debugging Multi-threaded Applications in NetBeans IDE," [Online]. Available:
142] <https://NetBeans.org/kb/docs/java/debug-multithreaded.html>.

Oracle, "Debugging Multi-threaded Applications in NetBeans IDE," [Online]. Available:
143] <http://wiki.netbeans.org/AnalyzeStackTrace>.

J. Huber, "Proposed Method for Achieving Increased Software Maintainability Through Documentation," *The Midwest Instruction and Computing Symposium*.

Oracle, "Java SE (Standard Edition)," [Online]. Available:
145] <https://netbeans.org/features/java/javase.html>.

D. Marx, "NetBeans 7 and Software Quality Environment," 27 August 2011. [Online]. Available: <http://marxsoftware.blogspot.co.uk/2011/08/netbeans-7-and-software->

146] quality.html.

University of Maryland, "FindBugs™ Fact Sheet," 07 July 2014. [Online]. Available:
147] <http://findbugs.sourceforge.net/factSheet.html>.

C. B. Almazan, N. Rutar and J. S. Foster, "A Comparison of Bug Finding Tools for Java,"
148] *Software Reliability Engineering, 2004. ISSRE 2004. 15th International Symposium on*, pp.
245-256, 2-5 November 2004.

University of Maryland, "FindBugs Bug Descriptions," 07 July 2014. [Online]. Available:
149] <http://findbugs.sourceforge.net/bugDescriptions.html>.

University of Maryland, "FindBugs FAQ," University of Maryland, 06 03 2015. [Online].
150] Available: <http://findbugs.sourceforge.net/FAQ.html>. [Accessed 07 04 2015].

J. Tessier, "Dependency Finder," 30 January 2014. [Online]. Available:
151] <http://depfind.sourceforge.net/>.

Sourceforge, "Checkstyle 5.7," 03 February 2014. [Online]. Available:
152] <http://checkstyle.sourceforge.net/>.

Sourceforge, "Writing Checks," 03 February 2014. [Online]. Available:
153] <http://checkstyle.sourceforge.net/writingchecks.html>.

JUnit, "JUnit," 05 August 2014. [Online]. Available: <http://junit.org/>.
154]

A. Schmid , "Getting started," 01 September 2014. [Online]. Available:
155] <https://github.com/junit-team/junit/wiki/Getting-started>.

M. Clark, "Frequently Asked Questions," 05 August 2014. [Online]. Available:
156] http://junit.org/faq.html#misc_3.

MSDN, "Debugging in Visual Studio," [Online]. Available: <http://msdn.microsoft.com/en-us/library/sc65sadd.aspx>.
157]

- MSDN, "How to: Watch an Expression in the Debugger," [Online]. Available:
158] <http://msdn.microsoft.com/en-GB/library/0taedcee.aspx>.
- MSDN, "Using the Assert Classes," [Online]. Available: <http://msdn.microsoft.com/en-GB/library/ms182530.aspx>.
159]
- MSDN, "Debug Your App by Recording Code Execution with IntelliTrace," [Online].
160] Available: <http://msdn.microsoft.com/en-gb/library/dd264915.aspx>.
- Microsoft, "Microsoft Visual Studio Ultimate 2012," [Online]. Available:
161] <http://www.microsoft.com/en-gb/download/details.aspx?id=30678>.
- MSDN, "Breakpoints and Tracepoints," [Online]. Available: [http://msdn.microsoft.com/en-us/library/ktf38f66\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/ktf38f66(v=vs.90).aspx).
162]
- Z. Naboulsi, "Setting a Tracepoint in source code," 2 July 2010. [Online]. Available:
163] <http://blogs.msdn.com/b/zainnab/archive/2010/02/07/setting-a-tracepoint-in-source-code-vstipdebug0010.aspx>.
- B. Sullivan, "Tracepoints," 10 10 2013. [Online]. Available:
164] <http://blogs.msdn.com/b/visualstudioalm/archive/2013/10/10/tracepoints.aspx>.
- R. Kath, "The Debugging Application Programming Interface," 5 November 1992. [Online].
165] Available: <http://msdn.microsoft.com/en-us/library/ms809754.aspx>.
- MSDN, "Editing Code (Visual C#)," July 2008. [Online]. Available:
166] [http://msdn.microsoft.com/en-us/library/ms228282\(v=vs.90\).aspx](http://msdn.microsoft.com/en-us/library/ms228282(v=vs.90).aspx).
- B. Johnson, "Getting Started," in *Professional Visual Studio 2012*, John Wiley & Sons,
167] 2012, pp. 79-174.
- MSDN, "C# Compiler Errors," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ms228296.aspx>.
168]
- MSDN, "Code Analysis for Managed Code Warnings," 2013. [Online]. Available:

- 169] <http://msdn.microsoft.com/en-us/library/ee1hzekz.aspx>.
- D. Kamstra, "How to write custom static code analysis rules and integrate them into Visual Studio 2010," 26 March 2012. [Online]. Available: <http://blogs.msdn.com/b/codeanalysis/archive/2010/03/26/how-to-write-custom-static-code-analysis-rules-and-integrate-them-into-visual-studio-2010.aspx>.
- MSDN, "F# Development Environment Features," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ee803793.aspx>.
- 171] <http://msdn.microsoft.com/en-us/library/ee803793.aspx>.
- MSDN, "Code Metrics Values," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/bb385914.aspx>.
- 172] <http://msdn.microsoft.com/en-us/library/bb385914.aspx>.
- Z. Naboulsi, "Code Metrics – Maintainability Index," 26 May 2011. [Online]. Available: <http://blogs.msdn.com/b/zainnab/archive/2011/05/26/code-metrics-maintainability-index.aspx>.
- 173] <http://blogs.msdn.com/b/zainnab/archive/2011/05/26/code-metrics-maintainability-index.aspx>.
- Alexander, "Code analysis and metrics in .NET applications," 19 July 2007. [Online]. Available: <http://appdevchronicles.blogspot.co.uk/2007/07/code-analysis-and-metrics-in-net.html>.
- 174] <http://appdevchronicles.blogspot.co.uk/2007/07/code-analysis-and-metrics-in-net.html>.
- V. Sarda, "Analyze Solution For Code Clones," 2 December 2012. [Online]. Available: <http://www.c-sharpcorner.com/UploadFile/d2ee01/analyze-solution-for-code-clones/>.
- 175] <http://www.c-sharpcorner.com/UploadFile/d2ee01/analyze-solution-for-code-clones/>.
- MSDN, "Debug Your App by Recording Code Execution with IntelliTrace," 2012. [Online]. Available: [http://msdn.microsoft.com/en-us/library/dd264915\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(v=vs.110).aspx).
- 176] [http://msdn.microsoft.com/en-us/library/dd264915\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/dd264915(v=vs.110).aspx).
- NetBeans, "NetBeans is Open Source," NetBeans, [Online]. Available: <https://netbeans.org/about/os/>. [Accessed 09 04 2015].
- 177] <https://netbeans.org/about/os/>. [Accessed 09 04 2015].
- MSDN, "Developing Visual Studio Extensions," Microsoft, 2013. [Online]. Available: <https://msdn.microsoft.com/en-us/library/dd885119.aspx>. [Accessed 09 04 2015].
- 178] <https://msdn.microsoft.com/en-us/library/dd885119.aspx>. [Accessed 09 04 2015].
- J. Collins, Interviewee, *WinFPT*. [Interview]. 2012.
- 179]

- Software Validation Ltd, "INLINE," [Online]. Available:
180] http://simconglobal.com/fpt_ref_inline.html.
- Software Validation Ltd, "UNWIND," [Online]. Available:
181] http://simconglobal.com/fpt_ref_unwind.html.
- The MathWorks, Inc., "Polyspace Code Prover," [Online]. Available:
182] <http://www.mathworks.co.uk/products/polyspace-code-prover/>.
- The MathWorks, Inc., "http://www.mathworks.co.uk/products/polyspace-bug-
183] finder/features.html," [Online]. Available:
<http://www.mathworks.co.uk/products/polyspace-bug-finder/features.html>.
- ISO, "ISO 26262-1:2011," ISO, 2011. [Online]. Available:
184] http://www.iso.org/iso/catalogue_detail?csnumber=43464. [Accessed 07 04 2015].
- IEC, "Functional safety - the IEC," 2015. [Online]. Available:
185] http://www.iec.ch/about/brochures/pdf/technology/functional_safety.pdf. [Accessed 07 04 2015].
- CENELEC, *50128 : Railway Applications: Software for Railway Control and Protection Systems*, European Committee for Electrotechnical Standardization, CENELEC, EN50, 1997.
- International Electrotechnical Commission, "Medical device software – Software life cycle
187] processes," *International IEC Standard 62304 First Edition*, 2006.
- C. M. Holloway, "Towards understanding the DO-178C/ED-12C assurance case.," *7th International IET System Safety Conference, Incorporating the Cyber Security Conference*, vol. 15, no. 18, 2012.
- AdaCore, "What is DO-278?," AdaCore, [Online]. Available:
189] <http://www.adacore.com/gnatpro-safety-critical/atm/do-278-overview/>. [Accessed 07 04 2015].

- The MathWorks, Inc., "Simulink," [Online]. Available:
190] <http://www.mathworks.co.uk/products/simulink/>.
- E. Mandrikov, "Plugin Class Loader," 15 September 2013. [Online]. Available:
191] docs.codehaus.org/display/SONAR/Plugin+Class+Loader. [Accessed 2014 September 22].
- D. Racodon, "Multi-language Analysis," 17 April 2014. [Online]. Available:
192] <http://docs.codehaus.org/display/SONAR/Release+4.2+Upgrade+Notes>. [Accessed 22 September 2014].
- SonarCommunity, "sonar-csharp / csharp-checks / src / main / java / com / sonar / csharp / checks," 10 April 2014. [Online]. Available: <https://github.com/SonarCommunity/sonar-csharp/tree/master/csharp-checks/src/main/java/com/sonar/csharp/checks>. [Accessed 22 September 2014].
- I. D. Baxter, "The Design Maintenance System (DMS) A Tool for Automating Software
194] Quality Enhancement," Semantic Designs, Inc., 2001.
- Semantic Designs, Incorporated, "DMS® Software Reengineering Toolkit™," Semantic
195] Designs, Incorporated, [Online]. Available:
<http://www.semdesigns.com/Products/DMS/DMSToolkit.html>. [Accessed 22 September 2014].
- I. D. Baxter, C. Pidgeon and M. Mehlich, "DMS : Program Transformations for Practical
196] Scalable Software Evolution," in *ICSE '04 Proceedings of the 26th International Conference on Software Engineering*, Washington, 2004.
- Semantic Designs, "Programming Language Tools," [Online]. Available:
197] <http://www.semdesigns.com/Products/LanguageTools/>. [Accessed 22 September 2014].
- G. Wielenga, "JDK 7 Support in NetBeans IDE 7.0," 11 October 2012. [Online]. Available:
198] <http://netbeans.dzone.com/news/jdk-7-support-netbeans-ide-70>.
- M. Jean, H. Gregg and R. A. Orso, "Representation and Analysis of Software," 30 May
199] 2012. [Online]. Available: <http://www.ics.uci.edu/~lopes/teaching/inf212W12/readings/rep->

analysis-soft.pdf.

200] L. Copeland, "Chapter 10: Control Flow Testing," in *A Practitioner's Guide to Software Test Design*, Artech House, 2004.

201] P. Jansen, "The TIOBE Quality Indicator," 28 May 2014. [Online]. Available: <http://www.tiobe.com/content/paperinfo/TIOBEQualityIndicator.pdf>.

202] A. H. Watson and T. J. McCabe, "Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric," September 1996. [Online]. Available: <http://www.mccabe.com/pdf/mccabe-nist235r.pdf>.

203] VS, "Cyclomatic Complexity with Example," 12 December 2011. [Online]. Available: <http://testingwarrior.blogspot.co.uk/2011/12/cyclomatic-complexity-with-example.html>.

204] Testwell Oy / Verifysoft Technology GmbH, "Halstead Metrics," 05 June 2010. [Online]. Available: http://www.verifysoft.com/en_halstead_metrics.html.

205] Waterloo Maple Inc., "SoftwareMetrics[HalsteadMetrics]," [Online]. Available: <http://www.maplesoft.com/support/help/Maple/view.aspx?path=SoftwareMetrics/HalsteadMetrics>.

206] Virtual Machinery, "The Halstead metrics," [Online]. Available: <http://www.virtualmachinery.com/sidebar2.htm>.

207] GrammaTech, "Halstead Metrics," [Online]. Available: <http://www.grammatech.com/codesonar/workflow-features/halstead>.

208] A. Abran, "Halstead's Metrics: Analysis of Their Designs," 2010. [Online]. Available: <http://profs.etsmtl.ca/aabran/English/Accueil/ChapersBook/Abran%20-%20Chapter%20007.pdf>.

209] A. Serebrenik, "2IS55 Software," Eindhoven University of Technology, 27 04 2011. [Online]. Available: <http://www.win.tue.nl/~aserebre/2IS55/2010-2011/10.pdf>. [Accessed 09 04 2015].

- 210] D. Kouba , "THE ALGEBRA OF SUMMATION NOTATION," University of California,
21 04 1991. [Online]. Available:
<https://www.math.ucdavis.edu/~kouba/CalcTwoDIRECTORY/summationdirectory/Summation.html>. [Accessed 08 04 2015].
- 211] Loyola Marymount University's Computer Science, "Programming Paradigms," [Online].
Available: <http://cs.lmu.edu/~ray/notes/paradigms/>. [Accessed 22 November 2014].
- 212] R. W. Sebesta, in *Concepts of Programming Languages*, Pearson, 2013, pp. 38-39.
- 213] T. Petricek and J. Skeet, in *Real-World Functional Programming: With Examples in F# and C#*, Manning Publications, 2010, p. 12.
- 214] R. W. Sebesta, in *Concepts of Programming Languages*, Pearson, 2013, p. 754.
- 215] P. Van Roy, "Programming paradigms for dummies: What every programmer should know.," *New computational paradigms for computer music*, no. 104, 2009.
- 216] O. J. Dahl, E. W. Dijkstra and C. A. R. Hoare, *Structured programming.*, Academic Press Ltd, 1972.
- 217] J. Cain, "Lecture 1 | Programming Paradigms (Stanford)," 18 July 2008. [Online].
Available: <https://www.youtube.com/watch?v=Ps8jOj7diA0>. [Accessed 19 1 2015].
- 218] H. M. Deitel, "Python: How to Program," in *Python: How to Program*, Prentice Hall, 2002,
p. 9 & 60.
- 219] S. Kedar and S. Thakare, *Principles of programming languages* 4th Edition, India: Technical Publications, 2009.
- 220] T. Budd, "Understanding Object-Oriented Programming with Java," in *Understanding Object-Oriented Programming with Java*, Pearson Education, 2002, p. 3.

- W. Savitch, "Absolute Java," in *Absolute Java*, Pearson Education, 2012, p. 3.
221]
- T. Budd, "Understanding Object-Oriented Programming with Java," in *Understanding*
222] *Object-Oriented Programming with Java*, Pearson Education, 2002, p. 54.
- J. Skeet and T. Petricek, "Real-World Functional Programming: With Examples in F# and
223] C#," in *Real-World Functional Programming: With Examples in F# and C#*, Manning
Publications, 2010, p. 4.
- K. Nørmarks, "Overview of the four main programming paradigms," 2 July 2013. [Online].
224] Available: [http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-
paradigm-overview-section.html](http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html). [Accessed 26 September 2014].
- P. Hudak and L. Smith, "Para-functional programming: a paradigm for programming
225] multiprocessor systems," *Proceedings of the 13th ACM SIGACT-SIGPLAN symposium on*
Principles of programming languages. ACM, pp. 243-254, 1986.
- MSDN, "Functional Programming vs. Imperative Programming," 2013. [Online]. Available:
226] <http://msdn.microsoft.com/en-GB/library/bb669144.aspx>. [Accessed 22 November 2014].
- E. R. Harold, "Why Functional Programming in Java is Dangerous," 20 January 2013.
227] [Online]. Available: [http://cafe.elharo.com/programming/java-programming/why-
functional-programming-in-java-is-dangerous/](http://cafe.elharo.com/programming/java-programming/why-functional-programming-in-java-is-dangerous/). [Accessed 22 November 2014].
- Oliver, "Functional programming: A step backward," *Java World*, 5 Jul 2012.
228]
- P. Seibel, "They Called It LISP for a Reason: List Processing," in *Practical Common Lisp*,
229] Apress, 2005.
- MIT, "Lists," January 2000. [Online]. Available:
230] https://groups.csail.mit.edu/mac/ftplib/scheme-7.4/doc-html/scheme_8.html. [Accessed 22
November 2014].

MSDN, "Lists (F#)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/dd233224.aspx>. [Accessed 22 Novemeber 2014].

Oracle, "Interface List<E>," [Online]. Available: <https://docs.oracle.com/javase/7/docs/api/java/util/List.html>. [Accessed 22 Novemeber 2014].

Oracle, "Interface Iterator<E>," [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>. [Accessed 22 November 2014].

LispWorks, "CAR, CDR," [Online]. Available: http://clhs.lisp.se/Body/f_car_c.htm. [Accessed 22 Novemeber 2014].

MSDN, "List.map<'T,'U> Function (F#)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/ee370378.aspx>. [Accessed 22 November 2014].

GNU, "7.7 Mapping of Lists," [Online]. Available: <https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Mapping-of-Lists.html>. [Accessed 22 November 2014].

GNU, "7.8 Reduction of Lists," [Online]. Available: <https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Reduction-of-Lists.html>. [Accessed 22 November 2014].

GNU, "7.5 Filtering Lists," [Online]. Available: <https://www.gnu.org/software/mit-scheme/documentation/mit-scheme-ref/Filtering-Lists.html>. [Accessed 22 November 2014].

Oracle, "Interface Stream<T>," [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>. [Accessed 22 November 2014].

D. Obasanjo, "Functional Programming: using Map, Reduce adn Filter," [Online]. Available: <http://www.25hoursaday.com/weblog/2008/06/16/FunctionalProgrammingInC30HowMapR>

educeFilterCanRockYourWorld.aspx. [Accessed 18 October 2014].

MASN, "Enumerable Class," [Online]. Available: <http://msdn.microsoft.com/en-us/library/system.linq.enumerable.aspx>. [Accessed 22 November 2014].

B. Harvey and M. Wright, "Lambda," in *Simply Scheme: Introducing Computer Science 2/e*, MIT Press, 1991.

MSDN, "Lambda Expressions: The fun Keyword (F#)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/dd233201.aspx>. [Accessed 14 November 2014].

MSDN, "Lambda Expressions (C# Programming Guide)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-GB/library/bb397687.aspx>. [Accessed 22 November 2014].

Oracle, "Lambda Expressions," [Online]. Available: <https://docs.oracle.com/javase/tutorial/java/javaOO/lambdaexpressions.html>. [Accessed 22 November 2014].

J. Cain, "Scheme: Functions As Data," 19 May 2008. [Online]. Available: <http://see.stanford.edu/materials/icsppcs107/31-Functions-As-Data.pdf>. [Accessed 22 November 2014].

MSDN, "Functions as First-Class Values (F#)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-us/library/dd233158.aspx>. [Accessed 22 November 2014].

Oracle, "Package java.util.function," [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html>. [Accessed 22 November 2014].

Oracle, "Interface Predicate<T>," [Online]. Available: <https://docs.oracle.com/javase/8/docs/api/java/util/function/Predicate.html>. [Accessed 22 November 2014].

MSDN, "Delegates (C# Programming Guide)," 2013. [Online]. Available: <http://msdn.microsoft.com/en-gb/library/ms173171.aspx>. [Accessed 22 November 2014].

- Oracle, "Method References," [Online]. Available:
251] <https://docs.oracle.com/javase/tutorial/java/javaOO/methodreferences.html>. [Accessed 22 November 2014].
- Oracle, "JDK 8u25 with NetBeans 8.0.1," [Online]. Available:
252] <http://www.oracle.com/technetwork/articles/javase/jdk-netbeans-jsp-142931.html>. [Accessed 22 November 2014].
- T. Zezula and A. Stashkova, "Overview of JDK 8 Support in NetBeans IDE," [Online].
253] Available: <https://netbeans.org/kb/docs/java/javase-jdk8.html#lambda>. [Accessed 22 November 2014].
- Haskell, "The Haskell Programming Language," 9 September 2013. [Online]. Available:
254] <https://www.haskell.org/haskellwiki/Haskell>. [Accessed 22 November 2014].
- Mitchell, Neil;, "HLint," [Online]. Available: <http://community.haskell.org/~ndm/hlint/>.
255] [Accessed 22 November 2014].
- Hackage, "The QuickCheck package," 18 December 2011. [Online]. Available:
256] <http://hackage.haskell.org/package/QuickCheck-2.4.2>. [Accessed 22 November 2014].
- Haskell Community Server, "Haskell style scanner," [Online]. Available:
257] <http://projects.haskell.org/style-scanner/>. [Accessed 22 November 2014].
- EFPL, "Object-Oriented Meets Functional," [Online]. Available: [http://www.scala-](http://www.scala-lang.org/)
258] [lang.org/](http://www.scala-lang.org/). [Accessed 22 November 2014].
- B. McKenna, "Flexible Scala code linting tool," 15 October 2014. [Online]. Available:
259] <https://github.com/typelevel/wartremover>. [Accessed 22 November 2014].
- "Scalastyle - Scala style checker," [Online]. Available: <http://www.scalastyle.org/>.
260] [Accessed 22 November 2014].
- C. Ryder and S. Thompson, "Software Metrics: Measuring Haskell," *UNSPECIFIED*, 2005.
261]

- M. Sherriff, L. Williams and M. Vouk, "Using In-Process Metrics to Predict Defect Density
262] in Haskell Programs," *Fast Abstract, International Symposium on Software Reliability Engineering*, 2004.
- N. Rodrigues and J. Vilaça, "Identifying Clones in Functional Programs for Refactoring," in
263] *ENTERprise Information Systems*, Berlin Heidelberg, Springer , 2010.
- extremeprogramming, "Refactor Mercilessly," [Online]. Available:
264] <http://www.extremeprogramming.org/rules/refactor.html>. [Accessed 22 Sepetember 2014].
- H. Li, C. Reinke and S. Thompson, "Tool support for refactoring functional programs,"
265] *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell.*, pp. 27-38, 2003.
- C. Ryder, "Software Measurement for Functional Programming," University of Kent at
266] Canterbury, 2004.
- MIT, "Datalog User Manual," 2004. [Online]. Available:
267] <http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html>. [Accessed 26 September 2014].
- R. Harper, "What, If Anything, Is A Declarative Language?," 18 July 2013. [Online].
268] Available: <https://existentialtype.wordpress.com/tag/imperative-programming/>. [Accessed 26 September 2014].
- P. Moura, "Logtalk," 19 November 2014. [Online]. Available: <http://logtalk.org/>. [Accessed
269] 22 November 2014].
- GNU, "<http://www.gprolog.org/>," [Online]. Available: <http://www.gprolog.org/>. [Accessed
270] 22 November 2014].
- LPA, "LPA Intelligence Server," 2014. [Online]. Available: <http://www.lpa.co.uk/int.htm>.
271] [Accessed 22 November 2014].
- Declarativa, "InterProlog 2.1.2: a Java front-end and enhancement for Prolog," [Online].
272] Available: <http://www.declarativa.com/interprolog/>. [Accessed 22 November 2014].

- JPL, "A Java Interface to Prolog," 18 March 2003. [Online]. Available: http://www.swi-prolog.org/packages/jpl/java_api/. [Accessed 22 November 2014].
- M. Hanus and F. Zartmann, "Mode analysis of functional logic programs," *Static Analysis*, pp. 26-42, 1994.
- aau, "Overview of the logic paradigm," 2 July 2013. [Online]. Available: http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html#paradigms_logic-paradigm-overview_title_1. [Accessed 22 November 2014].
- Charles Sturt University, "Software Design and Development," [Online]. Available: <http://hsc.csu.edu.au/sdd/options/para/3194/logic.htm>. [Accessed 22 November 2014].
- The MITRE Corporation, "Datalog User Manual," 2004. [Online]. Available: <http://www.ccs.neu.edu/home/ramsdell/tools/datalog/datalog.html>. [Accessed 22 November 2014].
- Trinity College, "Facts, Rules and Queries," [Online]. Available: <http://www.cs.trincoll.edu/~ram/cpsc352/notes/prolog/factsrules.html>. [Accessed 22 November 2014].
- Visual Prolog, "Visual Prolog 7.5 Release," 23 July 2014. [Online]. Available: <http://www.visual-prolog.com/>. [Accessed 22 November 2014].
- A. Aiken and T. Lakshman, Directional type checking of logic programs, Berlin Heidelberg: Springer, 1994.
- P. De Boeck, and B. Le Charlier, "Static type analysis of Prolog procedures for ensuring correctness," *Programming Language Implementation and Logic Programming*, pp. 222-237, 1990.
- M. Hanus and F. Zartmann, "Mode analysis of functional logic programs," *Static Analysis*, pp. 26-42, 1994.

- E. Rohwedder and F. Pfenning, "Mode and termination checking for higher-order logic
283] programs," *Programming Languages and Systems—ESOP'96*, pp. 296-310, 1996.
- SWI-Prolog, "SWI-Prolog IDE --- Execution Profiler," [Online]. Available: <http://www.swi-prolog.org/profile.html>. [Accessed 22 November 2014].
284]
- T. A. Budd, T. P. Justice and R. K. Pandey, "General-purpose multiparadigm programming
285] languages: an enabling technology for constructing complex systems," *Engineering of Complex Computer Systems, 1995. Held jointly with 5th CSESAW, 3rd IEEE RTAW and 20th IFAC/IFIP W RTP, Proceedings., First IEEE International Conference on*, pp. 334-337, 1995.
- W. Al-Ahmad and E. Steegmans, "Java and the object-oriented paradigm: comparison and
286] evaluation," *CW Reports*, vol. 249, no. 12, 1997.
- P. Wegner, "Concepts and paradigms of object-oriented programming," *ACM SIGPLAN
287] OOPS Messenger*, vol. 1, no. 1, pp. 7-87, 1990.
- P. Smolensky, "Connectionist AI, symbolic AI, and the brain," *Artificial Intelligence
288] Review*, vol. 1, no. 2, pp. 95-109, 1987.
- R. Toal, "Programming Paradigms," [Online]. Available:
289] <http://cs.lmu.edu/~ray/notes/paradigms/>. [Accessed 26 September 2014].
- aosd steering committee, "aosd.net," [Online]. Available: <http://aosd.net/>. [Accessed 26
290] September 2014].
- D. Temkin, "PROGRAMMING LANGUAGES," 11 November 2014. [Online]. Available:
291] <http://esoteric.codes/>. [Accessed 26 September 2014].
- L. Hartikka, "Arnold Schwarzenegger based programming language," 24 August 2014.
292] [Online]. Available: <https://github.com/lhartikk/ArnoldC>. [Accessed 26 September 2014].
- D. Pazel, "The Effigy Project—Moving Programming Concepts to a Visual Paradigm,"
293] *Visual End User Workshop at VL2000*, 2000.

- 294] J. Glimming, T. Altenkirch and J. Patrik , "WHAT IS THE NEXT PROGRAMMING PARADIGM?," in *THE SECOND INTERNATIONAL SOFTWARE TECHNOLOGY EXCHANGE WORKSHOP*, SWEDEN, 2012.
- 295] Functor AB, "THE LANGUAGE OF ALL LANGUAGES," 2013. [Online]. Available: <http://www.functor.se/products/scalor/scalor/>. [Accessed 22 November 2014].
- 296] Functor AB, "THE SCALOR™ PLATFORM FROM FUNCTOR," [Online]. Available: <http://www.functor.se/products/scalor/>. [Accessed 22 November 2014].
- 297] Functor AB, "CONSTRUCTIVE PROGRAMMING," [Online]. Available: <http://www.functor.se/products/scalor/constructive-by-example/>. [Accessed 22 November 2014].
- 298] MSDN, "LINQ (Language-Integrated Query)," MSDN, 2015. [Online]. Available: <https://msdn.microsoft.com/en-us/library/bb397926.aspx> .
- 299] C. Mims, "Why CPUs Aren't Getting Any Faster," *MIT Technology Review*, 12 October 2010.
- 300] J. Hruska, "The death of CPU scaling: From one core to many — and why we're still stuck," *Extreme Tech*, 1 February 2012.
- 301] J. Wyngaard, M. Inggs, J. Collins and B. Farrimond, "Towards a many-core architecture for HPC," In *Field Programmable Logic and Applications (FPL)*, 2013 23rd International Conference on, pp. 1-4, 2013.
- 302] ISO, "ISO/IEC 14977 : 1996(E)," 1996. [Online]. Available: <http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf>. [Accessed 17 October 2012].
- 303] M. Might, "The language of languages," [Online]. Available: <http://matt.might.net/articles/grammars-bnf-ebnf/>. [Accessed 22 November 2014].
- 304] S. C. Johnson, "Yacc: Yet Another Compiler-Compiler," [Online]. Available: <http://dinosaur.compilertools.net/yacc/>. [Accessed 28 November 2014].

- E. Berger, "The FORTRAN Automatic Coding System," 2011. [Online]. Available:
 305] <http://people.cs.umass.edu/~emery/classes/cmpsci691st/scribe/lecture2-fortran.pdf>.
 [Accessed 22 November 2014].
- w3schools, "XQuery Tutorial," [Online]. Available: <http://www.w3schools.com/xquery/>.
 306] [Accessed 22 November 2014].
- M. Rouse, "XQL (XML Query Language)," [Online]. Available:
 307] <http://searchsoa.techtarget.com/definition/XQL>. [Accessed 22 November 2014].
- J. Zemerick, "The following slides are based on the work presented in the MS CS thesis
 308] "Profiling, Extracting, and Analyzing Dynamic Software Metrics" of Jeffrey Zemerick,"
 2009. [Online]. Available: <http://www.csee.wvu.edu/~katerina/Teaching/CS-736-Fall-2010/CS-736-Program-Profiling.pdf>.
- D. A. Watt and O. L. Madsen, "Extended Attribute Grammars," *The Computer Journal*, vol.
 309] 26, no. 2, pp. 142-153, 1983.
- TSWP, "Living Glossary," [Online]. Available:
 310] http://www.testingstandards.co.uk/living_glossary.htm#Testing. [Accessed 11 12 2012].
- P. Smacchia, "Metrics Definitions," [Online]. Available: <http://ndepend.com/Metrics.aspx>.
 311]
- F. Pfenning and A. Platzer, "Lecture Notes on Liveness Analysis," [Online]. Available:
 312] <http://symbolaris.com/course/Compilers12/04-liveness.pdf>.
- R. Patton, Software Testing, 2nd Edition ed., N. Rowe, S. Qiu, C. Clapp and G. Nedeff,
 313] Eds., Indianapolis, Ind.: Sams Publishing, 2006.
- D. Owens and M. Anderson, "A generic framework for automated Quality Assurance of
 314] software models-Application of an Abstract Syntax Tree," *Science and Information Conference (SAI)*, 2013, pp. 207 - 211, 2013.
- T. Murnane, K. Reed, D. Grant and T. Chen, "A Preliminary Survey on Software

- 315] Testing Practices in Australia," *Software Engineering Conference, 2004. Proceedings. 2004 Australian*, pp. 116-125, 13-16 April 2004.
- A. Moller and M. I. Schwartzbach, "Static Program Analysis," 8 February 2012. [Online].
- 316] Available: <http://cs.au.dk/~mis/static.pdf>.
- Microsoft, "Windows," [Online]. Available: [http://windows.microsoft.com/en-](http://windows.microsoft.com/en-gb/windows/home)
- 317] [gb/windows/home](http://windows.microsoft.com/en-gb/windows/home).
- J.-B. Michel, Y. K. Shen, A. P. Aiden, A. Veres, M. K. Gray, W. Brockman, T. G. B. Team,
- 318] J. P. Pickett, D. Hoiberg, D. Clancy, P. Norvig, J. Orwant, S. Pinker, M. A. Nowak and E. L. Aiden, "Quantitative Analysis of Culture Using Millions of Digitized Books," Google, 16 12 2010. [Online]. Available: http://books.google.com/ngrams/graph?content=software+testing%2Csoftware+quality&year_start=1900&year_end=2008&corpus=0&smoothing=6. [Accessed 11 10 2012].
- K. A. Mayo, S. A. Wake and M. S. Henry, "Static and Dynamic Software Quality Metric
- 319] Tools," 1990.
- L. Lou, "Software Testing Techniques: Technology Maturation and Research Strategy,"
- 320] Class Report, 2001.
- ISO, "Appendix C ISO 9126 Metrics," 1996. [Online]. Available:
- 321] http://www.rockynook.com/samples/97/ISO_9126_Metrics.pdf. [Accessed 23 November 2012].
- IBM, "IBM Rational AppScan: Application security and risk managment," November 2011.
- 322] [Online]. Available: http://www.sebyde.nl/uploads/media/IBM_Rational_Appscan_family_Data_Sheet.pdf. [Accessed 09 11 2012].
- R. Harrison, "Comparing programming paradigms: an evaluation of functional and object-
- 323] oriented programs," *Software Engineering Journal*, vol. 11, no. 4, pp. 247-254, 1996.
- M. Harman and R. M. Hierons, "An Overview of Program Slicing," [Online]. Available:

- 324] <http://www0.cs.ucl.ac.uk/staff/mharman/sf.html>.
- M. Hanus, "Distributed Programming in a Multi-Paradigm Declarative Language,"
- 325] *Principles and Practice of Declarative Programming*, vol. 1702, pp. 188-205, 29 September 1999.
- D. R. Hanson, "lcc.NET: Targeting the .NET Common Intermediate Language from
- 326] Standard C," *Software: Practice and Experience*, vol. 34, no. 3, p. 265–286, 05 January 2004.
- . L. Gupta, "Inversion of control (IoC) and dependency injection (DI) patterns in spring
- 327] framework and related interview questions," 19 March 2013. [Online]. Available: <http://howtodoinjava.com/2013/03/19/inversion-of-control-ioc-and-dependency-injection-di-patterns-in-spring-framework-and-related-interview-questions/>.
- L. M. Garshol, "BNF and EBNF: What are they and how do they work?," 3 March 2005.
- 328] [Online]. Available: <http://www.garshol.priv.no/download/text/bnf.html>. [Accessed 17 October 2012].
- R. W. Floyd, "The paradigms of programming," *Communications of the ACM*, vol. 22, no.
- 329] 8, pp. 455-460, 1979.
- P. Ferrara, "Static Type Analysis of Pattern Matching by Abstract Interpretation," *Formal*
- 330] *Techniques for Distributed Systems*, pp. 186-200, 2010.
- B. Edupuganty and B. Bryant, "Two-level Grammar as a Functional Programming
- 331] Language," *The Computer Journal*, vol. 32, no. 1, pp. 36-44, 1989.
- G. Deltombe and O. L. & B. F. Goer, "Bridging KDM and ASTM for Model-Driven
- 332] Software Modernization," *SEKE*, pp. 517-524, 2012.
- B. Cui, J. Li, T. Guo, J. X. Wang and D. Ma, "Code Comparison System based on Abstract
- 333] Syntax Tree," 2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT), 2010.

- 334] J. Collins, B. Farrimond, M. Anderson, D. Owens and D. Bayliss, "Automated Quality Assurance Analysis: WRF–A Case Study," *Journal of Software*, vol. 8, no. 9, pp. 2177-2184, 2013.
- 335] J. K. Chhabra and V. Gupta, "A survey of dynamic software metrics.," *Journal of computer science and technology*, vol. 25, no. 5, pp. 1016-1029, 2010.
- 336] J. R. Cary, S. G. Shasharina, J. C. Cummings, J. V. Reynders and P. J. Hinker, "Comparison of C++ and Fortran90 for object-oriented scientific programming," *Computer Physics Communications*, vol. 105, no. 1, pp. 20 - 36, September 1997.
- 337] C. Britton and J. Doake, "Testing and Handing Over the System," in *Software System Development : A Gentle Introduction*, 4th Edition ed., K. Reade, K. Mosman, A. Duijser and J. Bishop, Eds., Maidenhead, Birkshire: MCGraw-Hill Education, 2006, pp. 175-180.
- 338] E. Bolshakova, "PROGRAMMING PARADIGMS IN COMPUTER SCIENCE EDUCATION," *International Journal "Information Theories & Applications"*, vol. 12, pp. 285-290.
- 339] J. E. Bentley, "Software testing fundamentals–concepts, roles, and terminology," in *Proceedings of SAS Conference*, 2005.
- 340] A. Ambler, M. Burnett and B. Zimmerman, "Browse Journals & Magazines > Computer ...> Volume:25 Issue:9 [Online]. Available: http://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/execution-monitor/user-guide.html.
- 341] MSDN, "Using the Profiler Tool to analyze the performance of your code," 17 July 2013. [Online]. Available: [http://msdn.microsoft.com/en-us/library/ie/gg699341\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/ie/gg699341(v=vs.85).aspx).
- 342] Canonical, [Online]. Available: <http://www.ubuntu.com/>.
- 343] Boot Test, "The Execution Monitor user's guide," [Online]. Available: http://www.boost.org/doc/libs/1_45_0/libs/test/doc/html/execution-monitor/user-guide.html.

- McCabe Software, "Software Metrics Glossary," [Online]. Available:
344] http://www.mccabe.com/iq_research_metrics.htm.
- MathWorks, "Polyspace Bug Finder - Key Features," [Online]. Available:
345] www.mathworks.co.uk/products/polyspace-bug-finder/features.html. [Accessed 19
September 2014].
- Aivosto Oy, "Lines of code metrics (LOC)," [Online]. Available:
346] <http://www.aivosto.com/project/help/pm-loc.html>.
- Bourns College of Engineering, "Lecture 3 Data Flow Analysis," 3 March 2009. [Online].
347] Available: <http://www.cs.ucr.edu/~gupta/teaching/201-09/My3.pdf>.
- Penn Engineering - Computer and Information Science, "Introduction to Data-flow
348] Analysis," [Online]. Available: <http://www.cis.upenn.edu/~cis570/slides/lecture04.pdf>.
- SimCon, "http://www.simconglobal.com/fpt_ref_index.html," [Online]. Available:
349] http://www.simconglobal.com/fpt_ref_index.html. [Accessed 19 September 2014].
- Free Software Foundation, Inc., "GCC, the GNU Compiler Collection," 14 August 2014.
350] [Online]. Available: <https://gcc.gnu.org/>.
- IBM, "Execution trace," [Online]. Available:
351] http://pic.dhe.ibm.com/infocenter/brdotnet/v7r1/index.jsp?topic=%2Fcom.ibm.websphere.ilog.brdotnet.doc%2FContent%2FBusiness_Rules%2FDocumentation%2Fpubskel%2FRules_for_DotNET%2Fps_RFDN_Global385.html.
- The Computer Language Company Inc., "Definition of:Java Virtual Machine," [Online].
352] Available: <http://www.pcmag.com/encyclopedia/term/45578/java-virtual-machine>.
- MSDN, "Definite Assignment," [Online]. Available: [http://msdn.microsoft.com/en-us/library/aa277915\(v=vs.60\).aspx](http://msdn.microsoft.com/en-us/library/aa277915(v=vs.60).aspx).
353]
- Sourceforge.net, "Current Rulesets," 11 August 2013. [Online]. Available:
354] <http://pmd.sourceforge.net/pmd-5.0.5/rules/index.html>. [Accessed 2014c September 19].

- Nullstone Corporation, "Constant Folding," [Online]. Available:
355] http://www.compileroptimizations.com/category/constant_folding.htm.
- Nullstone Corporation, "Compiler Optimizations," [Online]. Available:
356] http://www.compileroptimizations.com/category/dead_code_elimination.htm.
- Nullstone Corporation, "Compiler Optimizations," [Online]. Available:
357] http://www.compileroptimizations.com/category/constant_propagation.htm.
- Microsoft, "Common Language Runtime (CLR)," [Online].
358]
- GrammaTech, "CODESONAR®," [Online]. Available:
359] <http://www.grammatech.com/codesonar/metrics>.
- Computer Science University of Meryland, "CMSC 631 — Program Analysis and
360] Understanding," 2003. [Online]. Available:
<http://www.cs.umd.edu/class/fall2003/cmsc631/lectures/l02.pdf>.
- Sourceforge, "Available Checks," 03 February 2014. [Online]. Available:
361] <http://checkstyle.sourceforge.net/availablechecks.html>. [Accessed 19 September 2014].
- Merriam-Webster, "Automation," [Online]. Available: [http://www.merriam-](http://www.merriam-webster.com/dictionary/automation)
362] [webster.com/dictionary/automation](http://www.merriam-webster.com/dictionary/automation).
- Realsearch, "An Introduction to Object-Oriented Metrics," [Online]. Available:
363] <http://agile.csc.ncsu.edu/SEMaterials/OOMetrics.htm>.

Chapter 9. Glossary

AST	-	Abstract Syntax Tree
ASTM	-	Abstract Syntax Tree Meta-Model
BNF	-	Backus–Naur Form
CFG	-	Control Flow Graph
CIL	-	Common Intermediate Language
CLR	-	Common Language Runtime
EHU	-	Edge Hill University
EBNF	-	Extended Backus–Naur Form
GASTM	-	Generic Abstract Syntax Tree Meta-Model
GCC	-	GNU Compiler Collection
IDE	-	Integrated Development Environment
IR	-	Internal Representation
LIQA	-	Language Independent Quality Assurance
MSIL	-	Microsoft Intermediate Language
OMG	-	Object Management Group
OS	-	Operating System
QA	-	Quality Assurance
QACC	-	Quality Assurance in Climate Code
RCP	-	Rich Client Platform
REF	-	Research Excellence Framework
SDK	-	Software Development Kit

SSCLI	-	Shared Source Common Languages Infrastructure
ST	-	Software Testing
UCAR	-	University Corporation for Atmospheric Research
VM	-	Virtual Machine
WRF	-	Weather Research & Forecasting Model
WSH	-	Windows Script Host
XML	-	Extensible Markup Language