



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

Investigating the Potential of a Codebase Dependency Visualisation System for Implementation at the Driver and Vehicle Licensing Agency

by

Salih Osman

Supervisor: Nicola Whitehead

Second Marker: Kemi Ademoye

Project submitted as part of the requirements for the
award of

MSc (Hons) Software Engineering (Web Development)

January 2020

DECLARATION

I, Salih Osman, declare that I am the sole author of this Project; that all references cited have been consulted; that I have conducted all work of which this is a record, and that the finished work lies within the prescribed word limits.

This work has not previously been accepted as part of any other degree submission.

Signed : Salih Osman

Date : 25/01/2020

FORM OF CONSENT

I, Salih Osman, hereby consent that my Project, submitted in candidature for the M.Sc.(Hons) Software Engineering (Web Development) degree, if successful, may be made available for inter-library loan or photocopying (subject to the law of copyright), and that the title and abstract may be made available to outside organisations.

Signed : Salih Osman

Date : 25/01/2020

Abstract

A prototype software system for visualising the dependency structure of java codebases was developed. This system is capable of extracting the complete set of inter-package dependencies from a given memory location and automatically generating both graphical and matrix-based displays of the dependency network at different levels of detail. The development of this system was guided by in-depths research of the theory of dependency visualisation, as well as the existing technologies which provide inspiration and context for this system. The ultimate motivation for developing this system was to assess its potential for implementation at the Driver and Vehicle Licensing Agency of the British Government (DVLA). The potential functionality of the system was assessed through the use of quantitative and qualitative questionnaires as well as recorded interviews with 8 employees at the DVLA. The system usability scale score from this evaluation was 77.2, increasing to 95 after the system was updated and then retested with 2 of the initial participants. This, in combination with the qualitative and quantitative data collected via a system-specific questionnaire, provided very strong evidence for the usability, effectiveness and potential of the developed system, for implementation at the DVLA and potentially in a more public context.

Acknowledgements

The number of people who've helped me more than I deserve in my life is beyond counting, what follows is simply a sample, if I've forgotten you, please, you should already know I'm forgetful.

First of all, I must thank my MSc project supervisor Dr Nik Whitehead. Without her guidance, encouragement and unending sea of patience this work would not be complete (or possible in general!). Nik, I feel privileged to have had you as a supervisor, thank you for helping me get through the hard times. I hope many more students are blessed with your wise and enthusiastic support

I cannot go further without expressing my sincere thanks, appreciation and love towards Uwais Ashraf, who has been simply a brother to me. It is his example which has inspired me to get into code and enjoy it, even from the early days in Imperial. Uwi, thank you for spending so many hours of your time on the phone, patiently talking me through python etc, you're the real branch tracker.

To all the friends who helped me proof-read this thesis, thank you. Mosa, you actually volunteered for this, you're a natural bootstrapper bro, I hope this acknowledgement counts for something when you make it. Jam, there are few people who I relate to so much, I hope that our brotherly empathy hasn't caused you to ignore my mistakes and that you've kept me straight here as you do in life. Daniel, thanks for putting in the time and for telling me to use Grammarly, don't think this makes us even from the meme project though.

To all my colleagues at the DVLA who helped me out by taking part in surveys etc, you know who you are, please also know that you're appreciated, it can't have been easy pretending to be interested in my strange schemes to graph code. Also, to my line manager, who let me borrow his workers and also tried to help me manage my time, thank you.

I must take a moment to appreciate my dear cousin Qaasim Ali, who has in many ways been closer to me than a blood-brother could be. I know you probably hate being acknowledged like this, but it must be done, you've been a childhood playmate and an adulthood inspiration. I have remained afloat thanks to our little side projects. May Allah enable us to have good, to be good and to do good, together now and in the hereafter.

To my dear parents, Mustafa Farah Osman and Shahbaz Begum Osman. You have been two lights which have remained steady and shining on me my entire life. I love you more than I love my own soul, please forgive me for the times when I seem to forget this. I only ask God to give me the strength to be a person who you are proud to call your son, because if I have achieved that I will have achieved most of the important things in life. Abo and Umijaan, each of you has, in your own way, taught me how to think and how to dream, perhaps the primary gifts of man and on top of that you blessed me with another light in my beloved sister Faatima. As my own words have failed me, I turn to the words of our Lord in [17:24].

More than any person or group, however, all praise and thanks are due to Almighty God, the Ever-Living and the Self-Subsisting, Allah, who brought me out of the womb of my mother not knowing a thing and made for me hearing and vision and intellect that perhaps I might be grateful. My Lord, glory be to you, who have brought me from darkness into light, increase me in sincere gratitude to you. Oh Lord of the mighty throne, without your guidance I am truly lost, keep me upon your path and bring me closer to you. Oh, Most Compassionate, Most Merciful, without your kindness I would have I would have long since fallen, increase my love for you and make me a person worthy of your love.

الْحَمْدُ لِلَّهِ الَّذِي بِنِعْمَتِهِ تَتِمُّ الصَّالِحَاتُ

Table of Contents

1	Introduction.....	10
2	Literature Review	12
2.1	Overview	12
2.1.1	Code Analysis	12
2.1.2	Code Quality.....	13
2.1.3	Information Visualisation.....	16
2.2	Dependency Extraction Engine	19
2.2.1	Forms of Dependency	19
2.2.2	Accessing the Structure of a Codebase.....	21
2.3	Dependency Analysis	25
2.3.1	Dependency Structure Matrix.....	25
2.3.2	Dependency Measurement and Pattern Identification	26
2.4	Displaying Dependencies	27
2.4.1	Graphs.....	27
2.4.2	Display Tools	28
2.5	Existing Technologies	31
2.5.1	Manual Systems	31
2.5.2	Automated Systems	33
2.6	Conclusion	36
3	Methodology.....	38
3.1	Research Methodology	38
3.1.1	Philosophy.....	38
3.1.2	Strategy	38
3.1.3	Approaches	39
3.1.4	Risks & Limitations	39
3.1.5	Validity, Reliability & Generalisability	40
3.1.6	Data Evaluation Methods.....	40
3.2	Development Methodology	41
3.2.1	Software Development Methodology OR Other Relevant Major Methodology	41
3.3	Resources	41
3.3.1	Hardware Resources	41
3.3.2	Software Resources	41
3.3.3	Information Resources	42
3.3.4	Human Resources	42
4	Design and Implementation.....	43
4.1	MoSCoW Requirements	43
4.1.1	Functional Requirements.....	43
4.1.2	Non-functional requirements	44
4.2	Time box 1	46
4.2.1	Features	46
4.2.2	Design and Implementation.....	47
4.2.3	Testing.....	53

4.2.4	Evaluation	53
4.2.5	Requirements Addressed.....	53
4.3	Timebox 2	54
4.3.1	Features	54
4.3.2	Design and Implementation.....	54
4.3.3	Testing.....	58
4.3.4	Evaluation	58
4.3.5	Requirements Addressed.....	59
4.4	Timebox 3	60
4.4.1	Features	60
4.4.2	Design and Implementation.....	60
4.4.3	Testing.....	64
4.4.4	Evaluation	65
4.4.5	Requirements Addressed.....	66
4.5	Timebox 4	67
4.5.1	Features	67
4.5.2	Design and Implementation.....	67
4.5.3	Testing.....	71
4.5.4	Evaluation	71
4.5.5	Requirements Addressed.....	71
4.6	Timebox 5	72
4.6.1	Features	72
4.6.2	Design and Implementation.....	72
4.6.3	Testing.....	76
4.6.4	Evaluation	76
4.6.5	Requirements Addressed.....	77
4.7	Timebox 6	78
4.7.1	Features	78
4.7.2	Design and Implementation.....	78
4.7.3	Findings	79
4.7.4	Summary	89
4.7.5	New requirements to be addressed in the following timebox	90
4.8	Timebox 7	91
4.8.1	Features	91
4.8.2	Design and Implementation.....	91
	Testing.....	95
4.8.3	Evaluation	95
4.8.4	Requirements Addressed.....	95
4.9	Timebox 8	96
4.9.1	Features	96
4.9.2	Findings	96
4.9.3	Summary	98
5	Conclusions and Recommendations	100
5.1	Conclusions.....	100
5.2	Further Work	102
6	Reflections.....	103

7	Works cited	104
8	Appendices	110
8.1	Appendix A – User Task List.....	110
8.2	Appendix B – User Questionnaire.....	112
8.3	Appendix C – System Usability Scale (SUS)	114

Table of Figures

Figure 1:	Functional cohesion encompasses all benefits from lower levels of cohesion.....	13
Figure 2:	the delineated processes are both communicationally cohesive.....	15
Figure 3:	External connections represent coupling, internal connections represent cohesion [17] ...	16
Figure 4:	Beck’s abstraction (right) greatly simplified the pre-1931 tube map (left) [31]	17
Figure 5:	The original graph presented in an attempt to demonstrate O-ring failure.....	18
Figure 6:	Abstract Syntax Tree (right) for simplified Java class and recursive function (left) [41].....	22
Figure 7:	Sample screenshot of PSI structure viewer (right) for Java code in IntelliJ (left).....	23
Figure 8:	Simple undirected graph G complete with formal notation indicating components of G....	27
Figure 9:	Example graphic representing the Unix Operating system created via graphviz [75]	29
Figure 10:	Sample screenshot of class diagram for airline system produced using UMLet [78]	29
Figure 11:	Level 3 C4 component diagram (left) vs level 4 UML class diagram (right) of 1 component of same system [86].....	32
Figure 12:	Simple NDepend DSM with cyclic dependency highlighting.....	33
Figure 13:	Snapshot of Sourcetrail display, showing graph and code views as well as search functionality [90]	34
Figure 14:	Layered graph view (left) and separate cycle viewer (right) of Sonargraph Explorer [91] .	35
Figure 15:	CodeMR Edge node dependency graph (right) illustrating cohesion and coupling between modules within an application along with visual class statistics key (left) [93]	35
Figure 16 -	Use case diagram for extracting dependencies	47
Figure 17:	Sequence diagram for dependency extraction	48
Figure 18:	Class Diagram for dependency extraction	48
Figure 19:	logic interface as represented within source code during timebox 1.....	49
Figure 20:	Representative screenshot of the console interface with input and output.....	50
Figure 21:	Example of dependency extraction formatted in the DOT language	51
Figure 22:	Package dependency graph of the package dependency extraction system created by itself	52
Figure 23:	Use case diagram for displaying dependency strength irrespective of package size	55
Figure 24:	UML class diagram for displaying dependency strength irrespective of package size	55
Figure 25:	View of dependency matrix as output by system (top line indicates order of packages) ...	57
Figure 26:	Package dependency graph of this system with weighted edges and no java library dependencies.....	57
Figure 27:	Use case diagram for automatic, simultaneous matrix and graph display	61
Figure 28:	Sequence diagram for automatic, simultaneous matrix and graph display.....	61
Figure 29:	UML class diagram for automatic simultaneous display of complete matrix and graph....	62
Figure 30:	Comparing generated graphs for deepest layer of package detail vs with one layer of package detail stripped away	63
Figure 31:	source code for writing to .dot file and then producing a pdf of that file using graphviz ..	63

Figure 32: The updated View.ForLogic interface, indicating changes to the DSM creation process ...	64
Figure 33: Use case diagram for max package depth slicing and heatmap GUI DSM display.....	68
Figure 34: UML class diagram for automatic simultaneous display of complete matrix and graph....	68
Figure 35: Colour coded DSM in heatmap form, automatically produced via Java GUI	69
Figure 36: Segment of the DSM produced for large DVLA codebase	69
Figure 37: Dependency structure for system codebase displayed without slicing (left) vs with max package depth of 2 (right)	70
Figure 38: Use case for generating system outputs in a more visually consumable fashion.....	73
Figure 39: UML class diagram for generating system outputs in a more visually consumable fashion	73
Figure 40: Representing command line entry of visualisation parameters	74
Figure 41: Internal dependencies of free open source peer evaluation tool 'Teammates' graphed to max package depth of 3 layers, showing colour graduations for connectivity.....	75
Figure 42: Section of DSM for free open source peer evaluation tool 'Teammates' graphed to max package depth of 3 layers, showing colour graduations for dependency strength.....	75
Figure 43: DSM for this system, with tooltip demonstrated	76
Figure 44: Graph illustrating quantitative data for system specific questionnaire responses of participant 1.....	80
Figure 45: Graph illustrating quantitative data for system specific questionnaire responses of participant 2.....	81
Figure 46: Graph illustrating quantitative data for system specific questionnaire responses of participant 3.....	83
Figure 47: Graph illustrating quantitative data for system specific questionnaire responses of participant 4.....	84
Figure 48: Graph illustrating quantitative data for system specific questionnaire responses of participant 5.....	85
Figure 49: Graph illustrating quantitative data for system specific questionnaire responses of participant 6.....	86
Figure 50: Graph illustrating quantitative data for system specific questionnaire responses of participant 7.....	87
Figure 51: Graph illustrating quantitative data for system specific questionnaire responses of participant 8.....	88
Figure 52: Graph illustrating average values of quantitative data for system specific questionnaire.	89
Figure 53: Use case diagram for displaying key, trimmed package names and colour contrasted external packages	92
Figure 54: UML class diagram for displaying key, trimmed package names and colour contrasted external packages	92
Figure 55: Key/Legend displayed to illustrate key aspects of the dependency graph format.....	93
Figure 56: Dependency graph for visualisation system, showing trimmed package names and blue external package nodes	94
Figure 57: Code used to remove prefixes shared by all packages from package set.....	94
Figure 58: Graph illustrating quantitative data for repeat system specific questionnaire with participant 1.....	97
Figure 59: Graph illustrating quantitative data for repeat system specific questionnaire of participant 3	98
Figure 60: Graph illustrating average quantitative data for the second testing timebox.....	99
Figure 61: Graph illustrating average quantitative data for the second testing timebox.....	101

Table of Tables

Table 1: Summarised dependency descriptions	19
Table 2: Summarised description of extraction technique options	21
Table 3 - Functional Must Requirements.....	43
Table 4 - Functional Should Requirements	44
Table 5 - Functional Could Requirements.....	44
Table 6 - Functional Won't Requirements	44
Table 7 - Non-Functional Must Requirements	44
Table 8 - Non-Functional Should Requirements	44
Table 9 - Non-Functional Could Requirements.....	44
Table 10: Use case description for extraction of package dependencies	47
Table 11: Timebox 1 test sheet.....	53
Table 12: Use case description for displaying dependency strengths	54
Table 13: Timebox 2 test sheet.....	58
Table 14: Use case description for simultaneously viewing graph and matrix view of sliceable package structure	60
Table 15: Timebox 3 test sheet.....	65
Table 16: Use case description for allowing max package depth slicing and heatmap GUI DSM.....	67
Table 17: Timebox 4 test sheet.....	71
Table 18: Use case description for rendering system output more visually consumable	72
Table 17: Timebox 4 test sheet.....	76
Table 16: Use case description for key, package name trimming and colour change	91
Table 21: Timebox 7 test sheet.....	95

1 Introduction

The original motivation for this project has its source in the early experiences of the researcher whilst integrating into the DVLA work environment as a novice software engineer. The compound effect of being new to the field of development in general, whilst also being unfamiliar with the large existing codebases at the DVLA was somewhat overwhelming. The ability to pick up user stories and begin work on new software features was inhibited by a lack of understanding of the ways in which the different elements of the codebase interacted and how a new piece of code would fit into the existing body. The only established method there seemed to be for increasing familiarity and comprehension of the structure of the codebase was direct inspection of the source code. Although there is much merit in getting an unfiltered view of the code, this technique is slow in building up meaningful understanding of the system as a whole. Furthermore, for new developers, or for non-programming employees, code inspection is certainly not an intuitive or reliable method of increasing comprehension, it requires a significant degree of pre-existing knowledge before one can even begin to effectively interpret the source code. Other than this, some extremely high-level system diagrams were in place, but they lacked a direct connection to the code itself, there seemed to be something of a disconnect between the architectural perspective and the development perspective.

This research project, then, seeks to address this issue, by coming up with a more intuitive solution to aid with source code comprehension. It aims to develop a prototype codebase visualisation system and determine its potential for implementation within the DVLA as a more developed static analysis package. It is intended that the prototype system will be capable of extracting the basic dependency structure from a codebase for subsequent display.

The objectives of this project include analysing and determining best practise techniques for the implementation of a codebase visualisation system written in java via mind-mapped research and a literature review, with focus on relevant existing technologies within the broader field of static analysis tools. This is addressed in section 2 of this report, the literature review.

This will be followed by the design and implementation of a software engine suitable for discerning the method and order in which different packages (and, with further development, classes) within a java codebase invoke, and are invoked within, each other. This may be done using parsing tools for the java language in combination with the autogenerated files such as the program structure interface from the IntelliJ integrated development environment. The configuration used to represent the derived dependencies will be stored via a suitable persistence structure. The Agile Feature Driven Development (FDD) methodology will be used in this process, the choice of different methodologies in the execution of this project is justified in section 3 of this report. The software implementation cycle has been split into timeboxes, which are one of the recommended processes of FDD. These timeboxes are addressed in section 4: Design and Implementation.

A further objective will be to similarly develop a modularised visualisation system capable of displaying source code dependency data in an understandable format. This is intended to take the form of a directed graph system, with each node representing a module, and each edge representing a module invocation. Initially, online tools will be used to verify the effectivity of the engine. Subsequently, a pipeline will be constructed to allow auto-generation of diagrams as a direct result of the dependency parsing operation. In addition to this, the benefit of utilising inbuilt java graphing libraries will be considered.

There will then be exploration of the capability of the system to produce further alternatives to the graphical representation of the source-code structure, via the inclusion of such features as a dependency structure matrix. The possibility of identifying cyclic relationships and representing the cohesion and coupling characteristics of the codebase in a more quantifiable manner. All development pertaining to the aforementioned extraction engine, dependency graph and dependency structure matrix are addressed in timeboxes 1-5 of this report, found between section 4.2 and section 4.6, inclusive. Timebox 7 in section 4.8, represents a final development timebox, implemented after having received user feedback.

The final objective of the project will be to determine the potential of the prototype system in facilitating the software development process via practical research. This will involve the participation of DVLA employees in a series of interviews and/or questionnaires centred around the visualisation system. It will be important to include input from the perspectives of a range of different roles, such as: developer, business analyst, agile delivery manager, etc. The process of gathering this data is illustrated in timebox 6 and timebox 8, sections 4.7 and 4.9 respectively, which represent the first testing period and the second retest with a small sample of original participants.

Ultimately, if this research project is successful, it will provide an answer to the question of whether or not it is realistic to suggest the implementation of a codebase dependency visualisation such as the one developed during this project, within the DVLA. The primary reason for this not having been addressed to date is most likely lack of formal acknowledgement of the issue, as well as lack of time allocated to research and solve it. This MSc project provided an ideal opportunity to build up the required background knowledge, develop a practical solution and then perform real world tests of its effectiveness. The possible advantages of the successful execution of the project may include, the adoption of new software development tools by the DVLA, an increase in code quality and understanding of codebase structure amongst the development community and a means of bridging the divide between established software developers and new inductees or less software-oriented colleagues. The project developed to address the above motivations is presented in the following sections of this report. A solid background of research and relevant information is given, before proceeding to the process of executing the project, overall findings and their ultimate analysis.

2 Literature Review

2.1 Overview

This section seeks to provide a conceptual overview of some of the topics which are deemed fundamental to this project, as well as introducing some of the key terminology and relevant background. These topics relate more to the long-term motivations of the project, i.e. taking a step towards a visual static analysis tool to improve code quality. Topics more relevant to specific details of design, implementation, etc., rather than the ultimate goals, are covered in subsequent sections of the review.

2.1.1 Code Analysis

It has long been established that code analysis, whether manually or through the use of automated tools, is a key factor in developing reliable code, increasing quality and reducing the number of defects [1]. The longer a potential fault or system weakness takes to be detected, the more negative its overall impact on system performance, with increasingly more effort, time and financial expenditure required to amend the issue [2]. It is for this reason that the implementation of effective code analysis techniques, such as the prototype represented by this model, is so fundamental within the software engineering industry.

2.1.1.1 Human Analysis

An essential part of the code analysis process relies on manual, or human analysis. This often takes the form of code review, in which members of the development team visually check over the work of other members in order to increase quality assurance [3]. This process is extremely valuable and important, as it helps prevent, for example, poor quality or defective code from being merged into the main code branch for release.

In addition to catching defects and identifying bad code smells (basically an experienced programmer's intuitive feeling that something is wrong) by simply observing the code structure, for example identifying unreasonable invocation clauses or a cyclic dependency between two modules, code review yields other important benefits. In particular, code review: acts as a medium for education and improvement of more junior developers; improves coordination of the development squad and facilitates "outside the box" thinking with respect to problem solving [4].

However, there are, of course, limitations on what can be achieved by even an extremely proficient developer during manual code analysis. It stands to reason that the larger a codebase, and the larger the pieces of code needing to be reviewed, the more likely it will be that defects or substandard code will slip past a human reviewer. This is backed up by research which indicates that decreased "scan time" during visual review leads to a less effective quality assurance process [5]. It is these limitations on human effectivity in identifying patterns and anti-patterns within large and often poorly documented codebases, that provided the original motivation for this project, as well as being more generally responsible for the use of static analysis tools in general.

2.1.1.2 Static Analysis

Static analysis refers to the process of examination and evaluation of a codebase pre-runtime, i.e. without executing the code, through the use of automated tooling which seeks to identify flawed code via approximation of program behaviour [6]. Static analysis tools have been put to good use, particularly in large codebases with millions of lines of source code, such as the Facebook codebase, where dependable error detection via human analysis and manual code inspection becomes far more

challenging [7]. The “existing technologies” section of this report will address examples of contemporary static analysis tools in further detail.

The range of static analysis tools available to the average development squad is immense [8]. They provide such functionality as: identifying security vulnerabilities; finding potential memory leaks; highlighting code smells and problematic dependency patterns, as explored within this project.

However, despite this range, there is significant evidence that static analysis tools are underused by developers in general, as suggested by Johnson *et al* [9]. They go on to conclude that a major reason for this is the lack of intuitive presentation options offered by conventional static analysis tooling. This puts forward a strong case for the focus on information presentation and visualisation when identifying possible defects or poor-quality code via static analysis.

2.1.2 Code Quality

The concept of code quality is intrinsic to the aims and objectives of the project. As discussed, all static analysis techniques seek to deliver the same ultimate result: “good” code, that is code which is, among other things, efficient, understandable and maintainable [10]. When developing a static analysis tool then, it is essential to obtain an understanding of the mechanics by which code is endowed with these qualities.

2.1.2.1 Cohesion

The phrase “cohesion”, as it pertains to software development, was brought into being by Constantine [11] during the 1960s, as a direct result of his founding work in the field of “Structured Design”, which sought to relate design methods more closely with, and make them more representative of, the actual necessities of the system being developed. The expression was later given a formal definition in Constantine’s 1970’s work “Structured Design”, which specifies that cohesion is indicative of the level of interrelation or connectedness between the constituent elements of a class or package, i.e. how tightly or loosely “bound” these elements are [12]. The concept of cohesion was quickly categorised, with separate groups representing the different degrees of cohesion. The currently recognised classification designates seven levels of cohesion that a codebase may exhibit, roughly from lowest to

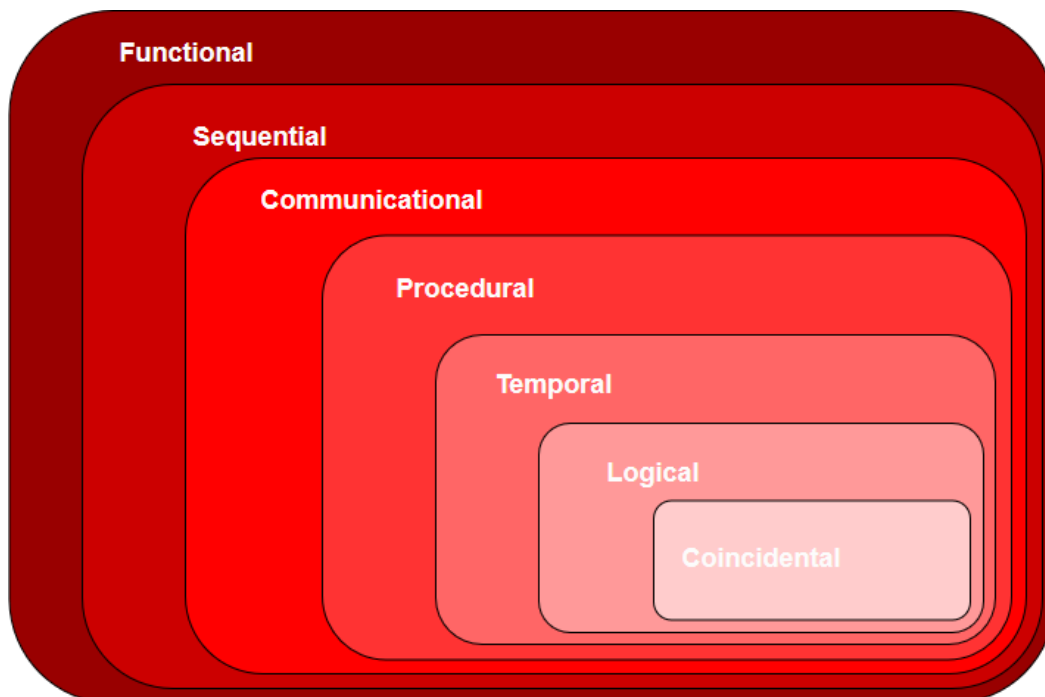


Figure 1: Functional cohesion encompasses all benefits from lower levels of cohesion

highest: coincidental, logical, temporal, procedural, communicational, sequential and functional as displayed in Figure 1 [13].

At the lowest level, signifying the smallest degree of correlation between module elements, coincidental cohesion should be avoided wherever possible. It indicates that there is no pattern or system behind the choice of structure and content, with each class/package basically constituting an arbitrary collection of functions. As such, the contents of each module must therefore be made accessible to multiple other modules. Low levels of cohesion, such as this, fall in direct contradiction to the principle of information hiding, which requires that the visibility or accessibility of mutable data be minimised to as few modules as possible [14].

Representing an important upgrade from coincidental cohesion, logical implies that some degree of thought has been put into the separation of module contents. Simply put, a module will be composed of elements that seem to be linked in terms of their purpose. As discussed by O'Hearn, Yang and Reynolds [15], the logical separation of a system into "conceptually distinct modules", leads to an increase in comprehensibility of the codebase, in accordance with the principle of modularity. The word "logical" here, however, refers to basic judgement, at a superficial level, with no real regard given for the degree of dissimilarity, or even incompatibility, between elements of the module when it comes to their internal workings. An example of such cohesion might include all output related functionality being grouped together, irrespective of whether they might be very different features, such as printing search results, flagging up a system warning or giving the system shutdown signal.

It is possible to think of temporal cohesion simply as an expansion of the previous category [13]. When this form of cohesion is present in a module, the constituent elements have been grouped according to their period of operation, or time at which they are invoked by the system, as suggested by the term "temporal" [16]. An illustration of this might be if the elements related to closing an application were combined into one module, with time of activation being the only obvious link between functions such as verifying the command via the UI, storing work to a disk and calling an end to ongoing processes.

Procedural cohesion represents a moderate improvement on basic temporal correlation. In procedural cohesion, module contents are related by a set structure or sequence in which they are invoked. This is often seen when the code execution is centred around an iterative or conditional control flow statement [17]. For example, in order to place an order online, the user might enter their name, select items from a menu, then enter their credit card information, all of which follow a procedure without being mutually dependent on each other.

Upon achieving communicational cohesion, it is finally possible to refer to the relationship between modules as fundamentally "problem-dependent". This means that, at this degree of cohesion, the correlation between modules begins to become more structurally involved, with the components of a given class or package being unified in having a single point of reference for input or output data, as illustrated in Figure 2 [13]. As each defined level of cohesion includes and improves on the previous level, communicational cohesion, of course, implies procedural cohesion, with the added data link increasing the "integrity of methods" within the system [18]. An illustration of this type of cohesion can be seen in a module which handles/calculates information relating to car make for a car dealer factory. The shared input might be the model record, whilst the output might be number of models sold, length of time since first stocked and overall profit yielded.

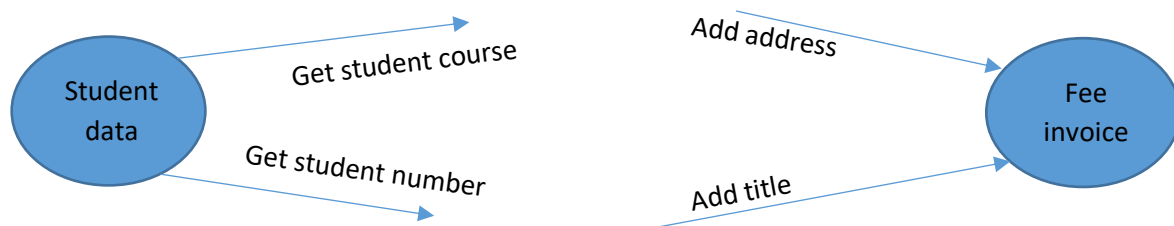


Figure 2: the delineated processes are both communicationaly cohesive

Contrastingly, sequential cohesion implies that, instead of sharing one point of reference, the control flow of the system chains each element of the module together, with information propagating through the module in a sequential fashion. This kind of flow can be understood to a basic level by considering a commonplace example, such as washing dishes. The sequence might be something similar to: apply detergent to sponge; soap dishes using sponge; wash soap from dishes. Here it can be clearly see that each element is dependent on the result of the preceding element, with failure of one component leading to failure of all subsequent tasks, in other words, the control flow connecting them is “regular” [19].

Lastly, the greatest degree of cohesion, with maximal interrelation between module elements, is designated as functional cohesion, by far the most desired form, fitting closely with the object oriented concept of “chunking” code for reuse [20]. For a class or package to demonstrate this form of cohesion, each of its internal elements must contribute to the completion of the same single process or task. In other words, to find the length of the hypotenuse of a right-angled triangle, the constituent elements might be “square length of other two sides”, “take sum of squares”, and “take square root”. The addition of a single extraneous element to this module, would cause it to lose functional cohesion. A more meaningful, real world example can be found in the use of libraries to ensure functional cohesion, for example the Comdlg32 dynamic link library in the Windows OS is responsible for much of the functionality which allows a user to interact with a dialogue box.

One can find multiple justifications for a prioritisation of cohesion in accordance with the list designated above. For example, it is instantly apparent that applying cohesion-based requirements will cause a codebase to become more comprehensible and easier for a developer to scan through. Grouping all elements in a module according to individual goals and functions significantly reduces the difficulty of working with a codebase in comparison to one in which the module contents are essentially random, with no real correlation between the individual elements [21].

In addition, cohesion is fundamental to maintaining the structural reliability of the individual classes and packages within an object-oriented codebase. Without giving consideration to cohesion, these modules will increasingly become larger and more complex, meaning that high cohesion is fundamental to code efficiency and efficacy [22]. The very existence of tactics and designed metrics which act as a method of quantifying the level of cohesion within a class or package, are evidence of the essential nature of cohesion to manageable, maintainable and problem-free codebases [23].

2.1.2.2 Coupling

Just as with cohesion, “coupling” owes its origin as a technical term to the early work of Constantine, who used it in relation to the intrinsic calibre of a codebase [12]. Having defined cohesion, it is now less difficult to formulate an understanding of the term “coupling” by comparison. Fundamentally, in contrast to the intra-module considerations specified by cohesion, coupling refers to the “inter-module” relationships, i.e. degree of correlation between separate classes or packages, as shown in Figure 3 [24]. Contrary to cohesion, however, coupling is a characteristic which should be reduced as far as possible, with the ideal codebase exhibiting high cohesion and loose coupling [24].

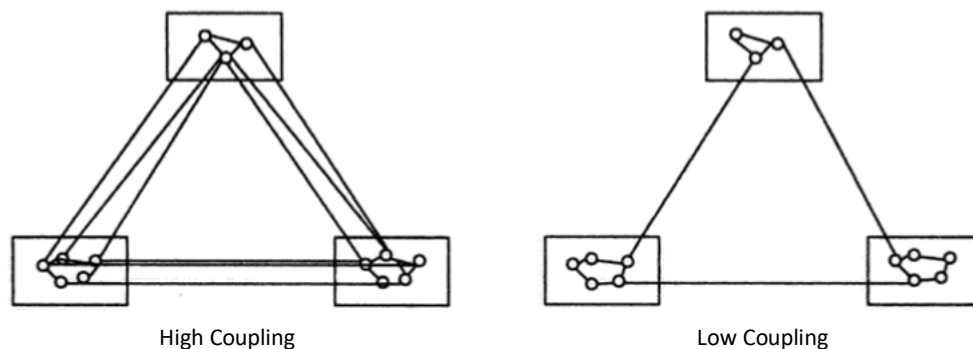


Figure 3: External connections represent coupling, internal connections represent cohesion [17]

Just as with cohesion, coupling can be sub-categorised into distinct types: interaction, component and inheritance [25], [26]. The first of these, interaction coupling, is concerned not only with the degree and type of interaction between elements of different modules, but also the volume and intricacy of data linkage they exhibit. This category of coupling can be split again into a set of distinct sub-types, spanning from the least desirable, content coupling, which would allow one module to have unlimited use of the data and functions in a separate module, to data coupling, which exhibits the least inter-module dependence, allowing only relevant parameters to be transferred between modules [26].

Contrastingly, component coupling, whilst implying that interaction coupling is present, seeks to highlight a different perspective, with relevance to how apparent or concealed the compositional connection between two modules is. Hidden coupling is taken to be the least desirable type of component coupling, whilst specified coupling is the most ideal. On the other hand, inheritance coupling refers to when there is an inheritance based link between modules, with one being an extension of the other, with variables originating from the parent but being present in the child class [26].

Notwithstanding attempts to unify the two properties in the overall concept of “connascence”, it can readily be observed that a strong analogy can be drawn between coupling and cohesion [27]. In other words, if there is strong, functional cohesion exhibited within the different modules of a problem, it will naturally be more possible to maintain a degree of separation between these modules. Low cohesion, on the other hand, with random or poorly considered module composition, makes it harder to minimise dependencies, lending itself towards high levels of interaction coupling. On the other hand, if attempts to increase cohesion leads to excessive subdivision of modules, coupling will, of course, begin to increase again, as the individual modules essentially become too specialised. Maintaining the ideal balance, with loose coupling and high cohesion, is fundamental to ensuring good quality code.

2.1.3 Information Visualisation

As previously discussed, one of the major disadvantages indicated by research to explain the lack of uptake of static analysis tools by developers was that they found the manner in which the generated

information was displayed was insufficiently “user-friendly and intuitive” [9]. From this, there is a clear link to the importance of the field of information visualisation. This discipline has no doubt existed in some form for as long as human beings have possessed eyes. However, information visualisation in its modern format, which has since become intrinsically associated with the idea of the human computer interface, has existed within a more manageable timeframe, with Bertin’s “Semiology of Graphics” [28], published in 1983, one of the formative works in the field. A brief summary of modern information visualisation might mention that it implies the study of “abstract phenomena” for the purpose of distinguishing trends and increasing understanding via optimised graphical display [29].

This idea of abstraction is a key aspect of presenting data in a comprehensible fashion. The word itself refers to the process of isolating particular elements of the whole in order to reduce its complexity, allow concentration and understanding of specific attributes and achieve a clearer appreciation of the fundamental ethos of the concept itself [30]. A practical example of this definition was demonstrated in Harry Beck’s simplification and abstraction of the map of the London underground system in 1931, see Figure 4 [31].

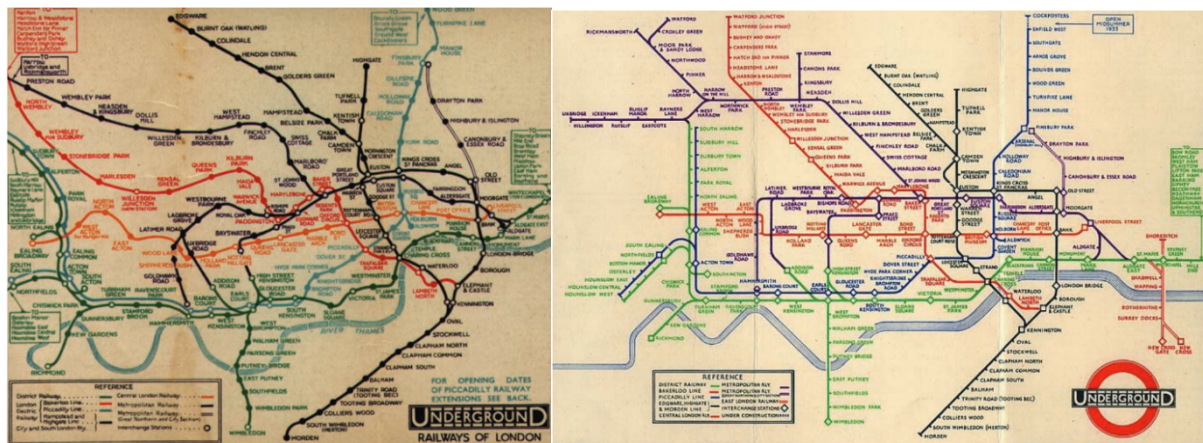


Figure 4: Beck’s abstraction (right) greatly simplified the pre-1931 tube map (left) [31]

The fundamental concept behind Beck’s alterations was abstraction. In effect, all properties, such as representative line shape; accurate distance scaling and so on, which were not intrinsically essential to the understanding and use of underground passengers, were dispensed with. All that really remained was a diagram which displayed, in a comprehensible fashion, the degree and nature of connectivity exhibited by each station, exactly as required in order to navigate the underground system.

Contrary to Beck’s elegant visualisation of the fundamentally important aspects of the London underground system, a tragic example of inhibited communication as a result of poor visualisation techniques was observed in the wake of the 1986 Challenger space shuttle disaster. As identified by renowned data-visualisation expert, Edward R. Tufte [32], in his book: “Visual explanations: images and quantities, evidence and narrative” there was a clear area of improvement in terms of pre-launch communication. Within the 24-hour period preceding the disaster, there had been extensive arguments put forward by the Challenger engineers, proposing a postponement of launch. The graphic used in an attempt to present the possibility of low temperature leading to dangerous levels of deterioration in the shuttle’s rubber “O-ring” seals is shown in Figure 5 [33]. Tufte points out numerous problems with this image, predominantly, the unnecessary and distracting inclusion of irrelevant space shuttle cartoons in the image, along with the unintuitive presentation according to date, as opposed to in order from low to high temperature. The inability of this graphic to sufficiently support the argument for launch postponement can be seen as one of the key factors in the

subsequent explosion of the shuttle, 73 seconds after launch, imparting a highly poignant reminder on the essential nature of effective visualisation of information.

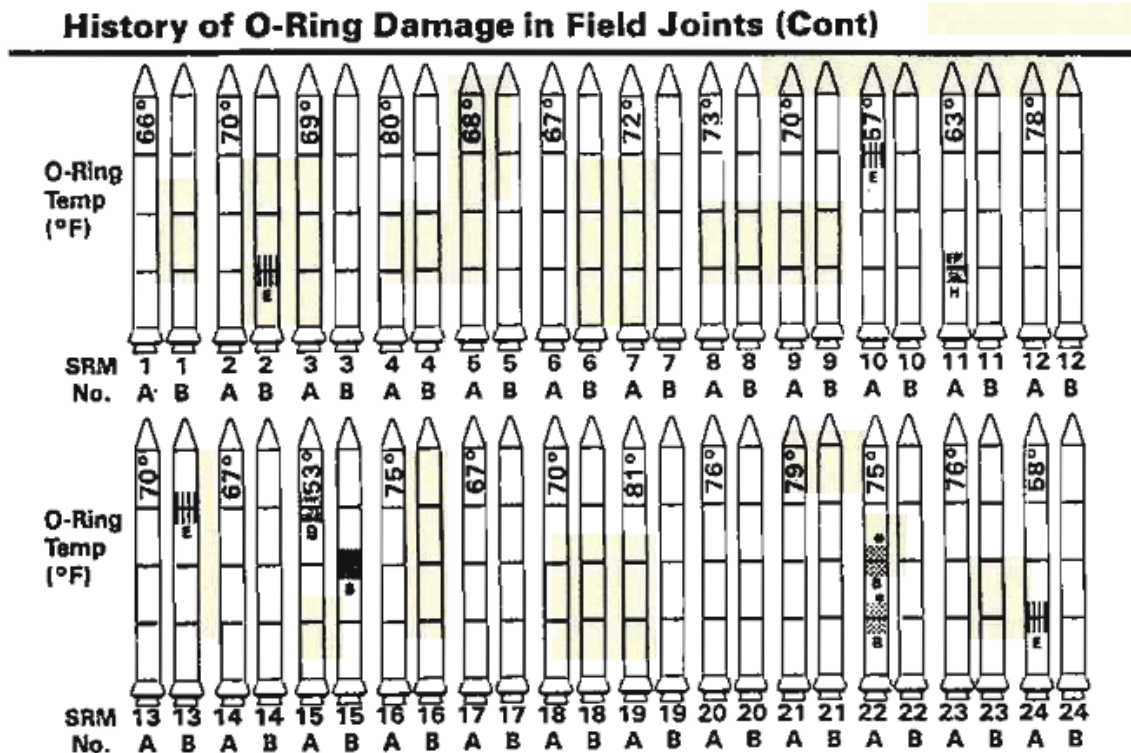


Figure 5: The original graph presented in an attempt to demonstrate O-ring failure [33]

Just as with the other sections of the overview chapter of this report, the concept of information visualisation and abstraction have particular relevance to both the general aim and specific objectives of this project. Further details on visualisation technologies are provided in the “display” and “existing technologies” sections of this report. The overall purpose of providing understandable static analysis via a graphical depiction of the codebase system, is well suited by the essential principles of modern information visualisation. A very definite analogy can be drawn between the example displayed in Figure 4 and the concept of representing source code dependencies and structure. Both hinge on accurately portraying the links and connections between, modules or stations, whilst removing extraneous details, such as contents or shape. From this, the idea of a directed graph representation of modules as nodes, connected by dependencies as edges, can easily be justified. It is for this reason that an overview section was deemed appropriate, to clarify not only the ultimate sources of motivation for the project but also to provide insight into the specific implementations it exhibits.

2.2 Dependency Extraction Engine

The aim of this section is to provide a brief review of the theory behind codebase dependency extraction, whilst also touching on some of the manual techniques and mechanics which might be implemented within the engine of a dependency visualisation system.

2.2.1 Forms of Dependency

The word dependency, as it applies in the context of this project, can essentially be used interchangeably with the word coupling which we defined in the overview section. In other words, if one piece of code has a dependency, this means that it won't work in isolation, instead it relies upon and is linked to another code segment and changes to that code may affect the function of the original element [34].

Dependencies between classes are categorised into a spectrum of different types within the Unified Modelling Language (UML), which will be addressed further in the display and existing technologies section of this report. The relationships between different entities in a UML class diagram, arranged in order from loosest to tightest coupling, are designated as: dependency, association, aggregation, composition, realisation and generalisation (or inheritance), as shown in Table 1 [35].

Dependency Form	Description
Dependency	The weakest form of coupling, any link between two classes or modules
Association	A more permanent form of the preceding dependency type,
Aggregation	The two elements are linked by a "has a" relationship
Composition	The two elements are linked by a "contains a" relationship
Realisation	A class "implements" an interface, which acts as a form or pattern
Generalisation	The strongest form of coupling, child class "is an" extension of the parent class

Table 1: Summarised dependency descriptions

Whilst all of the above categories represent different degrees of dependency, the type of relationship officially classified under the title "dependency" actually represents the loosest form of coupling. A dependency relationship implies only that one class must "know" about another class [36]. In other words, there is a one-way relationship between the two classes, with the dependent class being unable to exist if the independent class is removed. A metaphor of this might be given by the real-world example of a conversation, during which the car, house or behaviour of external person is referenced. That person is actually not present inside the conversation, but they must be known in order for that part of the conversation to make sense.

One step up from this in terms of coupling strength is the association relationship bracket. This implies that one of the attributes of the class contains a reference to another class [37]. In this way, it becomes a more permanent relationship than the dependency example above. Going back to the real-world example used previously, in the case of association, the conversation would centre more strongly around the external individual. Although they are still not actually a member of the conversation, they are explicitly referenced within it and form part of the framing for it. It can be seen that in this case, the reference is much less fleeting, the context of the conversation requires that the participants are aware of the other individual from the onset of the discussion.

Just as association is a stronger form of dependency, an aggregation relationship can be viewed as a stronger type of association [38]. This is commonly referred to using the “has a” phraseology, in other words, one class owns another class. The metaphor for this can be obtained if one takes the conversation example from a different approach. The relationship between the speakers and the conversation itself is one of aggregation, i.e. the conversation “has” participants. Once the conversation ends, the individual speakers will still remain, they do not persist with the conversation.

Contrastingly, a composition relationship, whilst very similar in theory to aggregation, represents an increase in strength as it implies that the composing elements of the class are in fact terminated when the entity which they compose is terminated. This is signified using “contains a” terminology, with an example of a composition relationship being given by the conversation, as before, and the words within the conversation. In other words, the conversation “contains” words, and when the conversation is ended, the words are likewise finished.

Realisation relationships occur when one entity defines the function and operation which is then “implemented” by another entity [39]. In other words, one class essentially provides the pattern for another class, acting as an interface, with the other class providing the realisation of that pattern. The analogy for this might relate the written set of talking points as the defining entity, with the actual conversation that matures from these being the implementing entity which “realises” that list.

Lastly, the most significant level of coupling is implied by the generalisation relationship. In other words, inheritance from one class to another. This is summarised using the “is a” phraseology, in other words a lion “is a” type of mammal, it possesses all the attributes of the kingdom Mammalia; warm blood, vertebrate, production of milk, etc, whilst also having specific characteristics of its own, such as a mane and a dewclaw.

It is fundamental that we link these somewhat abstract concepts of coupling or dependency types to the more applied idea of their relevance to an actual programming language. In the case of this project, the language of choice is the Java programming language. This is because it is the primary language of choice within the DVLA’s software engineering departments.

The ways in which a dependency can be created within Java can be separated into four basic types. An inheritance relationship can be explicitly called out using the “extends” keyword, identifying one class as the direct descendent of the class which follows the keyword. Similarly, a realisation relationship is called out using the “implements” keyword, which indicates that the structure of the implementing class is determined by a Java interface.

The other forms of basic dependency, association, aggregation and composition are addressed in one of two ways within Java. A reference to a class can be explicitly called out using the “new” keyword, i.e. when instantiating an object, or instance of that class, within another class. This might generally be thought to tend towards the more tightly coupled form of dependency, such a composition or aggregation, because of the fact that the object creation is unambiguous during compile time.

Lastly, the fourth and final callout type within Java, is something of a conglomerate of subtypes and tends to be used to represent the looser forms of coupling, i.e. basic dependencies and associations. These are signified when a class is called out within another class as either, an uninitialized variable type assignation, a return type of a method/function or as a parameter type passed into a method/function. All of these forms of coupling have an intrinsic impermanence to them, as any assignation of value to an object defined in such a way would happen during runtime.

2.2.2 Accessing the Structure of a Codebase

The previous section provided an introduction to the theory of dependencies in object-oriented programming in general, before touching on the specifics of Java codebase dependencies. It is now necessary that we examine the manner in which it is possible to interact with and extract these dependencies both on a more general theoretical level and more specifically to the project. Some of the different possibilities considered for choice of extraction technique are listed in Table 2 below.

Extraction Method	Brief Description
Textual analysis	Essentially writing a form of lexer/parser from the ground up, based on callout recognition
PSI analysis	Abstract syntax tree can be built and analysed via the Program Structure Interface files
ANTLR	Generating a lexer and parser pair automatically via this recognition tool
JavaParser	JavaParser and JavaSymbolSolver libraries can extract information directly from Java files

Table 2: Summarised description of extraction technique options

2.2.2.1 Textual Analysis

Perhaps the most conceptually intuitive method, although not necessarily the most straightforward or easiest to design or implement from a programming perspective, would be to simply view the codebase as a purely textual entity.

In other words, taking a basic Java project as our example, the source code may be separated into modules which are made up of classes, each class being a “.Java” file, containing keywords and class references as previously discussed. It is possible to create a file reading program which analyses these class files, identifying the relevant keywords and building up a picture of the dependency structure from this.

Sticking with the Java programming language, one initially notices a very important keyword in the form of “import”. This phrase is used whenever classes from different packages invoke each other. At first glance it appears that the entire coupling substructure can be extracted using this keyword alone, as it provides a very definite dependency mapping between classes. However, two issues would appear if one were only to analyse the class import statements for each file. Firstly, the strength or degree of coupling between two classes could not be accurately measured using only imports. This is because an import statement only appears once, irrespective of number of usages of that class. As well as not quantifying the dependency, it also does not provide any information as to the type of dependency being created. Secondly, and perhaps more importantly, the “import” keyword is not required when invoking classes within the same package. In other words, any dependencies between classes in the same package are not called out within the import statements and therefore would be missed by a file reader which centres solely around these statements.

When identifying the inter-package dependencies, the discussed issues focussing on the import keyword are largely minimised. However, if one wishes to descend to the lower level of module and extract class dependencies, it will be necessary to include reference to the other dependency keywords mentioned previously. Furthermore, as discussed, dependencies can be created without referencing a specific keyword, if they come from runtime object initialisations.

As such, it can be supposed that absolute or complete dependency extraction via a file reader or textual analysis program cannot rely solely on the specific, dependency linked, keywords of the programming language, but will also have to investigate the use of class names. When taking into

account the presence of entities such as comments, strings and so on, which can produce character patterns identical to those of class names, it quickly becomes apparent that textual analysis is a non-trivial route to dependency extraction. It is also clear that any alternate means of accessing the dependencies would require a less black-box approach than pure language comprehension. In other words, the alternative to this would be a program that “knows” what Java is and recognises the structures and substructures of Java files as entities rather than simply as text.

2.2.2.2 AST, PSIs and parsing

The Abstract Syntax Tree (AST) is a generated entity which represents the structure of the source code. It identifies the components of a module in ever increasing levels of detail, following the tree graph format, with the leaves of the tree being composed of the module’s fundamental constituents. Meanwhile, the “edges” on the tree structure represent the manner in which the leaves interact with one another [40]. Gaining access to the AST allows one to identify all the components of a particular class, including dependencies, as demonstrated for an extremely minimal java class in Figure 6 [41].

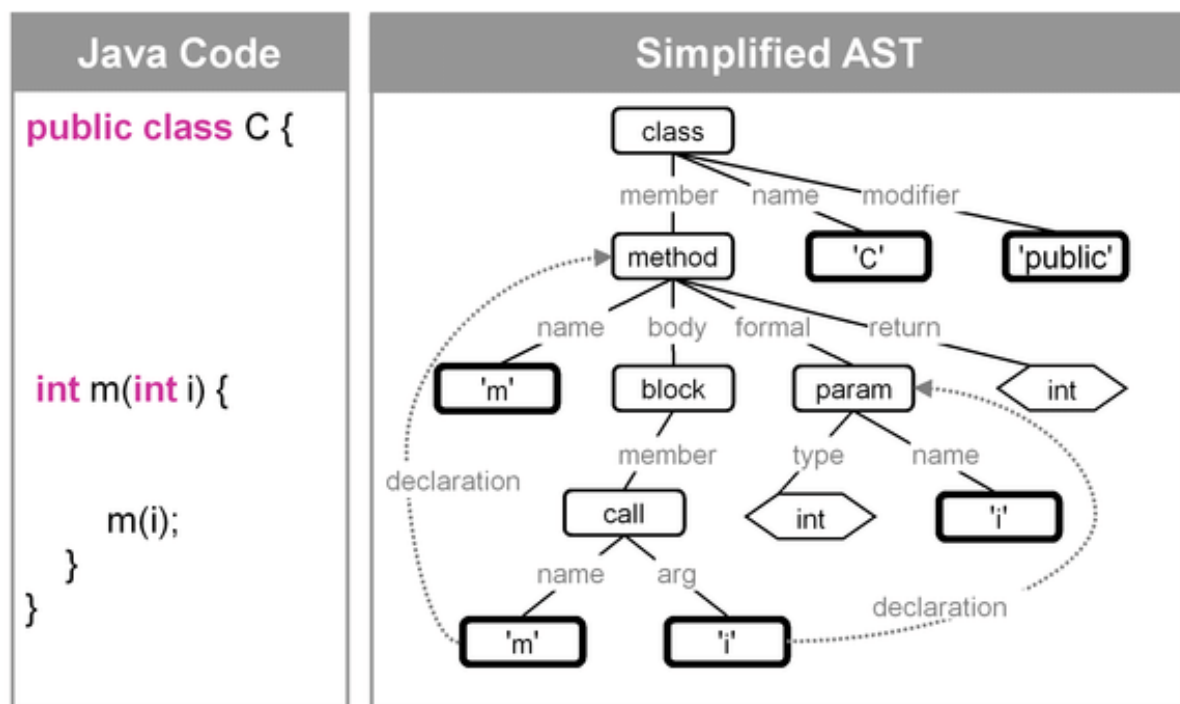


Figure 6: Abstract Syntax Tree (right) for simplified Java class and recursive function (left) [41]

In the most proliferate and default Integrated Development Environment (IDE) used at the DVLA for Java, JetBrains’ IntelliJ [42], an annotated AST is automatically created and stored in the form of a Program Structure Interface (PSI) file. This enables users to iterate through it, to isolate all of the constituent elements by essentially “walking” through the structure. This is extremely useful when creating plugins for Java on the IntelliJ IDE, as it provides an interface through which a plugin can obtain a meaningful structure for the different elements of a codebase, and also modify them individually. Plugins have been created specifically for the purpose of accessing the PSI files more interactively, for example the PsiViewer plugin [43]. This provides a layered list which identifies the locations of different elements and element types within the codebase as well as the properties of each element and the overall structure of the PSI, as shown in Figure 7, which highlights some of the initial PsiViewer analysis of a simple function.

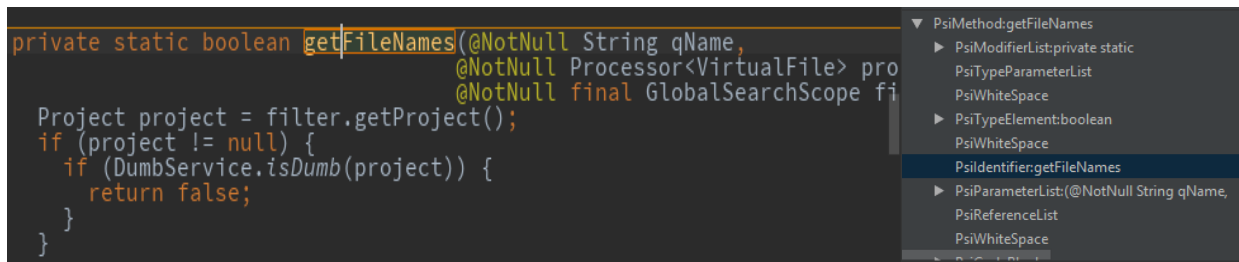


Figure 7: Sample screenshot of PSI structure viewer (right) for Java code in IntelliJ (left)

2.2.2.2.1 Lexical and Syntactic Analysis

Lexical analysis is the process of splitting up a language into its individual components or tokens for the purpose of associating them with a legal use-case or meaning [44]. Using a lexical analysis program or “lexer” in this way, a section of code can be passed into the program for the purpose of identifying its fundamental units for further use [45]. In a similar fashion, the program discussed in the preceding section can in some ways be viewed as a limited form of lexical analyser, which has the sole purpose of returning the “lexical items” which identify coupling to other modules of the codebase.

Closely related to lexical analysers are parsers, which act towards a similar general goal of attaching meaning to text, and the two are often combined to provide part of the functionality of a compiler program [46]. However, parsers are fundamentally different from lexers in operation and purpose, focussing on what is traditionally referred to as syntactic analysis. The difference between lexical analysis and syntactic analysis can be summarised by the general statement that a lexer essentially looks at the “word” within a sequence of a particular language, while a parser looks at its grammar [47]. In more formal terms, within the Chomsky hierarchy of grammars, a way of classifying the different constituents of language introduced in 1956 by Chomsky [48], lexical analysis is relevant to “type-3” or “regular” grammars. These consist of regular expressions and do not interact with nested syntax, which can most easily be understood as the inclusion of layers of parentheses within a sentence, an example of one such grammar would be the Comma-Separated Values (CSV) format. Meanwhile, parsing is relevant to “type-2” or “context-free” grammars, which do not exclude nested syntax but prohibit modulation of meaning based on context-sensitivity and act as the basis for most programming languages, an example of one such format would be Hypertext Markup Language (HTML) [49].

An example of one of the established tools used to generate lexers and parsers is ANTLR (Another Tool for Language Recognition). This can, in some ways, be thought of as “code to write code to write code”. This somewhat convoluted statement can be understood when one realises that the ANTLR parser generator is used by developers to build further tools, for the purpose of static analysis, compilation etc, which they then use in the process of creating a program for their end users. The basis of ANTLR is that a “formal language description” or “grammar” is defined and then ingested by the program and used to create a language recogniser (composed of a lexer and parser) which is used to create an AST from input code. ANTLR also includes the capability to create code which will allow traversal of this tree, facilitating interaction with the input code [50].

One alternative to generating a language recogniser specifically for the Java language would be to use the existing JavaParser library, which acts as a tool to help interact with, modify and analyse code Java codebases [51]. Using the functionality provided by JavaParser, it is possible to not only generate and traverse an AST but also transform the code itself, as well as generating additional code from this, such as automating the generation of a class file with a particular format, including variables and methods [52]. Using the JavaSymbolSolver library, which was initially created as a separate but linked

project, it is possible, in addition, to find the codebase locations of different elements and create further links within the AST graph structure. This feature seems particularly relevant to the purpose of building up a structural representation of the coupling present within the source code. Examples of use of the JavaParser library include its involvement in the creation of purpose genetic improvement frameworks for Java (which aim to automatically optimise codebases through small changes or mutations) providing some evidence for its suitability as an interface to small scale Java automation projects [53]. In addition, Fauzi, Hendradjaya and Sunindyo [54] note its inclusion in their research project, involving the automated generation of a UML sequence diagram from source code via an AST constructed using JavaParser.

The objective behind this section of the literature review was to provide a basic grounding, not only in the fundamentals of the conceptual entities dealt with by this project, i.e. the dependencies themselves, but also to give a more “applied” introduction to the mechanics of the extraction process. For this purpose, it was necessary to address some of the more specific aspects of the Java language, as well as defining some of the related data structures that can be generated from it and some of the ways in which these data structures can be generated and interacted with. This gives sufficient basis to begin an examination of some of the analytical techniques which can be used to process and examine such data once it has been extracted.

2.3 Dependency Analysis

In addition to accessing the dependency structure via parsing and extraction from the source code and subsequently providing a visualisation of this information represented as a coupling graph, it is possible to obtain a more quantitative understanding of the overall quality of the codebase via dependency analysis. This area, whilst not being an essential piece of functionality to the codebase visualisation system, holds significant potential for enhancement of overall understanding and could otherwise easily be overlooked. This section intends to introduce one of the structures which could yield a more analytical representation of coupling within the codebase, whilst also providing a brief overview of the quantification of source code dependencies and manners of identifying poor quality via, for example, pattern-based indicators.

2.3.1 Dependency Structure Matrix

The phrase “Design Structure Matrix” (later also referred to as a Dependency Structure Matrix or DSM) was brought into being by Steward [55], as evidenced by his publications during the early 1980s, to refer to a mathematical entity used within engineering management and planning to help establish the “consequences of a change in any variable on the rest of the variables in the system”. DSMs are used extensively within a range of different fields, such as product, organisation, process and multidomain architecture [56]. Within the field of computer science, a DSM typically signifies a 2-dimensional array, both dimensions of which are made up of an identical list containing the modules or sub-elements within the codebase. The tabular mapping between these two dimensions then corresponds to a representation of coupling within the codebase, with each cell, i.e. each intersection of row and column, signifying the presence or absence of a dependency between the two elements corresponding to that row, column pair.

One of the reasons for the potential inclusion of such a technique within a coupling static analysis tool would be the possible advantage gained when investigating large codebases. Although graphs are more visual and often more easily digestible to the human viewer, it can be quickly and easily understood that the scaling of visual complexity of a simple codebase dependency graph will exhibit a strong, positive, non-linear correlation with the scaling of the codebase itself. As such, a DSM might provide a form of visualisation which is less prone to getting “cluttered” than the graphical representation, whilst still delivering a much more intuitive interface than direct codebase inspection or simple listing of dependencies. Indeed, this has already been demonstrated in research comparing the degree modularity exhibited by open source codebases with that found within proprietary codebases. The DSM visualisation approach was used to great effect in highlighting the initial quality difference between the large Mozilla and Linux codebases, in addition to the significant structural change to Mozilla source code after a directed effort to increase modularity. This also demonstrates the efficacy of the DSM in combining a visual representation with a quantitative method of extracting data, as observation of either only the coupling density statistics or only the patterns within the matrix would not have provided a complete understanding [57].

Furthermore, the DSM contents need not be limited to simply a binary “coupling or no coupling” system. The DSM density, mentioned above, may actually be included into the matrix representation, with the contents in each cell being equal to the number of separate dependency instances within the pair of classes or modules that cell refers to. This is one of the methods utilised within the NDepend DSM feature, with numbers which show unidirectional vs bidirectional dependencies being distinguished using colour [58]. However, the technique of incrementing this density number equally for all forms of coupling is not necessarily the most appropriate quantification, as indicated in the section on forms of dependency, there is a basic spectrum of dependency strength based on type. The

next section seeks to examine differences in the way coupling can affect code quality, and possible measures thereof. In addition, although it is possible to quickly identify dependency patterns within the codebase by simple visual inspection of the DSM, a brief look at dependency patterns in general may prove beneficial [59].

2.3.2 Dependency Measurement and Pattern Identification

Identifying a numerical manner in which to quantify degree of coupling is an area of potential utility. One example of such a method for class dependencies takes into account the amount of code shared between the dependent class and the class upon which it depends. The ratio of this with the total amount of code within the second class is then calculated. This is then multiplied by a coefficient representing the scope between the two elements to obtain a real number between 0 and 1 which quantifies the degree of coupling between the two classes [60].

Other methods for quantifying degree of coupling include slicing methods such as that proposed by Kumar and Khalsa [61]. This involves essentially removing extraneous elements from the program so as to focus only on the features which are being analysed. Use of this method indicated potential to minimise coupling whilst maximising cohesion, producing an increase in system quality. Other metrics used in a similar fashion include Number of Dependencies In/Out (NDepIN/NDepOut) which quantify the number of classes which depend on a given class/are depended on by a given class [62].

Identification of patterns within the dependency structure is of similar relevance. Perhaps the most obvious example is that of cyclic dependencies [63]. It should be noted that although the individual classes in a particular section of source code may be highly cohesive and not exhibit a high dependency density value, their effect on code quality can still be significantly negative if they exhibit cyclic dependencies. This would imply that there is a closed loop of dependent classes, for example: A relies on B, B relies on C, C relies on A. In this situation the entire cyclic group must be taken as a single element, and cannot truly be separated into its classes, as a change on any one constituent will affect every other constituent. As such, it is possible to have worked hard to create a codebase with the appearance of object oriented modularity, which is, however, not truly modular and essentially ends up as the archetypal “big ball of mud” [64].

Having given some thought to the analysis and quantification of the extracted data, it is now appropriate to approach the more qualitative area of visualisation. Although the DSM provides a good balance between statistics and pattern recognition, it is in the display functionality which produce a truly intuitive grasp of the coupling within a codebase.

2.4 Displaying Dependencies

It could arguably be said that the dependency extraction engine represents the essence of the entire static analysis tool, providing the data on which all further analysis and presentation is based. However, as addressed in the overview section, information alone is not always sufficient as a means of providing comprehension and understanding of the key features of a system. The isolation of dependency information from all extraneous information (in this case the rest of the codebase) can be viewed as a form of abstraction, following which, it is necessary to visualise in a simple manner which is as intuitive as possible. This section aims to discuss the basis for a graphical display methodology, as well as different display patterns which might be utilised within the dependency visualisation system.

2.4.1 Graphs

The word graph has come to signify a wide range of different visualisation media in today's world. However, the more colloquially understood meaning of entities such as line and bar charts used to represent data, first introduced by Playfair [65] in 1786, is less relevant to this project and will not be addressed. Rather, what must be dealt with here is the mathematically understood meaning of the word graph, whose origin is somewhat harder to pin down, often attributed to Euler in the 1730s, but having been refined over a range of time preceding and following him [66].

The accepted definition of a graph as a mathematical entity is that a discrete group of points or items which exhibit some degree of connectivity with each other, i.e. they are linked by some form of relationship. In a manner similar to Beck's underground map, discussed in the overview section 1.3, graphs represent a mathematical abstraction of a particular data set or system. The presence of different data objects within the system is represented by their appearance as nodes within the graph system, whilst the existence or absence of a relationship between any two objects is shown by the presence or absence of a line, joining the two points. These points are referred to as vertices and the lines joining them together are known as edges, providing a more formal mathematical description of a graph as the following "ordered triple" [67]:

$$G = (V(G), E(G), \psi_G)$$

Here the graph G is represented by the complete set of vertices V , the set of edges E , and the "incidence function" ψ which provides a mapping between each edge and a pair of vertices as demonstrated in the following simplified example shown in Figure 8.

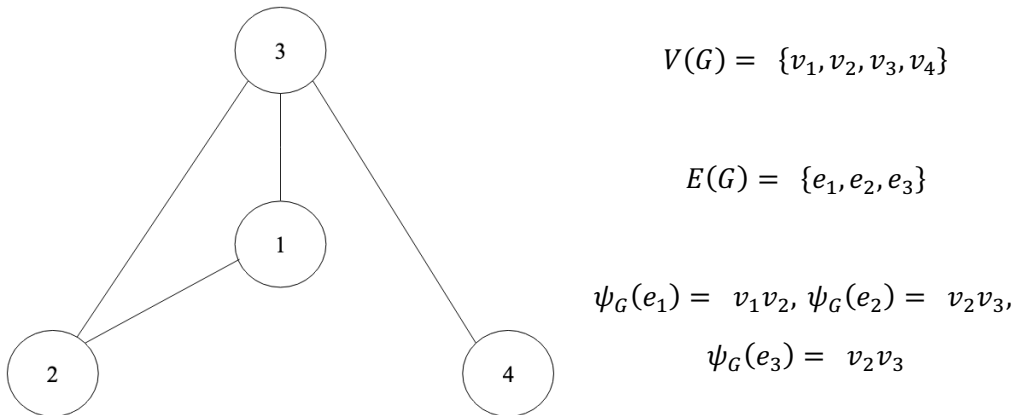


Figure 8: Simple undirected graph G complete with formal notation indicating components of G

It should be noted that Figure 8 displays an undirected graph. In the case of dependencies, it can quickly be understood that a directed graph is necessary in order to comprehensively represent the structure. In other words, in order to display a dependency of module 2 upon module 1, e.g. 2 inherits from 1, the edge connecting the two must be directed from node 2 to node 1. The clear difference between this form of directed graph over an undirected graph is that in the latter the incidence function links each edge to an unordered pair of vertices. In the former, on the other hand, each edge is linked to an ordered pair, which would be represented by the inclusion of arrows to the edges, denoting vector direction between nodes [68].

There are additional properties of a graph structure which make it well-suited to the purpose of containing and representing the dependency substructure of a codebase. As mentioned in the analysis section, one dependency feature which can represent a problem for a codebase, or poor-quality code, is the presence of a dependency cycle. This is well addressed by the concept of “cyclic” vs “acyclic” graphs, with the former being a graph which contains at least one node pattern in which a node can be revisited in a path which does not revisit any of the intervening nodes and the latter being a graph in which this is not possible [69]. An example of a cycle can be seen in Figure 8 including nodes 1, 2 and 3, where starting at node 1, it is possible to visit 2, visit 3 and then return to 1 without repetition, making the graph cyclic, if any one of the edges connecting these nodes were removed, the graph would become acyclic. In addition to this, the property of connectivity within graph theory can be linked to the aim of reducing coupling, as it identifies the most efficient way to split a graph into subgraphs, therefore reducing long chains of dependency [70].

Finally, although there is no single uniform syntax or markup language for communicating graph structures, there are a number of well-defined and well documented formats in use. These include graph description languages such as the DOT language, which is supported by the graphviz package; GraphML which extends upon a basic XML format and the Directed Graph Markup Language (DGML) which was developed by Microsoft to represent cyclic and acyclic directed graphs [71]–[73]. This is of particular use when considering the design of a possible storage interface between the dependency extraction engine and the display option. Use of a defined and accepted format will facilitate modularisation and separation of concerns between the extraction process and the display tools, which are discussed in the following section.

2.4.2 Display Tools

In addition to the range of possible graph description languages touched on above, there is an even larger range of available options for the purpose of visually displaying a graph structure. These range from pre-existing software packages available, which are either downloadable or available via an online interface, to graphing libraries compatible with the java language or even developing graphing options from scratch, despite Java’s lack of a default graphing framework [74]. A key concept when considering these options will be drawing the distinction of where it is necessary to create unique functionality, with full control, and when such efforts are best avoided to prevent “recreating the wheel”.

One of the available options for producing visual representations of graph structures is Graphviz, an open source package which utilises the DOT language and has online accessibility in the form of WebGraphviz [75]. First brought into development in the 1990s, predominantly by Gansner and Ellson [76], graphviz focuses on visualisation of numerous types of static graph, including directed graphs. An important feature of the tool is its inbuilt algorithms for spacing and de-cluttering, which, as shown

in Figure 9, go some way towards helping to prevent the input graphs from being displayed in an unintuitive and unwieldy fashion [77].

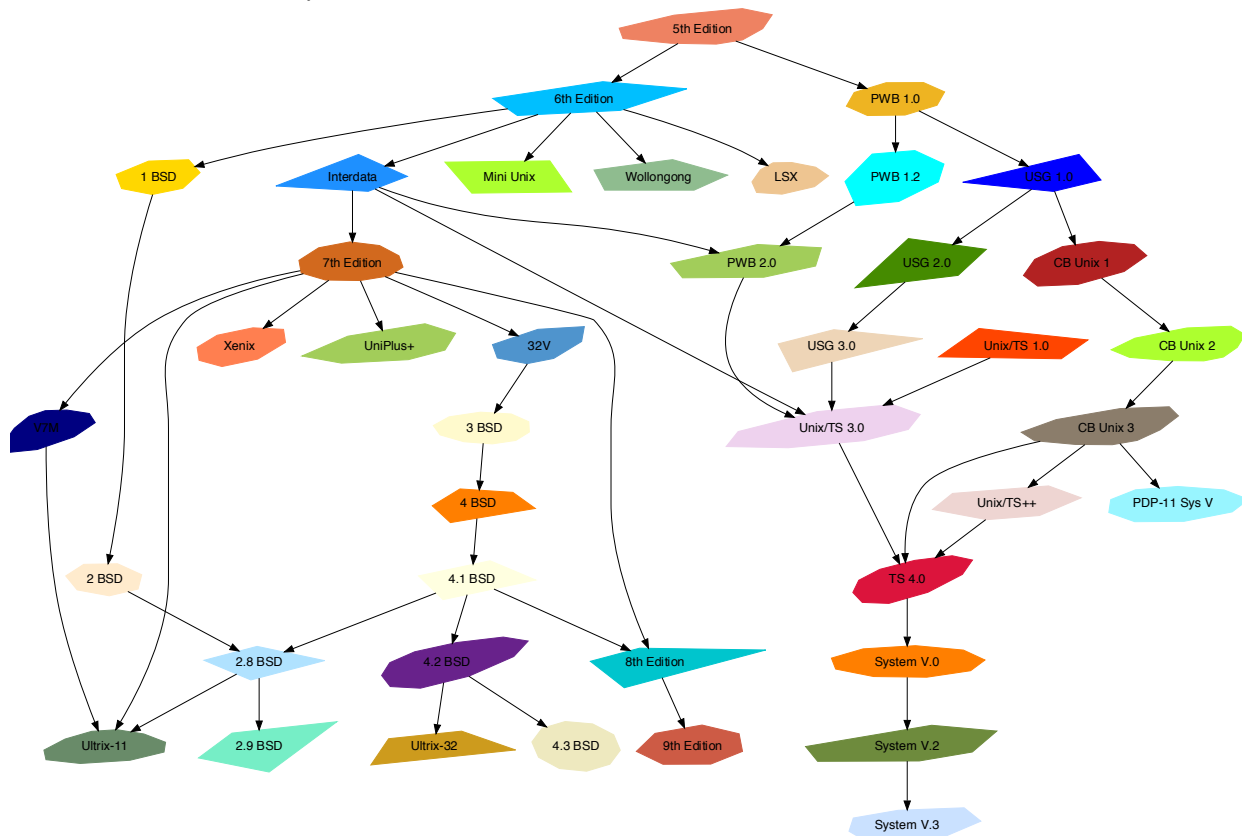


Figure 9: Example graphic representing the Unix Operating system created via graphviz [75]

Another display tool of potential relevance to the process of displaying dependencies is the UMLet application, another open-source visualisation tool, a sample screenshot of which can be seen in Figure 10 [78]. UMLet is designed specifically towards visualisation of code and code functionality, based around the UML developmental modelling language, with set formats for a significant range of different software components and relations [79]. It is designed to be lightweight and has come out favourably in comparisons with the popular commercially available UML design software Rational Rose, with emphasis on its speed and ease of use [80].

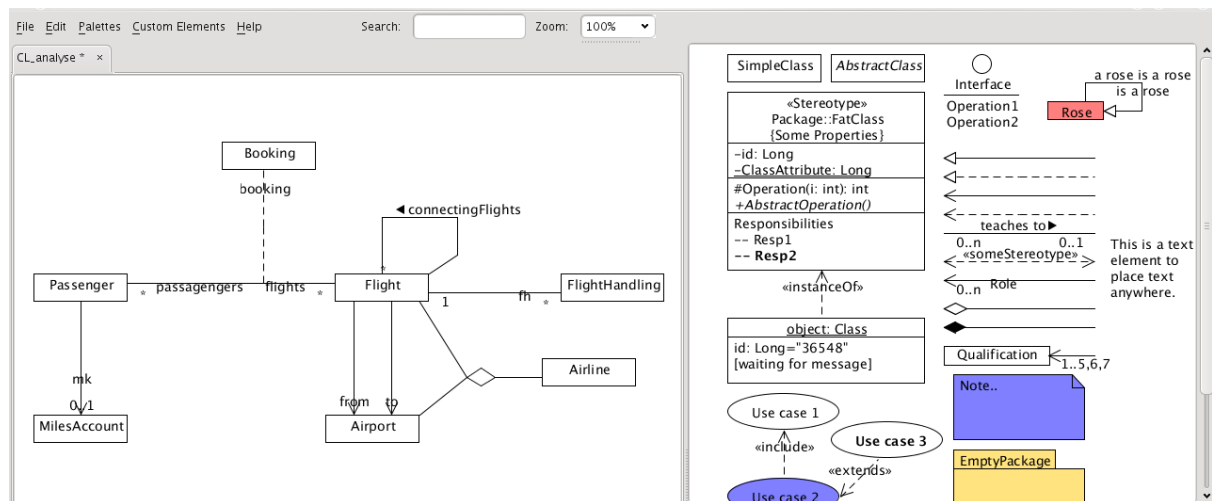


Figure 10: Sample screenshot of class diagram for airline system produced using UMLet [78]

There are a large number of other potential options when it comes to graphing the codebase, for example the Gephi software package offers further layout handling control, as well as dynamic and interactive functionality [81]. In order to obtain finer control of layout parameters, the use of a java graphing library such as the JGraphT component or the Java Universal Network/Graph (JUNG) framework [82], [83]. Although the implementation of a library affords one more control and choice, the use of a pre-grouped graphing package will ultimately enable more functionality.

This section has addressed the area of the visualisation system related to display, through first providing a basic theoretical understanding of the mathematical graph entity and its representation as a data structure. In addition, it has introduced some of the different options available when it comes to implementing the visualisation of such a data structure, through graphing packages and libraries. The natural progression of this is to then examine the technologies which provide not only display, but the entire functionality, i.e. existing dependency visualisation systems, which are covered in the following section.

2.5 Existing Technologies

The previous sections of this literature review have provided an overview of the research area in general and introduced each of the conceptually separate elements which will constitute the visualisation system. This final section now seeks to provide a brief rundown of some of the existing technologies and frameworks relevant to the project field of static analysis of dependencies structure and codebase structure visualisation. These can be summed up as either manual frameworks which serve the same general purpose as the proposed system or designated software packages/plugins which provide a variety of related features on a range of different platforms.

2.5.1 Manual Systems

Perhaps the predominant method for visual communication of structural information about source code is the Unified Modelling Language, which has discussed in section 2.2.1 on forms of dependency and has been mentioned in the previous section 2.4.2 on display tools. UML, developed by “the Three Amigos” (Jim Rumbaugh, Grady Booch and Ivar Jacobson) in the mid 1990s, extends far beyond the boundaries of dependency visualisation in the scope of its usability [84]. There are numerous different diagram types, used to represent customer use cases, the logical flow of a system and more. The primary focus of this project, however, is on the class diagrams which represent the way in which different components of a system invoke and interact with each other.

Although UML is often used as a design tool for planning before having written the code, with the advent of agile methodologies, the involvement of UML has become somewhat more integrated with the development process. It is not uncommon in organisations such as the DVLA for UML or UML-inspired diagrams to be drawn up either on paper or using one of the many software packages which offer UML drawing tools, such as StarUML, Visio or Edraw [85]. Alternatives to UML include the C4 model, which uses a set of four levels of visualisation: context, container, component and code, each of which addresses the structure of the code in an increased degree of detail from the previous level of focus, as shown in Figure 11 [86]. The degrees of detail most relevant to this project correspond to the component and code visualisations, with component representing the foundational stage of the visualiser, i.e. displaying inter-package relationships, and with code representing a minimal UML type class diagram. Another alternative visualisation method used by software engineers takes the form of platforms such as the WebGraphviz application, mentioned in section 2.4.2. This allows manual entry of code dependency structures via browser using a set format, before displaying a directed graph of nodes and edges which visualises the coupling within the codebase.

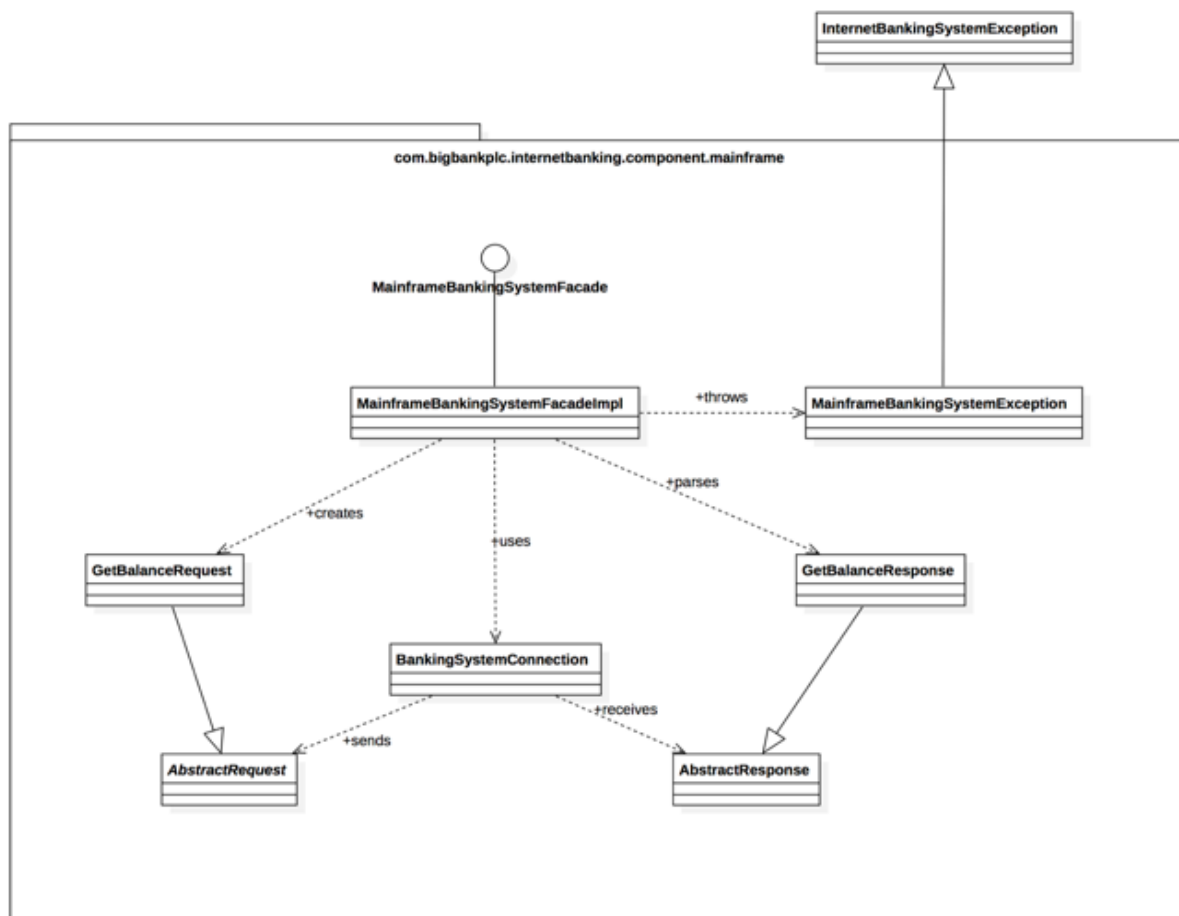
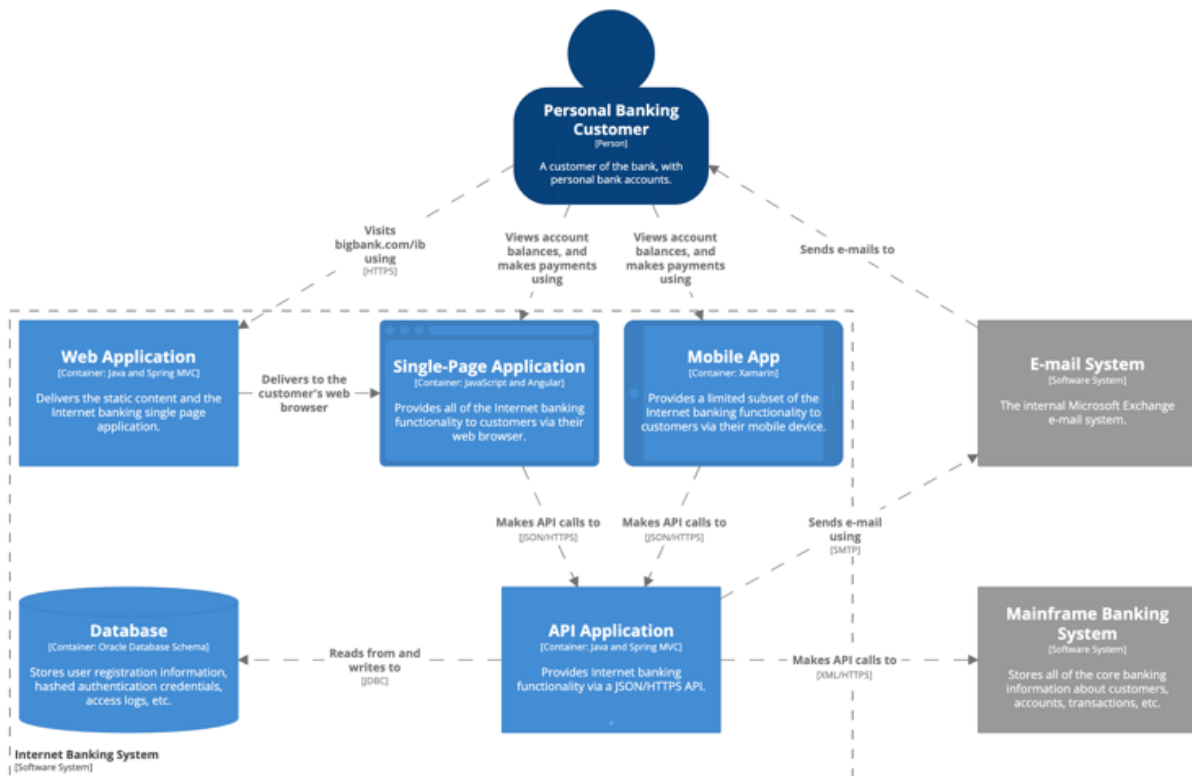


Figure 11: Level 3 C4 component diagram (left) vs level 4 UML class diagram (right) of 1 component of same system [86]

In addition to its uses as a tool for planning out the structure of a new system and visualising an existing one, UML can actually be used as a contract on which to generate the source code. This can be seen in functionality provided by Microsoft Visual Studio for C# code. The structural clarity provided by the class diagram system and the inter-class relationships/dependency mapping is such that a skeleton of the codebase itself can be generated from the diagram [87]. Naturally, it is possible to do the opposite, reverse engineer a class diagram from the body of code itself in an automated fashion, with some developers going beyond simple UML representations, and creating a tool to autogenerate a 3D relationship diagram for visualising large codebases [88]. This general area of generation of visualisation media automatically from the codebase is the topic of the following subsection.

2.5.2 Automated Systems

One of the pre-eminent code dependency visualisation systems is the commercially available NDepend software package. This is a static analysis tool for .NET language code which includes a range of different features, including heat maps and the provision of detailed code quality metrics [89]. The features of particular relevance to this project, however, are the dependency graphing application which presents classes as a system of nodes connected by edge dependencies and the DSM format discussed in section 2.3.1, which provides a more numerical representation of dependency structure, as shown in Figure 12. This is one of the features which allows NDepend to more easily capture the structure of larger codebases, where a graph view may become too visually complex to interpret. By integrating the matrix and graph functions, NDepend is able to provide specific views of subsections of the codebase, with the DSM being interactive to allow display of selected elements in graph mode. Furthermore, the identification of dependency cycles is also well-implemented in NDepend, with red borders on the DSM being used to highlight these sections (see Figure 12), allowing them to be examined more closely in graphical format.

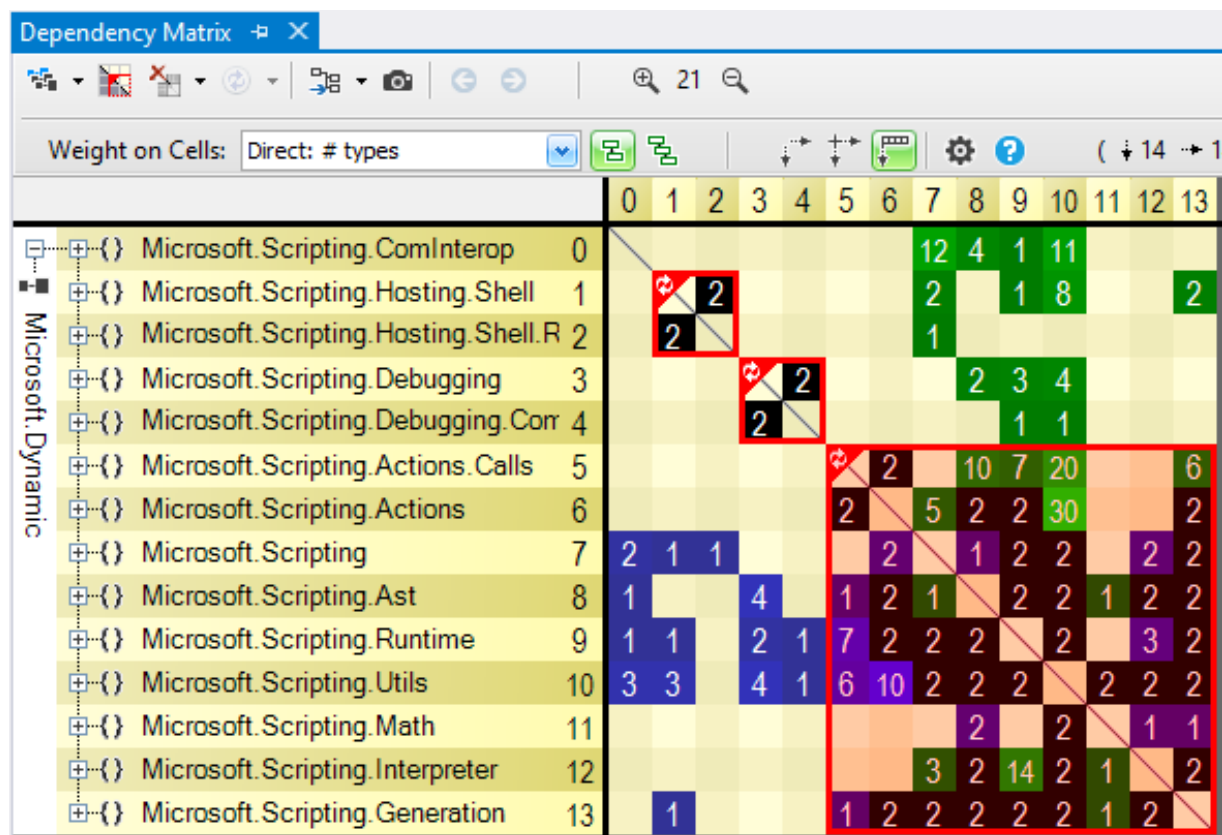


Figure 12: Simple NDepend DSM with cyclic dependency highlighting

A similar application to NDepend, Sourcetrail, has the advantage of being available as a free version for non-commercial use, supporting the C, C++, Java and Python languages [90]. Like NDepend, it is a static analysis tool, however, it focuses more exclusively on dependency mapping. One of the key advantageous features offered by this software package is the interactivity of the Graphical User Interface (GUI) that it provides. Sourcetrail's display format integrates a view of the source code into the graphical dependency view, with the ability to link in the editor used to develop the source code, allowing effective presentation of areas of the code in which quality could be increased, as shown in Figure 13. This, in combination with the symbol search options provided in-program and the ability to switch dynamically between views of different sections of the codebase, via either the source code or the graph view, allows successful decomposition of the code into its constituent modules.

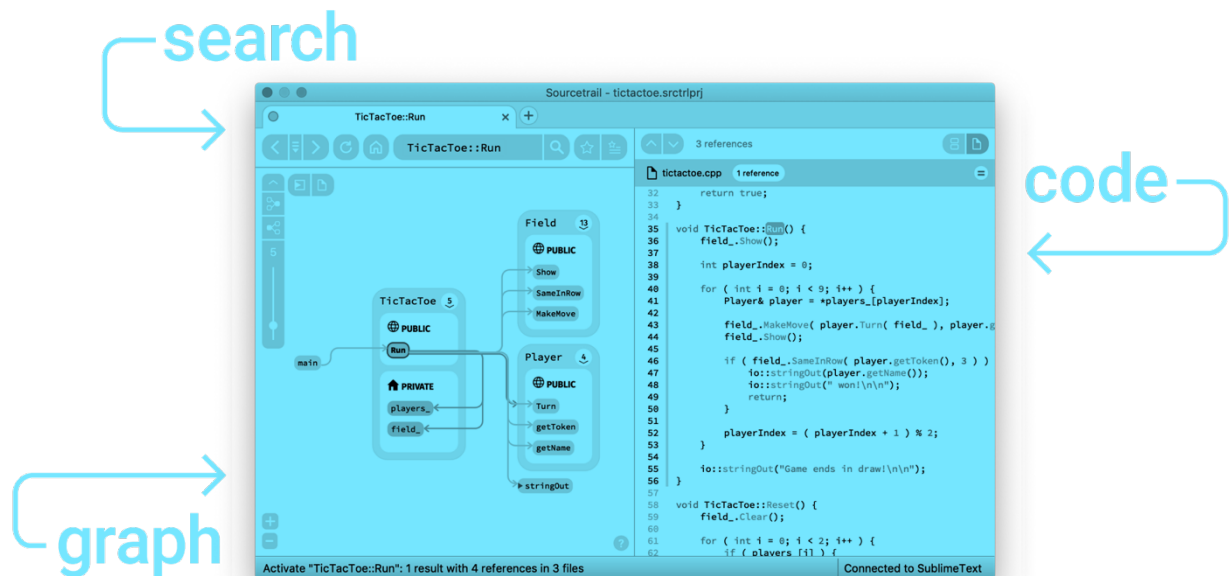


Figure 13: Snapshot of Sourcetrail display, showing graph and code views as well as search functionality [90]

Sonargraph offers a third option in the field of codebase dependency visualisation software, exhibiting a range of products, with Sonargraph-Explorer being a free license dependency analysis tool for Java, C# and Python, whilst Sonargraph-Architect incorporates those features into a more powerful commercial version which also supports C and C++ [91]. Amongst the unique features of Sonargraph-Explorer are the layered graph view used to present dependency patterns, providing a solid foundation for applying the principle of separation of concerns, as shown in Figure 14. Perhaps more importantly, the sonargraph tool focuses on the identification of cyclic dependencies, with each cyclically linked group of elements being presented as a single module. This module may then be further analysed in a separate cycle viewer function, see Figure 14, with an additional cycle detection option being provided by the hierarchical "exploration view" of the program. The use of Sonargraph dependency analysis tools has been credited for a 90% reduction in defect incidence rate in the BMW company, as presented in a case study by Wagner and Zitzewitz [92].

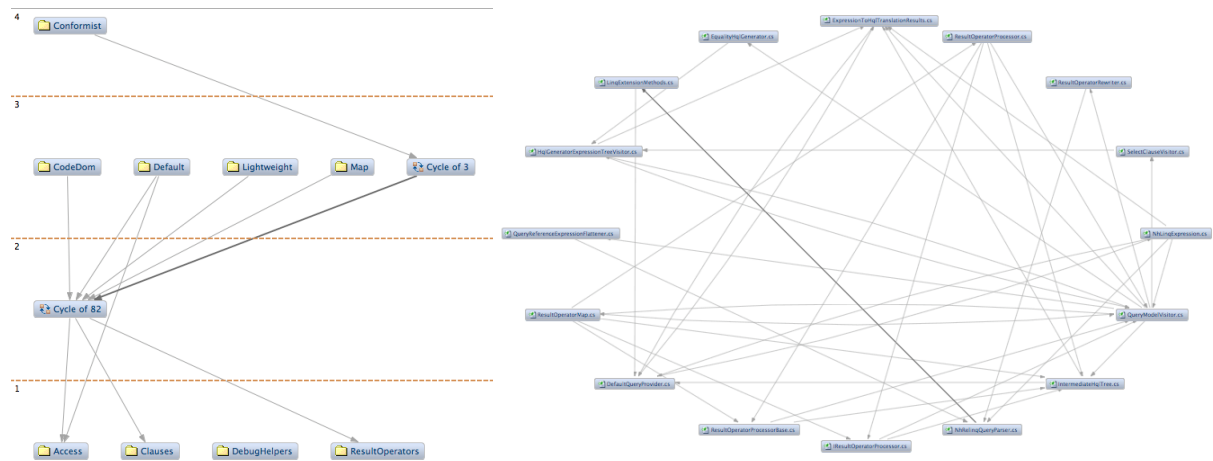


Figure 14: Layered graph view (left) and separate cycle viewer (right) of Sonargraph Explorer [91]

One source of dependency analysis technologies which has not yet been addressed is that of IDE plugins. The CodeMR program is available as a plugin for both the IntelliJ and Eclipse IDEs, in the form of a commercial product as well as a free version, which exhibits stricter limits on the size of project which can be analysed etc. The CodeMR tool makes particularly effective use of colour and shape as indications of coupling and code quality in its directed graph view, as illustrated in Figure 15. Degree of complexity exhibited by a module is visually represented to great effect by the gradient of colour from green (low) to red (high). Meanwhile, the degree of coupling is indicated by the number of vertices in the shape of the node itself, with a circular shape indicating low coupling, whilst a star shape, with a higher number of vertices, indicates maximum complexity [93]. The CodeMR program also offers a wide range of different graphical representations, in addition to the edge node format shown in Figure 15, these include hollow pie chart formats, sunbursts and treemaps, all of which present a different perspective on the quality indicators of the codebase.

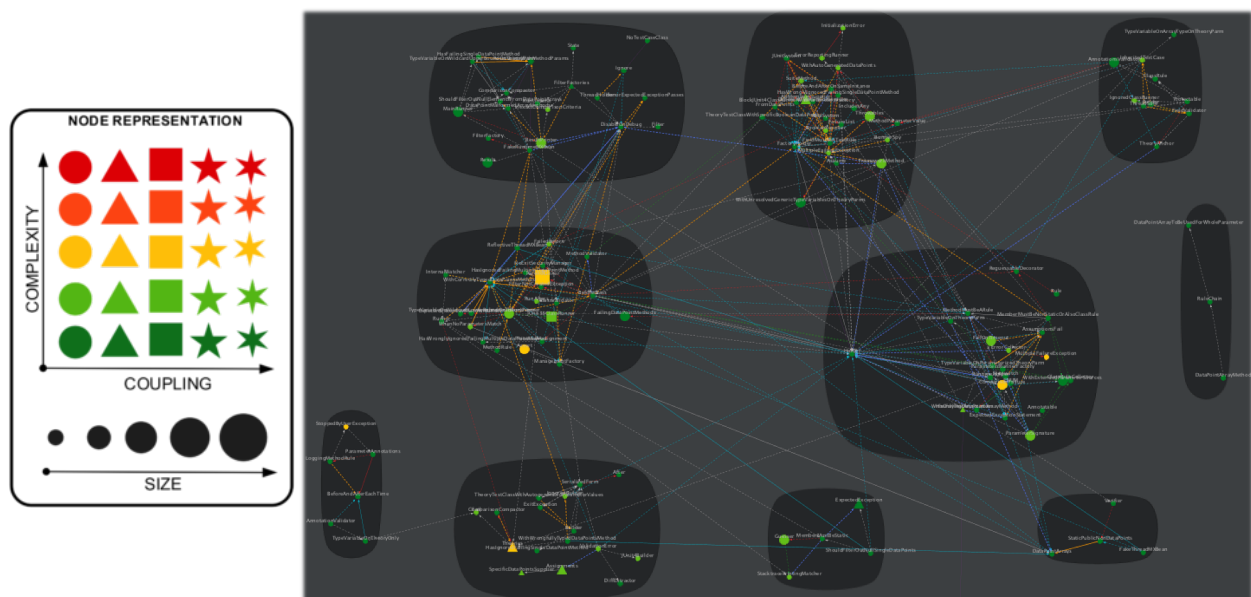


Figure 15: CodeMR Edge node dependency graph (right) illustrating cohesion and coupling between modules within an application along with visual class statistics key (left) [93]

2.6 Conclusion

The previous section has provided a summary of a number of the existing dependency visualisation techniques, from manual systems, to packages and technologies available both commercially and as freeware. Before providing such information, it was first necessary to provide an overview of the area of code analysis and static analysis tools in general, followed by the specific relation of dependencies to the important code quality gauges of coupling and cohesion. A brief overview of the area of information visualisation was then provided, with emphasis on abstraction, as relevant to this project.

This overview was followed by a conceptual introduction to dependency types, including practical examples. This allowed the actual mechanics of dependency extraction to be addressed, with reference to different techniques of interacting with the structure, such as lexical analysis and parsing. Following this, reference was made to some of the ways in which dependency extraction data could be refined and analysed, introducing the DSM and the concept of dependency patterns

Lastly, the display section of this literature review introduced edge node graphs, both visually and as a mathematical concept. It then went on to discuss a number of the different display tools which may be used for the purpose of dependency graphing in particular. The separate elements represented by these sections were addressed in the logical order in which they are utilised by a codebase dependency visualisation system. This allowed them to be united in the previous section, as existing examples of such a system and will now allow exploration of the nature of the specific dependency visualisation system which is proposed within this report.

A further benefit of this review is that it provides a clear outline of the benefit and use case for this system, with a major element being the provision of a freeware codebase dependency analyser without limits on size and functionality. In addition, it seeks to combine some of the features of other technologies, such as: edge node graphing; dependency structure matrix provision; presentation of different views or slices of source code; the use of visual parameters such as thickness and colour to denote quality and cycle analysis.

Ultimately, this is in line with the stated aims of the project to develop a prototype codebase visualisation system and determine its potential for implementation within the Driver and Vehicle Licensing Agency as a more developed static analysis package. It is intended that the prototype system will be capable of extracting the basic dependency structure from a codebase for subsequent display. This is achieved through implementation of the following objectives, firstly: to analyse and determine best practise techniques for the implementation of a codebase visualisation system written in java via mind-mapped research and a literature review, with focus on relevant existing technologies within the broader field of static analysis tools. Secondly, to use the agile Feature-Driven Development methodology to design and implement a software engine suitable for discerning the method and order in which different packages (and, with further development, classes) within a java codebase invoke, and are invoked within, each other. This may be done using parsing tools for the java language in combination with the autogenerated files such as the program structure interface from intellij. The configuration used to represent the derived dependencies will be stored via a suitable persistence structure. Thirdly, to similarly develop a platform-agnostic visualisation system capable of displaying sourcecode dependency data in an understandable format. This is intended to take the form of a directed graph system, with each node representing a module, and each edge representing a module invocation. Initially, online tools will be used to verify the effectivity of the engine. Subsequently, a pipeline will be constructed to allow auto-generation of diagrams as a direct result of the dependency parsing operation. In addition to this, the benefit of utilising inbuilt java graphing libraries will be considered. Fourthly, to explore the capability of the system to produce further alternatives to the

graphical representation of the source-code structure, via the inclusion of such features as a dependency structure matrix. The possibility of identifying cyclic relationships and representing the cohesion and coupling characteristics of the codebase in a more quantifiable manner. Finally, to determine the potential of the prototype system in facilitating the software development process via practical research. This will involve the participation of DVLA employees in a series of interviews and/or questionnaires centred around the visualisation system. It will be important to include input from the perspectives of a range of different roles, such as: developer, business analyst, agile delivery manager, etc. It is hoped that in this manner an effective prototype system may be developed and that ultimately some benefit might be provided to the wider workings of the DVLA.

3 Methodology

This section aims to provide an in-depth description of the processes the project utilises, the underlying reasons for why these procedures were chosen and the specific nature of their implementation.

3.1 Research Methodology

The project was formulated as a consequence of challenges directly observed by the researcher within the DVLA software engineer working environment. As such, it bears some resemblance to a case study, as it is research designed to be relevant to a particular sphere of a specific organisation. However, although the DVLA environment is very relevant to the progress of this research project, no attempt is being made to investigate the workings of the DVLA as an organisation. Rather, the project seeks to produce primary, phenomenological, inductive research into the potential of a self-built software tool within the given workplace.

3.1.1 Philosophy

The project outlined in this report can be viewed as widely phenomenological in nature although it can also be observed that there are elements of the positivist philosophy contained within it. This is because, although the software system that will be created as part of this project is real-world and non-subjective, a significant portion of the attributes that define that system are intrinsically subjective. The reason for this is that concepts such as “usability”, “usefulness”, “intuitiveness” etc, which are all features of the structural visualisations used in this project, lack inherent objectivity. This fact helps inform the choice of philosophy used in the project, i.e. pointing towards one which take into account the qualitative over quantitative nature of the data in question. The phenomenological philosophy and inductive approach are very appropriate from this perspective, with information-rich data being gathered via surveys as opposed to experiment, for the sake of developing a theory, instead of trying to substantiate or dispel an existing hypothesis. One potential challenge this raises is the fact that, ultimately, the project must seek to answer a well-defined question: “is there potential for further implementation of this prototype at the DVLA?”. Although the background of this question is intrinsically subjective, it is the responsibility of this project to produce, in response to that question, an answer which is as objective as possible. This dual nature will be addressed through the collation of qualitative interview data with more quantifiable usability questionnaire results, all based around the created software system.

3.1.2 Strategy

The project is predominantly inductive in its application of logic and reasoning. The justification for this is that the process started with a question: “is there a better way to get an understanding of codebase structure than simply reading through the source-code?” rather than a theory, indicating an inductive rather than deductive approach. The subsequent flow of the project confirms this by taking advantage of the established inductive pathway, moving to “how can we design and produce a more effective way of visualising codebase structure within this particular environment?” and then to the question “how much potential is there for expansion of this prototype at the DVLA?”. Although this provides a lot of flexibility and the opportunity to attain a very information rich process, one potential challenge of not starting with a well-defined deductive hypothesis, would be possible “creep” of the project in attempting to fulfil new goals and going beyond scope as a result of this inductive process.

3.1.3 Approaches

The three approaches used in this project are a literature review in combination with the development of real-world software and a survey approach.

3.1.3.1 Literature Review

Examination and exploration of published literature, both on the background theory of the subject and existing research and technology attempting to address the same issue, is invaluable to any research project of this nature. Specifically, with regards to this project, the literature review will provide guidance and scope for the development of the coded prototype, allowing selection of software features and providing a method for both integration and comparison with existing technologies. Furthermore, due to the self-instigated nature of this research project, and the relative lack of initial knowledge on the area, both from the perspective of the author and the DVLA in general, the literature review will provide an essential introduction to the area and a way of coming to grips with a new area of study and development.

3.1.3.2 Software Development

As the motivation for this project lies directly in the needs of the author, as a new developer, seeking to come to grips with DVLA codebases, it is natural that it must ultimately include the creation of a software tool to address these needs. In other words, it is a real-world problem which necessitates a real-world solution. The ultimate goal is to provide proof of concept for the possible future introduction of such codebase visualisation tools at the DVLA itself, as well as providing a positive insight into code structure, potentially allowing for improved code comprehension and quality. Furthermore, it provides the opportunity to expose real DVLA engineers to the prototype system and extract direct and meaningful feedback from them.

3.1.3.3 Survey

Pending completion of a working software system, results will be gathered via a combination of interview and questionnaire data. This will enable collection of information-rich, qualitative data along with a more quantifiable, although still subjective, data. The System Usability Scale (SUS) provides an effective method of estimating such data, whilst the use of video, audio or handwritten recording of interviews, as permitted, will be useful in retaining key information. In addition, the use of Likert-type scales will provide an appropriate method by which some program specific parameters, such as potential, can be quantified.

3.1.4 Risks & Limitations

It was possible to identify two key risks before engaging in the project outlined in this report. Firstly, it was crucial that implementation of the project be executed in an ordered series of discrete stages. Were this not the case, one might quickly face the very significant risk of being overcome by the potential magnitude and scope of the project, facing difficulties in identifying an appropriate starting place. Furthermore, it is essential that the questions posed as part of the interview phase are well planned out and possessed of a sound logical flow and structure, or else one risks the research process ending up too disjointed with ungeneralizable results. Finally, time is a key limitation that must be considered whilst completing this project. It would be very possible to ignore the potential impact of the very substantial percentage of time allocated to software engineering work at the DVLA being scheduled simultaneously to the duration of the research project. As a novice software engineer in particular, the researcher must not permit himself to underestimate the length of time that may be required for him to accomplish key requirements of the programme, as such miscalculation may have a significantly disruptive effect on the project timeline.

3.1.5 Validity, Reliability & Generalisability

3.1.5.1 Validity

The general validity, or “soundness”, of the research project is thought to be satisfactory. Custom software will be designed and built from scratch to answer an original research question based directly on an observed problem. This helps to guarantee that the findings will be genuinely representative of the initial phenomenon. Furthermore, the survey will be carried out on DVLA software engineering and testing staff, who are uniquely placed to provide the most authentic and realistic information on the different aspects of the system.

3.1.5.2 Reliability

As has been mentioned, the nature of the research and the information being addressed as part of this project is subjective. However, care will be taken during the literature review process to ensure a well-informed software system is designed and implemented. In addition, survey participants will originate from a range of different roles, at different grades and within different development squads of the DVLA, to facilitate reproducibility. Furthermore, the survey will include established questionnaire formats to increase objectivity of results, providing reliability.

3.1.5.3 Generalisability

As discussed, this research project is targeted towards the DVLA and specifically based on the experiences of a novice software engineer working in the “service creation” area of IT services. As such, the produced prototype, and the survey evaluation of it, are tailored towards what will be effective in that specific work environment. However, the fundamental issue being addressed: demystification of large-scale codebases, is one which applies in a vast array of software engineering organisations. Furthermore, the developers interviewed can be viewed as representative of front and back end web developers and testers in general, with the research questions relating to any object-oriented codebase, rather than a specific subsection of developers. As such, it is hoped that the potential of the system as a codebase visualisation tool is wide ranging and applicable to the majority of developers.

3.1.6 Data Evaluation Methods

3.1.6.1 Data Collection

Qualitative and quantitative survey data will be collected through a combination of interviews and questionnaires. A small sample of 8 participants will be taken from fellow developers at the DVLA, in accordance with the phenomenological philosophy. These respondents will be contacted in person, or via the DVLA staff messaging options during work hours. If insufficient data is collected/insufficient numbers of participants are available, attempts will be made to reach a wider audience via members of the DVLA management and administration staff, who will have access to a wider range of potential respondents. A second test will be carried out with a smaller sample from the participants in the first survey to allow evaluation of changes and tweaks made to the system as a result of initial findings.

3.1.6.2 Data Evaluation

Much of the collected data will be qualitative rather than quantitative in nature and must be addressed as such. However, the use of numerical scaling as part of the questionnaires will enable more quantitative analysis, such as the calculation of averages etc, to be obtained from the results. Furthermore, the use of the SUS gives rise to a designated method of calculating overall usability as a number. This can be summed up by the following equation:

$$Usability\ Score = \sum_0^{10} 6 \left(\frac{1 + (-1)^N}{2} \right) - S(-1)^N - 1$$

Where N is the relative position of the question on the questionnaire and S is the score allotted as a response to that specific question.

3.2 Development Methodology

As this project is centred around implementation within the DVLA software engineering environment, which made the transition to agile software development methodologies some years back, it is logical that the methodology used in its execution follows a similar approach.

3.2.1 Software Development Methodology OR Other Relevant Major Methodology

The agile software development methodology chosen for use in completing this project was Feature Driven Development (FDD). It should be noted that a methodology closer to that used at the DVLA, such as scrum or scrumban, would not be suitable for this project as it is an individual as opposed to a team or squad process. FDD, however, shares a crucial aspect with these development methodologies, in its iterative, timebox approach to structuring workload. Not only is the concept of “sprints” familiar from DVLA practice, the segmenting of work prevents significant deviation from key requirements whilst allowing flexibility of scheduling and accurate estimation of timescales and achievable targets.

Particularly beneficial from the FDD methodology is the concept of constructing a feature list, which can then be ordered according to priority, bringing requirements out into the open. The FDD process of then building according to these features allows very rapid generation of a Minimum Viable Product (MVP). This will be crucial in avoiding the outcome of reaching the deployment stage of the product only to realise that there is a key flaw in the project design. However, one challenge presented by this methodology is the possibility of feature creep, where new requirements are found and added whilst the project is ongoing, leading to mismanagement of scope. To avoid this, new requirements should provoke a review of the ordered feature list, placed in the correct position relative to the priority of the existing requirements, then brought into a timebox along with other features in the quantised FDD fashion.

3.3 Resources

The execution of this research project will necessitate the use of certain hardware, software, information and human resources which are listed below

3.3.1 Hardware Resources

The hardware used will include a development laptop, as supplied by the DVLA, initially a Toshiba, later a MacBook Pro.

In addition, a camera/audio device is required to record interviews where acceptable. The inbuilt video camera on the Huawei Honor 9 smartphone will be used for this purpose.

3.3.2 Software Resources

When working on the toshiba laptop, the operating system is windows 10, when using the MacBook, macOS Mojave.

The GraphViz software package is used for graph drawing via an interface using the DOT markup language, this software is open source and available for download from the GraphViz website [75].

The JetBrains IntelliJ IDEA development environment will be used for the practical elements of the research project. A commercial license for access to this software is provided by the DVLA to all software engineers as required.

The JavaParser software toolkit will be accessed via an added dependency in the maven Project Object Model (POM) file. Other Java libraries such as Swing will be imported as necessary.

3.3.3 Information Resources

The university library and online library will be used to gain access to the sources required for completion of the literature review and for other information required during the project

The project supervisor is a key source of information with regards to project direction, project structure and report make-up. In addition, initial feedback on MVP features are screened by the project supervisor before carrying out user tests.

Documentation on existing code visualisation technologies will be used as a means of comparison throughout the project.

3.3.4 Human Resources

8 employees of the DVLA will be required to take part in the interview and questionnaire processes separately. These should be from a range of different technical roles, such as software engineer, tester and Software Developer in Test (SDET). These should ideally be working on the tachograph project and related repositories, so as to enable standardisation of testing. They will be contacted individually, in person and through consultation with their line manager to ensure the research process does not interfere with their scheduled work.

The foregoing sections detail the underlying approaches and features which will be taken into account in the completion of this research project, both from a conceptual and from a practical standpoint. Ultimately, each project is individual and cannot be wholly summarised by its classification in terms of philosophy, strategy etc. In addition, for projects such as this, which dwell on the borders between classifications, there is added complexity in summarising the key features as well as predicting the outcome and evolution of the project. However, it is hoped that this discussion of the methodology of project will be mirrored in the following design and implementation section and the overall research findings.

4 Design and Implementation

The following section focuses on the manner in which the development period of this project was planned and carried out. The approach to documentation was predominantly timebox driven, with, on average, week-long timeboxes during which a set of requirements were pulled in, a method, order and structure of execution was decided, following which they were implemented. The design and implementation framework was kept fluid, in keeping with agile principles, and new features, which came up during sprints, were not shied away from but brought into subsequent timeboxes and addressed in accordance with the FDD methodology.

However, the implementation phase was not approached blind, with requirements being solely discovered and addressed on an ad hoc basis. Prior to embarking on the sequence of timeboxes, key functional and non-functional requirements were identified and formalised so as to define the scope and borders of the project. Advantage was taken of the period following the initial literature research phase for the project, as an opportunity to establish the features of the proposed prototype and categorise them as necessary, ideal, possible and unfeasible using the MoSCoW (Must have, Should have, Could have and Won't have) method for requirements analysis [94]. This resulted in an ordered list of requirements, separated according to priority and then further subdivided according to which of the three main concerns of the system (extraction, analysis or display) they related to, as shown in the following section.

4.1 MoSCoW Requirements

As discussed throughout the report, the dependency analysis system can be broadly addressed in terms of the three sections mentioned above. Although this is valuable in terms of separation of the physical layers in the architecture of the project, it also can be extended into the process of developing a requirements list. Considering the requirements of each section as separate from each other prevents the different concerns from becoming entangled with one another. The key below identifies the colour scheme which can be used to distinguish between requirements that correspond to the different layers of the project.

Key:	Extraction engine	Analysis	Display
------	-------------------	----------	---------

4.1.1 Functional Requirements

MUST
The system must be able to extract dependencies between packages on a java codebase
The system must write these dependencies to a suitable storage mechanism in a set format
The engine must be separated from the display through a uniform interface
The analyser must unify all dependencies into a single value for each module
The display option must read from the interface format and display the resulting dependency graph
The graph must be directed, with at least a node edge representation of modules and dependencies

Table 3 - Functional Must Requirements

SHOULD
The graph should be capable of displaying the strength of dependency between modules
The graphing tool should be called automatically sequential to the extraction process
A DSM should be created representing the dependency structure of the codebase
The system should extract strong dependencies between classes in the same package

Table 4 - Functional Should Requirements

COULD
Colour coding or UML class diagram notation could be introduced to identify dependency types (will have to bear in mind multiple dependency contributions)
The graphing tool could include zoom functionality to present different views of the modules
Dependency strength could be analysed as a function of more subtle weighting factors than dependency addition
Dependency patterns such as cycles could be identified automatically
The system could extract all dependencies between classes in the same package

Table 5 - Functional Could Requirements

WON'T
The display won't include interactive graphing on the basis of runtime dependencies
The analyser won't include high level clustering techniques or Artificial intelligence pattern recognition
The system won't identify complex dependency relations from packages such as spring

Table 6 - Functional Won't Requirements

4.1.2 Non-functional requirements

The following non-functional requirements relate to the manner in which the functionality of the system is delivered, as opposed to specific instances of functionality themselves. Although they tend to be broader in scope than functional requirements, designating “how the system works” rather than “what it should do”, they may still be separated out according to the three primary concerns of the system, using the same colour key as for functional requirements, as shown below.

MUST
The system must be easy to use
The display must be somewhat intuitive and cannot be messy to the extent of incomprehensibility
The analyser must provide an accurate and direct representation of what is displayed visually

Table 7 - Non-Functional Must Requirements

SHOULD
The analyser data should also be intuitive and presented in a visually consumable manner
The system should be able to run requiring very little user input and effort
The display should be clean-cut and arranged so as to be visually simple and understandable

Table 8 - Non-Functional Should Requirements

COULD
The system could exhibit extendability as a result of a good level of modularisation
The display could exhibit uniformity as a result of standardisation

Table 9 - Non-Functional Could Requirements

As can be seen from the above tables, the MVP for the system can be broadly designated by the “MUST” requirements, which outline a visualisation tool which will take in a directory address local to

the machine on which the system is running, extract the “out-of-package” dependencies between java files in that directory, group them according to packages, then format them for display as an edge node graph. Subsequent, less fundamental, requirements represent additional features and functionality such as extracting and representing relative strength of dependencies, providing a DSM view of the dependency structure, going deeper into the structure with interclass dependencies or reorienting the display to enhance comprehension. The timebox approach aims to cover these requirements in an iterative manner, with the most fundamental features being addressed in the earlier timeboxes and with further features (driven by less imperative requirements) drawn in over the course of subsequent cycles.

4.2 Time box 1

Date – 25th October 2019

Duration – 12 Days

Completion Date – 6th November

4.2.1 Features

1. Create the framework of the project
 1. Design architecture
 2. Set up interfaces
 3. Plan out implementations with order
2. The system must write dependencies to a suitable storage mechanism in a set format
 1. Decide format
 2. Command Line interface
 3. Write to set file format
3. The engine must be separated from the display through a uniform interface
4. The system must extract dependencies between packages on a java codebase
 1. Write basic implementation of business logic
 2. Engine comes last
5. The analyser must unify all dependencies into a single value for each module

4.2.2 Design and Implementation

4.2.2.1 Use Case Description

Use Case Name: Extract Package dependencies
Stakeholders & Interests: End User
Brief description: End user is required to extract package dependencies for graphing and further analysis
Trigger: A codebase needs to be analysed on a package level to determine its dependency structure
Normal flow of events: 1. End user runs the application in terminal 2. User interacts with command line interface and submits a project directory location 3. Dependencies are extracted from the packages within that project 4. The dependencies can be written to a text file in a graph format 5. The text file data can be input into an online graphing program for visual representation
Sub-flows: S-1 1. The user may quit the application
Alternative / Exceptional flow of events: S-1a: 1. The system cannot retrieve files from the string entered 2. The system returns an error for null/empty string for directory

Table 10: Use case description for extraction of package dependencies

4.2.2.2 Use Case Diagram

The graphic shown in Figure 16 displays the user's interaction with the system, i.e. they input to the system a repository from which they wish to extract the package dependencies. The extracted dependencies are written to a file in the DOT format. The user can then paste this into an online graphviz portal to get an immediate graphical dependency visualisation.

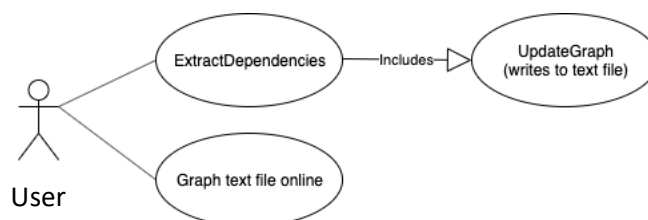


Figure 16 - Use case diagram for extracting dependencies

4.2.2.3 Sequence Diagram

The following sequence diagram in Figure 17 shows the flow through the system on a package by package level. As shown, the logic layer essentially coordinates the processes of the system.

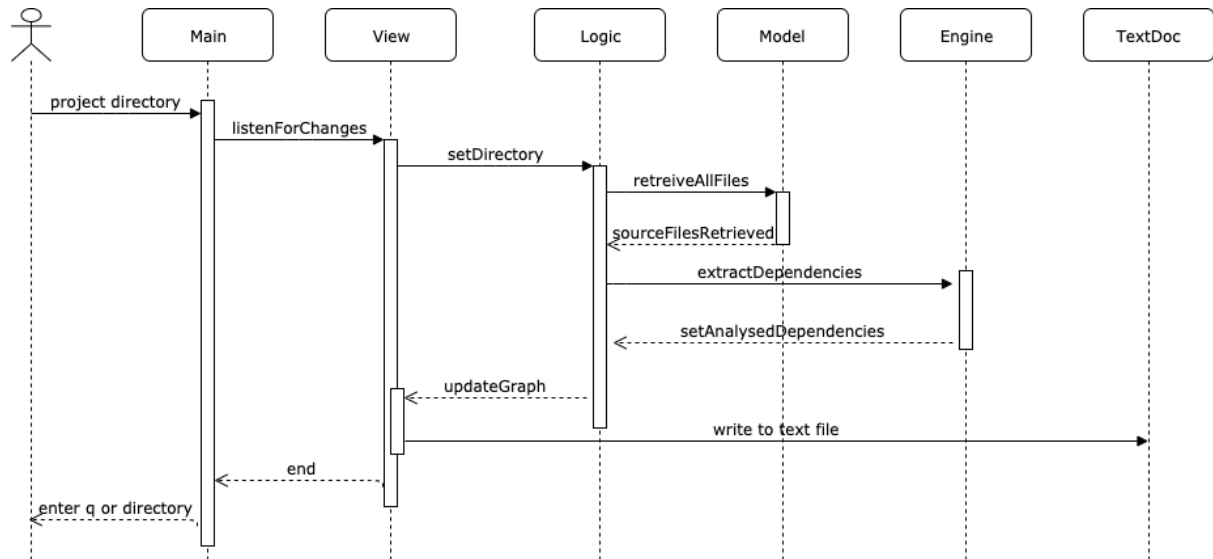


Figure 17: Sequence diagram for dependency extraction

4.2.2.4 Class Diagram

As shown in the UML class diagram format in Figure 18, the codebase has essentially been separated into modules, which only interact via their respective interfaces. A separate entities package (not drawn to reduce visual complexity) has been constructed which holds the bespoke objects indicated in the diagram. Cyclic dependencies are avoided through the separation of interfaces by direction of use as shown.

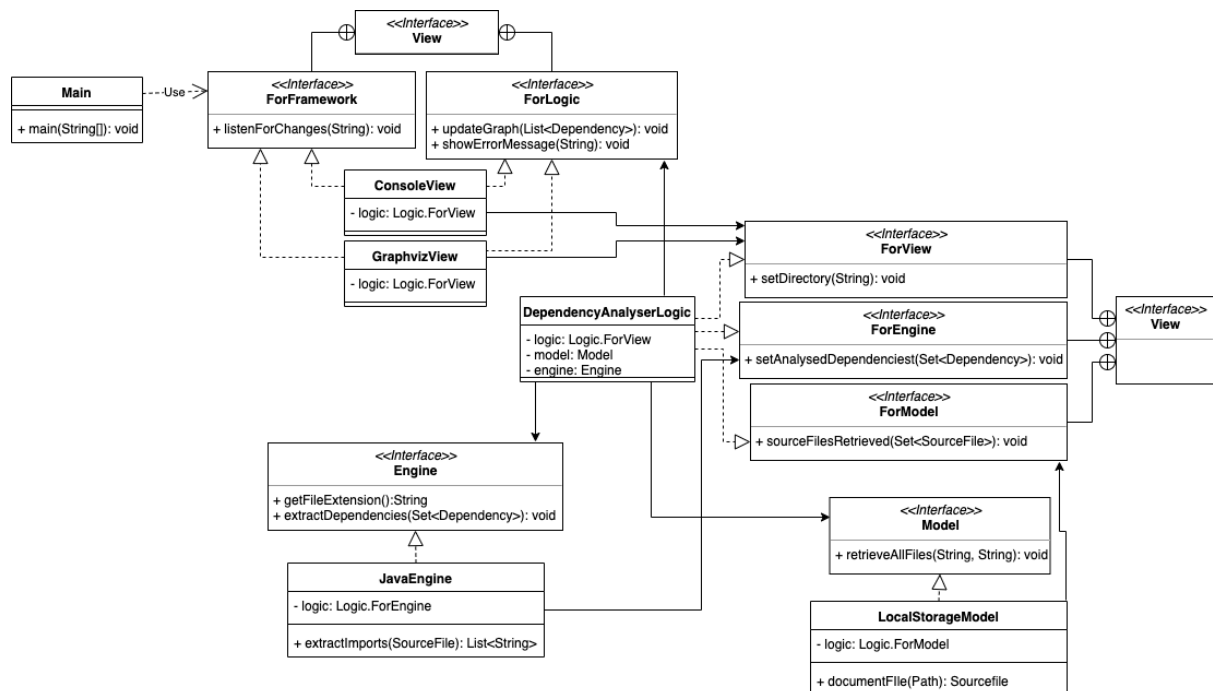


Figure 18: Class Diagram for dependency extraction

4.2.2.5 Timebox Overview

The system has been designed with the concepts of extendibility and separation of concerns as primary considerations. To this end, one of the first elements of the design considered was how to split up the functionality of the system into contained elements and how they might interact with each other. As exposed during the literature review, there are three main domains over which the system operates: the extraction mechanism, the analysis function and the display. These are split up in the codebase via the model, engine and view packages, with the model being responsible for physically interacting with the storage mechanism of the codebase to be analysed, the engine implementing many of the business rules in extracting the number and form of dependencies and the view responsible for front end interaction and presentation to the user.

In addition to these, 2 further elements were added, the entity package, which contains the objects which are used by different elements of the system and, crucially, the logic package, which is essentially responsible for coordinating all the other elements of the system.

Each of the elements discussed, excluding the entity package, has its own separate interface, in order to facilitate separation of concerns and allow modules to be swapped out easily, without implementation choice in one module having undue influence on the other modules. The logic interface can be seen in Figure 19 below:

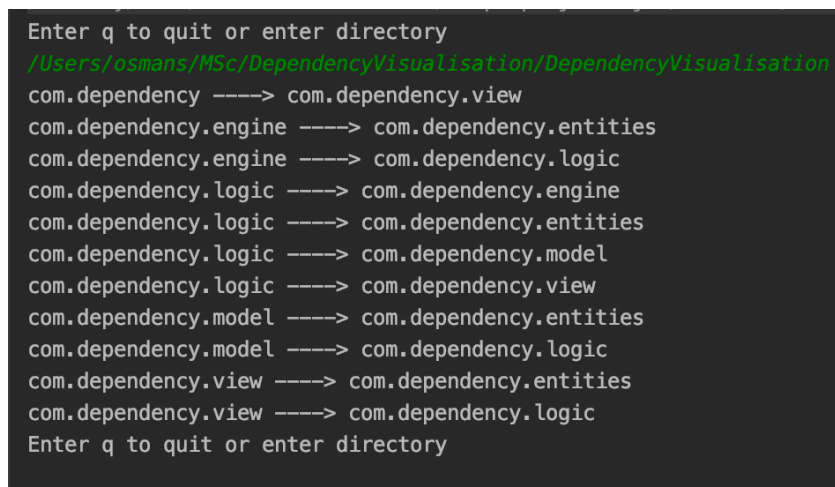
```
1      package com.dependency.logic;
2
3      import com.dependency.entities.Dependency;
4      import com.dependency.entities.SourceFile;
5      import java.util.Set;
6
7
8      public interface Logic {
9          interface ForView {
10             void setDirectory(String directory); // step 2
11          }
12
13          interface ForModel {
14             void sourceFilesRetrieved(Set<SourceFile> files); // step 4
15          }
16
17          interface ForEngine{
18             void setAnalysedDependencies(Set<Dependency> dependencies);
19          }
20      }
21  }
```

Figure 19: logic interface as represented within source code during timebox 1

It should be noted that the interface has been split up into a series of three discrete sub-interfaces. This is to allow true separation between the different modules and avoid circular dependencies. In other words, the each of the other modules addressed in the interface as shown, will only interact with their specific sub-interface of the logic, as opposed to the whole of the logic itself.

The current design of the system places the logic in a position comparable to that of a traffic policeman on a three-road intersection. As shown in Figure 17, which provides a graphical depiction of the logical flow within the system, the view passes a directory, as entered by the user, into an instance of the logic strongly typed as the “forView” interface. This is then passed by the logic into its instance of the engine, which passes back the extracted source code files via its instance of logic “forModel”. The logic then passes these files through to the engine for dependency extraction, with the extracted dependencies being passed back via the engine’s instance of the logic, typed in accordance with the “forEngine” interface. Finally, the logic passes the extracted dependencies back to the view for display via its view “forLogic” typed instance. In this way, the logic layer allows any of the constituent modules to be swapped out for different implementations of their same interface with only one line of code needing to be altered. This enhances separation of concerns, by limiting the impacts of changes to any module, whilst also facilitating extendibility, as new forms of functionality can be added, for example for different programming languages or different directory storage mechanisms, simply by producing a new implementation of the relevant interface

The user interface of the system was designed to be simple and is intended to remain so for as long as possible. The view module presents a command line interface as shown in Figure 20. A command line interface was considered appropriate and simple, as the vast majority of the target users for this system are back-end developers, who, at the DVLA, are often more familiar with command line interfaces for coding tools, as opposed to GUI. A standard directory location string is entered, with the initial return created as part of this timebox being a simple print line to console for each dependency as shown.



```
Enter q to quit or enter directory
/Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation
com.dependency ----> com.dependency.view
com.dependency.engine ----> com.dependency.entities
com.dependency.engine ----> com.dependency.logic
com.dependency.logic ----> com.dependency.engine
com.dependency.logic ----> com.dependency.entities
com.dependency.logic ----> com.dependency.model
com.dependency.logic ----> com.dependency.view
com.dependency.model ----> com.dependency.entities
com.dependency.model ----> com.dependency.logic
com.dependency.view ----> com.dependency.entities
com.dependency.view ----> com.dependency.logic
Enter q to quit or enter directory
```

Figure 20: Representative screenshot of the console interface with input and output

Subsequently, towards the end of the timebox, a different implementation of the view was instantiated, which wrote the output to a text file, as opposed to the console. Furthermore, this was a GraphViz oriented implementation, with dependency output formatted using in the DOT graph markup language as shown in Figure 21.

```
g {
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.model.LocalStorageModel;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.entities.SourceFile;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.Files;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.engine.Engine;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "java.io.PrintStream;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "com.dependency.entities.SourceFile;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/Model.java" -> "java.io.IOException;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.util.Set;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/Logic.java" -> "java.util.Set;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.Path;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/ConsoleView.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/Engine.java" -> "com.dependency.entities.SourceFile;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/Logic.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.attribute.BasicFileAttributes;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.Paths;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "com.dependency.entities.Usage;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "java.io.FileNotFoundException;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "com.dependency.entities.Usage;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.util.List;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/Logic.java" -> "com.dependency.entities.SourceFile;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/ConsoleView.java" -> "com.dependency.logic.Logic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "com.dependency.logic.Logic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.model.Model;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/ConsoleView.java" -> "com.dependency.logic.DependencyAnalyserLogic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.io.IOException;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "java.io.IOException;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/View.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.SimpleFileVisitor;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/Logic.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "java.util.Set;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.util.stream.Stream;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/Engine.java" -> "java.util.Set;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.Main.java" -> "com.dependency.view.ConsoleView;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "java.util.List;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.Main.java" -> "com.dependency.view.GraphvizView;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.file.FileVisitResult;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "com.dependency.entities.SourceFile;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "com.dependency.logic.DependencyAnalyserLogic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.util.ArrayList;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.view.View;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "com.dependency.entities.Usage;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.Main.java" -> "com.dependency.view.View;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.engine.JavaEngine;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "com.dependency.entities.Usage;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.logic/DependencyAnalyserLogic.java" -> "java.util.Set;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "java.util.HashMap;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "com.dependency.logic.Logic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.util.HashSet;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/ConsoleView.java" -> "com.dependency.entities.Usage;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/ConsoleView.java" -> "java.util.Map;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "com.dependency.logic.Logic;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/View.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/GraphvizView.java" -> "com.dependency.entities.Used;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.entities/SourceFile.java" -> "java.util.List;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.engine/JavaEngine.java" -> "java.util.stream.Collectors;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.model/LocalStorageModel.java" -> "java.nio.charset.StandardCharsets;"
"/Users/osmans/repos/DependencyVisualisation/src/main/java/com.dependency.view/View.java" -> "com.dependency.entities.Usage;"
}
```

Figure 21: Example of dependency extraction formatted in the DOT language

As shown in the above format, the initial implementation of the engine matched each class location to the classes outside of its package which it depends on. In order to more accurately represent the types of dependency being represented (i.e. package to package), this display was changed to unify dependencies into single packages, essentially decreasing the resolution to increase comprehension.

Further string manipulation was then performed within the engine to make the dependent and depended upon strings on the left-hand and right-hand side of the dependency structure displayed in Figure 21. This was necessary in order to allow the graphing software to understand that strings on either side of the dependency structure refer to the same node, as opposed to duplicate nodes being created in the graph. This then allowed the dependency structure to take a format similar to that shown in the printed output displayed in figure 5.

The resultant graph format was then taken from the text file and pasted into an online portal which <http://www.webgraphviz.com/>. This produced a graph of nodes and edges in accordance with the input as shown in in Figure 22.

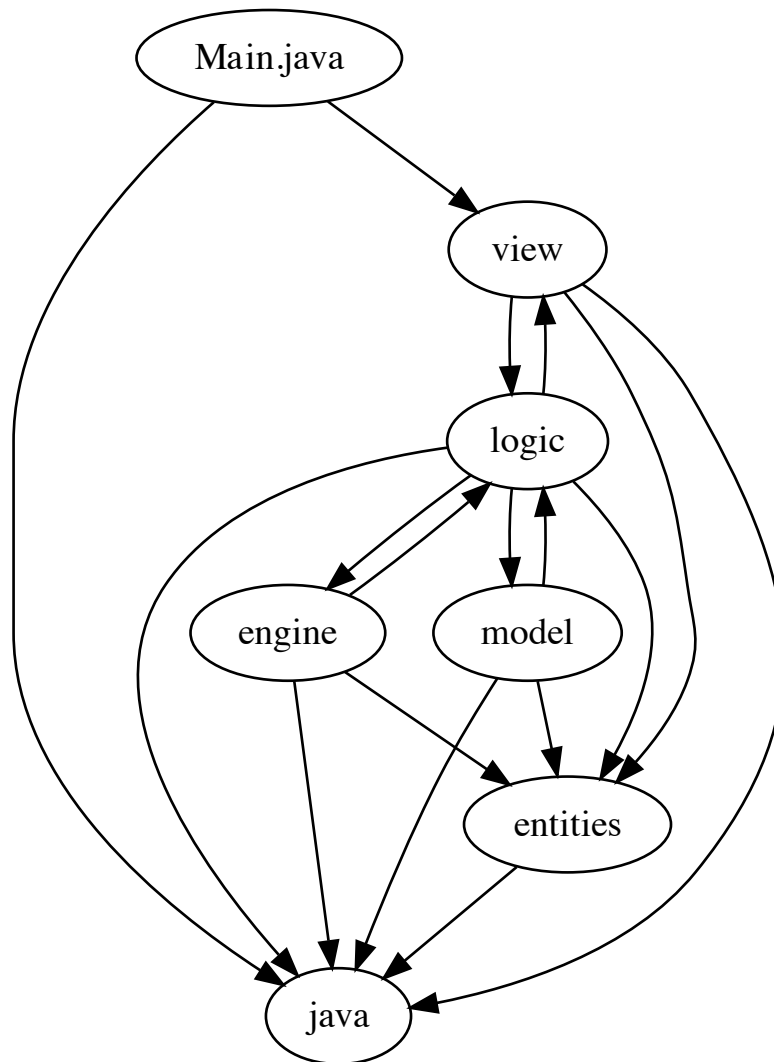


Figure 22: Package dependency graph of the package dependency extraction system created by itself

It can easily be seen from the graph that the project has maintained its separated structure, with the engine, model and view packages never invoking each. Instead, they all go through the logic module. This can be seen as evidence of the effectiveness of the system in correctly representing the package dependency

However, when trying to extract dependencies from significantly larger projects from the DVLA system, the returned dependencies were not representative of the system. This is because the method for slicing packages from strings in the engine is not currently capable of handling multi-level package projects with different depths of packages within each module.

4.2.3 Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Design write skeleton of system with interfaces	Structure of system defined	Structure of system defined		
2	Implement interfaces to produce working version of project which extracts dependencies and writes to a text file in the DOT format	System extracting package level dependencies from its own directory and writing them to a text file	System extracting package level dependencies from its own directory and writing them to a text file	Extraction not working for large projects with multiple irregular package layers	The method for slicing off the layers of classes/packages in this version of engine needs to be reworked Package dependency strength is uniform, should be taken from number of invocations
3	Graph the text output visually to ensure accuracy	Graph provides accurate visual representation of project	Graph provides accurate visual representation of project	Graph is too complex for large projects	Lower res view should be possible for large projects via slicing Basic Java dependencies can be removed from view

Table 11: Timebox 1 test sheet

4.2.4 Evaluation

All tasks were completed, however, the timebox had to be extended by a period of 48 hours. In addition to completing the features laid out in this timebox, completion of portions of the literature review was being addressed simultaneously. The first half of the timebox was dedicated to producing the structure of the dependency system and setting up all the interfaces etc. The second section was dedicated towards producing implementations of these structures. The first section took somewhat longer than anticipated due to the conceptual complexity of setting up the system in such a way as to be easily extendible.

The second section, implementation also required a somewhat larger time requirement than anticipated, partly due to the need of the developer to familiarise themselves with key elements of the java language such as Sets types, streams and overriding object methods. Ultimately the features were successfully delivered, but a key point for the imminent future was revealed, in that the dependency extraction engine needs to be reworked so as to be applicable to all projects, independent of size and package distribution.

4.2.5 Requirements Addressed

- The system must be able to extract dependencies between packages on a java codebase
- The system must write these dependencies to a suitable storage mechanism in a set format
- The engine must be separated from the display through a uniform interface
- The analyser must unify all dependencies into a single value for each module
- The graph must be directed, with at least a node edge representation of modules and dependencies

4.3 Timebox 2

Date – 6th November 2019

Duration – 5 Days

Completion Date – 11th November

4.3.1 Features

1. Refactor the extraction engine code to base dependency names off package structure rather than file structure
2. System should produce meaningful package dependency information irrespective of codebase size or package distribution
 1. Begin to introduce functionality for viewing codebase at different resolutions by slicing packages
3. Strength of dependencies should be visible as opposed to uniform weighting between all packages
 1. Extract number of dependency invocations to mapping structure
 2. Display in graph view using parameters such as line width
4. DSM should be produced as a way of representing quantified dependencies between internal packages
5. Remove unnecessary java imports node to increase diagram clarity

4.3.2 Design and Implementation

4.3.2.1 Use Case Description

Use Case Name: Display strength of dependencies between packages for any size of codebase	
Stakeholders & Interests: End User	
Brief description: End user is required to view dependencies between packages in terms of the number of invocations	
Trigger: The dependency structure and strength of dependencies between packages needs to be understood	
Normal flow of events: 1. The user enters a directory to analyse 2. Dependencies are extracted according to package structure, with number of instances of each package to package dependency passed through in map 3. The numerical dependencies between only internal packages are selected and displayed to user in simple matrix form via console	
Sub-flows:	
S-1	The user may view the numerical results not in matrix form but in dependency list form via console view
S-2	The user may view the dependency strengths in graph form via graphviz view
Alternative / Exceptional flow of events:	
S-1a: 1. The dependency structure of a large project may not be accurately displayed	

Table 12: Use case description for displaying dependency strengths

4.3.2.2 Use Case Diagram

The graphic shown in Figure 23 displays the user's interaction with the system. A number of different possible functionalities are available. The option exists to either have the dependencies printed to the console in simple text format, or in dependency structure matrix format or, as before, written to a text file in the DOT format. It should be noted that in the current format these separate use cases are not carried out simultaneously or in an automated sequence.

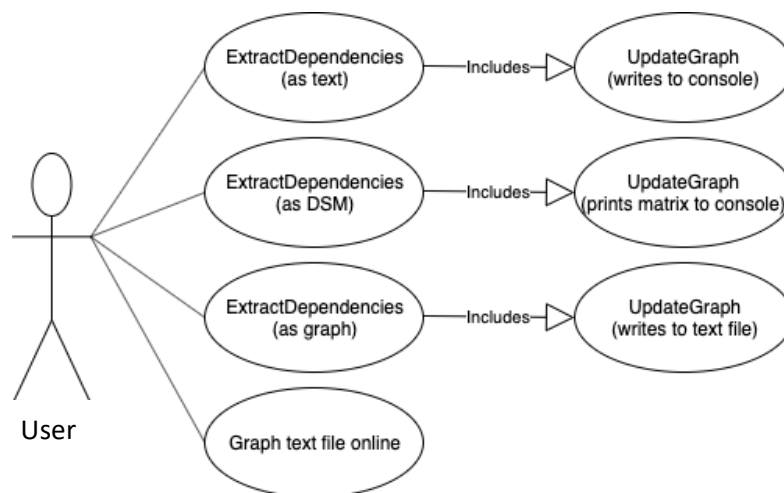


Figure 23: Use case diagram for displaying dependency strength irrespective of package size

4.3.2.3 Class Diagram

Figure 24 shows a UML class diagram of the updated system, after system alterations to return numerical dependency strength and handling of larger codebases with multiple packages. Code changes which have caused a visual change to the class diagram are shown in green.

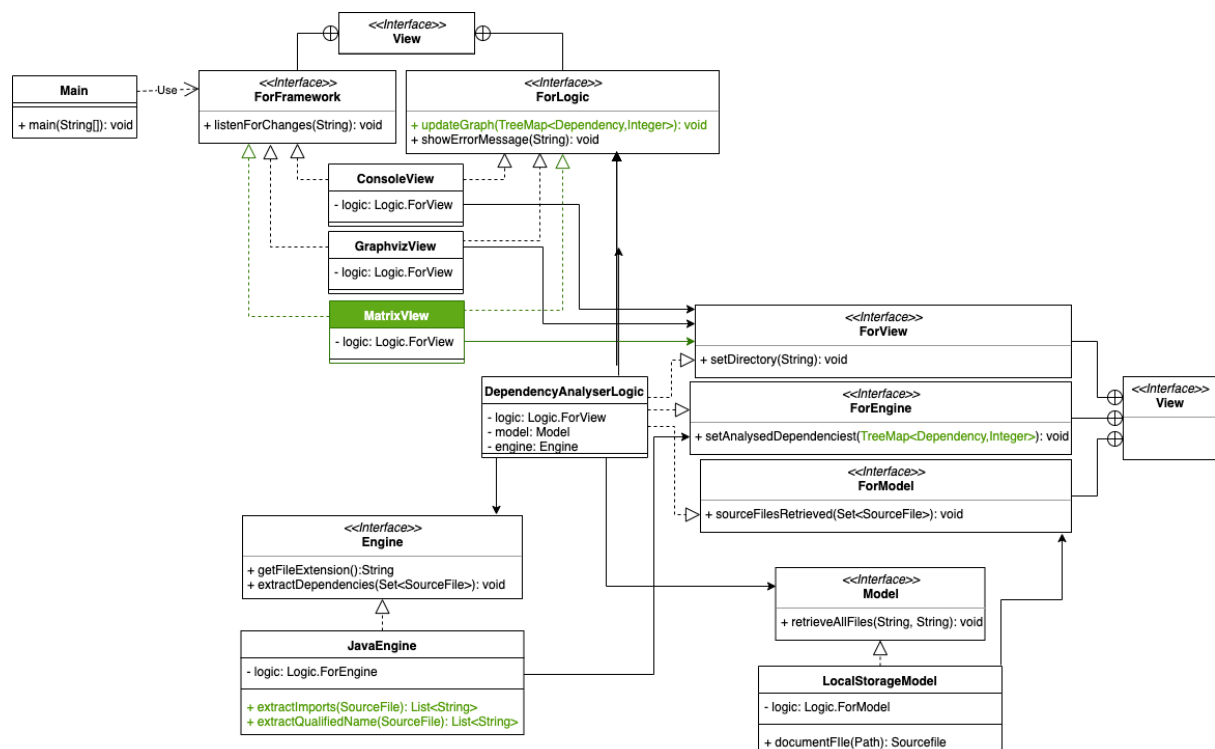


Figure 24: UML class diagram for displaying dependency strength irrespective of package size

4.3.2.4 Timebox Overview

This timebox was significantly shorter in duration, at 5 days, being less than half the length of the preceding timebox. The initial period of the timebox required some refactoring of the code in terms of the data structures that were being passed up from the extraction engine. There was unnecessary conversion of sets to lists which was removed and naturally a map is a more appropriate data item to correlate numerical strength values to each dependency.

The items to be accomplished over the course of this timebox can essentially be grouped into two categories, in order: debugging, improving and expanding the code to allow extraction of dependencies from all project sizes and package distributions and increasing the functionality of the extraction process to return the number of times one package invokes another.

The former was accomplished primarily by changing the manner in which the extraction engine interacted with and identified each individual project file. Prior to this, packages had been identified by their memory location, as returned from the model layer. However, due to the nature of the java language, all classes and interfaces etc, at least all those relevant to a package extraction engine, contain the necessary information to construct a unique identifier of themselves. Namely, they have a package string preceding the body of the file and somewhere below this a class/interface etc. name, for example: “public interface Model” inside the package: “com.dependency.model” can be parsed to extract the name: “com.dependency.model.Model”.

The result of concatenating these two to produce a unique class identifier is that, firstly, one is able to produce a more reliable representation of the package structure, as although matching physical directory structure in memory with java package structure is very good coding practice, it is in no way guaranteed. Secondly it allows for uniform “slicing off” of the class level of detail, to provide the first surrounding package, as one can guarantee that every element has been extracted first to the level of java class.

This item was tested by producing a graph of the java package dependency structure of a very large internal DVLA system coded in java. The dependency analyser had previously been unable of providing extraction at the necessary level of detail. However, it was now able to extract this level of detail and provide similar extraction from smaller projects without the code needing to be changed in-between. The diagram of the extraction from the large DVLA workspace has not been included for security/sensitivity reasons and due to the inherently complex nature of the visual image.

As a follow-up or expansion of this functionality, a further feature, or the beginning of a feature was pulled in. This was to introduce the capability of degrees of “slicing” into the system. The refactoring of the code which facilitated the previously discussed feature, also opened up the possibility of allowing the resolution at which the package structure was viewed to be decreased via a set parameter. So, for example, if one only wished to interact with the larger packages, one could choose to remove an entered number of layers of package detail from each of the dependencies, thus unifying the nodes presented in a graph or modules presented in a matrix. Some success was had with this approach when trying to remove the first level of packages, with a noted decrease in the number of package dependencies returned, however, at higher levels, it was noticed that the number of dependency nodes returned to its original value, indicating that slicing had not taken place at these levels.

The second of the initial two planned items, the extraction of numerical dependency strength, was closely correlated with the introduction of the map data structure for storing dependency types, as discussed. With this in place, within the engine layer, it was possible to increment an integer value

each time a dependency of the same meaning, i.e. from the same package to the same other package, was detected. This was tested by the introduction of one of the features discussed as one of the requirements that should be met by the system, the provision of a DSM. This DSM would show whether the engine had accurately extracted the structure of a system as a numerical matrix printed to the console as shown in Figure 25.

```
[com.dependency, com.dependency.engine, com.dependency.logic, com.dependency.model, com.dependency.view]
[0, 0, 0, 0, 2]
[0, 0, 1, 0, 0]
[0, 2, 0, 2, 1]
[0, 0, 1, 0, 0]
[0, 0, 4, 0, 0]
Enter q to quit or enter directory
```

Figure 25:View of dependency matrix as output by system (top line indicates order of packages)

As the DSM produced was found to accurately represent the project structure, additional functionality was again pulled in to provide a representation of dependency strength in the graph view of the system, another element from the “should” section of system requirements which was introduced as a testing measure. It was found that it was possible to display a graph which accurately represented the structure of the dependency analyser project, with inter-package dependency strengths represented as shown in Figure 26.

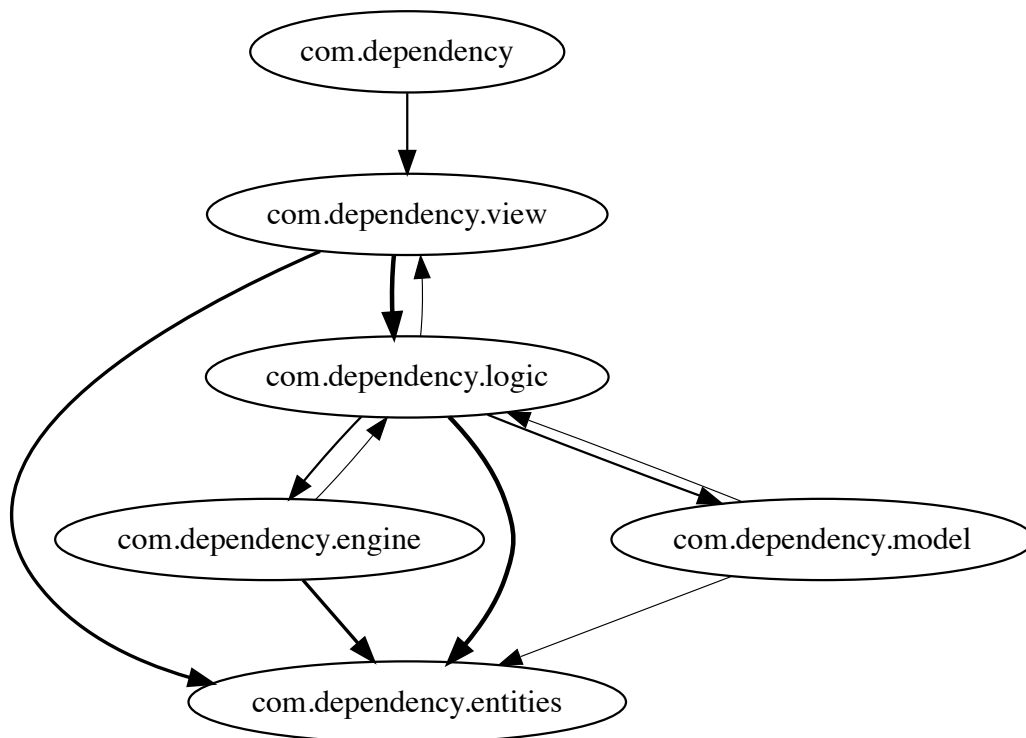


Figure 26:Package dependency graph of this system with weighted edges and no java library dependencies

In addition to this, some small functionality was introduced to prevent the extraction of dependencies between project files and basic java libraries such as java.util, java.io etc. This was to reduce unnecessary visual complexity in generated graphs, which can be seen when comparing the graphs of the project as shown in the documentation of the first timebox vs this timebox.

4.3.3 Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Refactor code to utilise more appropriate in memory structures	Dependency storage structure re-defined in terms of map	Dependency storage structure re-defined in terms of map		
2	Expand the code to allow extraction of dependencies from large projects with multiple layers of packages	System capable of extracting dependencies from deepest layer of packages in large DVLA system	System capable of extracting dependencies from deepest layer of packages in large DVLA system		Visual complexity of generated graph for large projects is significant The word static is now appearing in nodes for certain dependencies unnecessarily
3	Begin to expand to allow slicing of packages to provide different resolution views	Changing of in-code variable allows packages structure to be extracted and viewed at less depth	Initial decrease of package resolution shown, but not found when attempting to view upper levels of packages	Irregular package depths prevent uniform slicing of packages	Iterative approach to slicing instead of single conditional approach should improve results
4	Enable dependency structure matrix view of numerical strength of dependencies between packages	A matrix is produced which contains the correct numerical values of project package to package dependencies	A matrix was produced which contains the correct numerical values of project package to package dependencies	Only packages which contain a dependency on something are shown, those which are only depended on are not shown in matrix	
5	Enable graphViz view of numerical strength of dependencies between packages	A graph will be produced which contains the correct numerical values of project package to package dependencies	A graph was produced which contains the correct numerical values of project package to package dependencies		For larger projects the edges for packages with lower dependencies are too thin to be seen easily
6	Remove generic java library dependencies	No dependencies should be observed to java.util etc	No dependencies were observed to java.util etc		

Table 13: Timebox 2 test sheet

4.3.4 Evaluation

The primary goals of the project, to allow extraction of dependencies from large scale systems and be able to extract dependency strength and display it within a DSM, were achieved within the limits of the timebox. In addition, the documentation of the first timebox was expanded and additional features were drawn in from the requirements analysis and from new requirements that had come to light. The possibility of slicing the project to produce views of the package structure at different layers

of depths was functionality which had not previously been considered but was deemed as beneficial to explore and add in to the timebox, due to the high visual complexity of maximum depth dependency graphs for large systems. The decision to bring in the graphical representation of dependency strength was also made during the timebox, although it was a previously identified system requirement. The reason for pursuing these additional elements of functionality was twofold: firstly, there was sufficient time within the period to address additional features and secondly, both were natural extensions of the work previously covered within the timebox, the former as a result of changes to the extraction engine and the latter as a result of the extraction of number of instances of a particular dependency type.

For future work, it will be necessary to improve the package depths slicing functionality, making it iterative to avoid the situation where an attempt at viewing package structure at too shallow depths results in no slicing. The DSM graphing view must also be refactored to allow display of all internal project files, not only those with dependencies, and to allow graphical view and matrix view to be displayed simultaneously, instead of both representing separate view implementations. In addition, as discussed, the line thickness needs to have a lower limit to prevent problems with visibility. Ultimately the timebox was successful, with all scheduled features achieved and additional functionality drawn in and explored in accordance with the agile feature driven development methodology.

4.3.5 Requirements Addressed

- The system must be able to extract dependencies between packages on a java codebase
- The analyser must unify all dependencies into a single value for each module
- The graph should be capable of displaying strength of dependency between modules
- A DSM should be created representing the dependency structure of the codebase
- New requirement (not covered by MOSCOW approach) for slicing to view different depths of package structure

4.4 Timebox 3

Date – 11th November 2019

Duration – 7 Days

Completion Date – 18th November

4.4.1 Features

1. Ensure DSM and graph can be displayed simultaneously as opposed to separately
2. Ensure graphviz view is displayed automatically rather than requiring the user to copy and paste into separate graphviz portal
3. Expand slicing functionality to allow all packages to be sliced iteratively rather than on a simple conditional basis.
4. Ensure that DSM now contains and represents all internal packages rather than just ones which have some form of dependency on another package
5. Colour-code DSM as heat map
6. Ensure minimum thickness of edges on graph view, so that viewer can visually detect links, instead of allowing minimum thickness, rendering edges invisible on larger views
7. Ensure the “static” keyword is removed from nodes in graph view and from and dependency strings passed up from the engine

4.4.2 Design and Implementation

4.4.2.1 Use Case Description

Use Case Name: Display dependency graphs, with all edges visible, automatically with complete DSM	
Stakeholders & Interests: End User	
Brief description: End user wishes to view complete graph and dsm automatically after entering directory	
Trigger: The DSM and graph view of the dependency structure need to be viewed, in full, instantly after dependency extraction	
Normal flow of events: 4. The user enters a directory to analyse 5. Dependencies are extracted according to package structure, along with a set of internal packages, both sliced according to the chosen depth coefficient 6. The dependencies are displayed automatically, both as a graph with minimum edge thickness, and as a DSM for all internal packages	
Sub-flows:	
S-1	The user may view the numerical results not in graph matrix form but in dependency list form, along with a list of all packages, by running in console view
Alternative / Exceptional flow of events:	
S-1a:	
1.	If user slices package levels, they may misunderstand the structure of the codebase, as it decrements all package strings by one level, irrespective of length, instead of decrementing longer ones first

Table 14: Use case description for simultaneously viewing graph and matrix view of sliceable package structure

4.4.2.2 Use Case Diagram

The graphic shown in Figure 27 displays the user's interaction with the system. Two main paths are now available. The textual, console view provides a more limited, but less visually complex, means of viewing the dependency structure. Meanwhile, the graph and matrix view, having now been combined, lead to automatic generation of a graph image and a printed DSM.

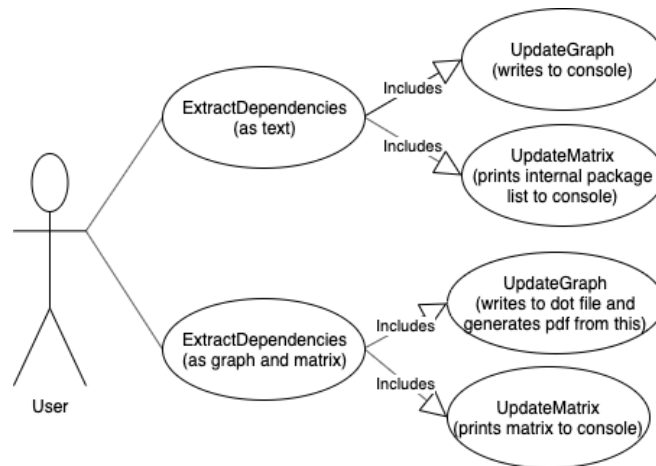


Figure 27: Use case diagram for automatic, simultaneous matrix and graph display

4.4.2.3 Sequence Diagram

The following sequence diagram in Figure 28 shows the flow through the system on a package level. As shown, the dependencies are now written to a .dot file instead of a text file (still in the DOT markup language). This is then fed into the installed graphviz viewing program to generate the pdf image corresponding to that graph. Subsequent to this, the dependency structure matrix is generated from the same information and printed to console. Visual changes to the sequence diagram since its last representation are highlighted in green.

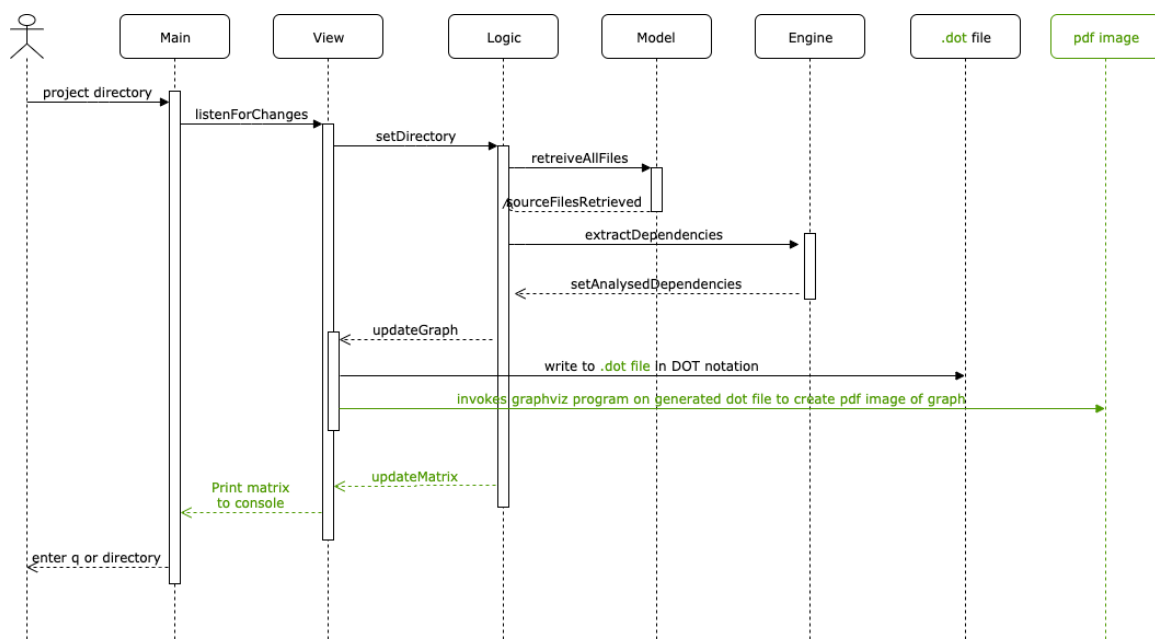


Figure 28: Sequence diagram for automatic, simultaneous matrix and graph display

4.4.2.4 Class Diagram

Figure 29 shows a UML class diagram of the updated system, after system alterations to return both a complete graph and DSM in a simultaneous and automatic fashion and improvements to the slicing method to allow all packages to be sliced iteratively. This can be partially understood by noting the replacement of the MatrixView implementation of the view interface with an additional function which can be called by a logic object on a view object, coupled with observation of the added functions to the JavaEngine class. Code changes which have caused a visual change to the class diagram are shown in green as always.

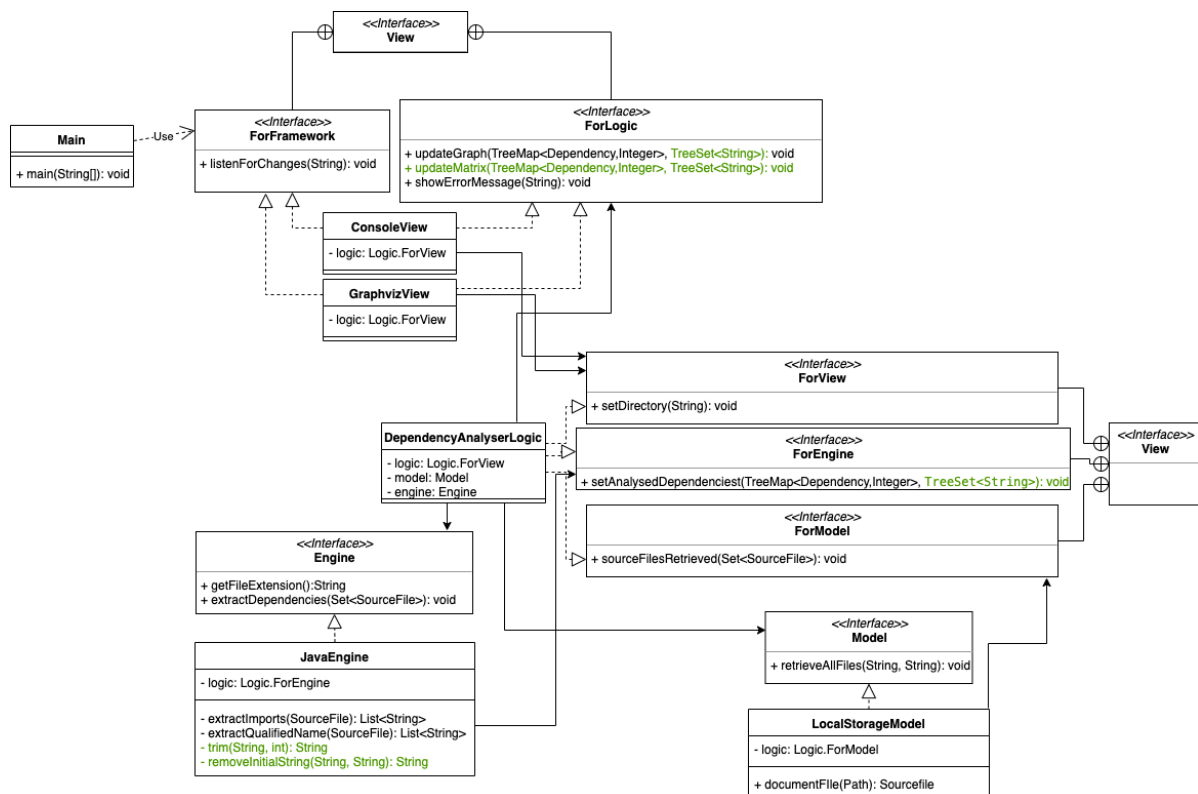


Figure 29: UML class diagram for automatic simultaneous display of complete matrix and graph

4.4.2.5 Timebox Overview

This was the first timebox of “standard length” being 7 days in length. It can in some ways be thought of as a “display timebox”, as many of the designated features of this cycle revolved around the visual aspect of the system, with the engine alterations being either geared towards this or being expansions or defect fixes on existing functionality.

The first feature to be addressed within this timebox was the “package slicing” functionality which has been explored and initiated in the preceding cycle. Test results from timebox 2 indicated that the introduction of an iterative, rather than a conditional “all or nothing” approach to stripping away package layers, would be more conducive to the extraction of staged changes to resolution of the package structure. This was done by simply extracting the logic necessary for implementing a slice decision on a package into a standard for loop and calling this function within the Java-Engine class. The result can be seen for the generated graph of the dependency analyser project displayed in Figure 30, with different degrees of slicing (no slicing, shown left, first-level slice shown right). It should be noted that, prior to refactoring, an attempt to slice more than a single level of package complexity for the codebase would have resulted in the graph shown on the left as opposed to the right, which would

have been expected. A remaining discrepancy between user expectation and the reduced resolution package view is found in the slicing mechanism itself. It is anticipated that, instead of intending a set maximum number of layers to be sliced off each package where possible, the user will expect to be able to set a maximum number of layers which should remain after the slice has occurred, this functionality remains dependent on future refactoring.



Figure 30: Comparing generated graphs for deepest layer of package detail vs with one layer of package detail stripped away

Key to the completion of the timebox was the successful introduction of automation into the graphing process. This was facilitated by first installing the graphviz software package on the host machine, an Apple MacBook, running macOS 10.14. The standard methods of interaction with the graphviz software allow commands to be entered via command line to generate image representations of DOT formatted graph data stored in a .dot file. To this end, the graph output file type was changed from .txt to .dot as shown in Figure 31, with the intention that this file would act as the interface between the dependency analysis tool and the graphviz program. The two were linked, via the use of a java command line access equivalent via Process and ProcessBuilder, as illustrated in Figure 31. An arbitrary sleep of 0.5s was introduced to ensure no overlap between the process for creating the pdf file, using the “dot -Tpdf {memory location} -o” graphviz terminal command, and opening it to display the image. Figure 30 provides a view of the screen which opens up to display the graph automatically when a directory is entered into the system. It should be noted that some view functionality is provided by this in the form of zoom in and zoom out buttons included in the default pdf viewer.

```
try(PrintStream ps = new PrintStream( "fileNames: "/Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation/TextTests/test.dot")) {
    ps.println(sb.toString());
} catch (FileNotFoundException e) {
    e.printStackTrace();
}
//Writing to dot file

String[] writeDotFileToPdf = new String[] {"/bin/bash", "-c", "dot -Tpdf /Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation/TextTests/test.dot -o " +
    " /Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation/OutputImage/outfile.pdf", "with", "args"};

String[] openPdf = new String[] {"/bin/bash", "-c", "open /Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation/OutputImage/outfile.pdf", "with", "args"};

try {
    Process proc = new ProcessBuilder(writeDotFileToPdf).start();
    TimeUnit.MILLISECONDS.sleep( 500);
    Process proc2 = new ProcessBuilder(openPdf).start();
} catch (IOException | InterruptedException e) {
    e.printStackTrace();
}
```

Figure 31: source code for writing to .dot file and then producing a pdf of that file using graphviz

Three other key features of this timebox related to the way in which the DSM view was created and viewed. As a result of work in the previous timebox, a DSM was created only for those packages which

had dependencies on other internal packages, not those which merely were “depended on”. Furthermore, due to initial construction of the DSM class as a separate implementation of the view interface, it was temporarily impossible to generate both the graph and matrix views simultaneously. The method for refactoring both of these issues is indicated by the code sample shown in Figure 32. The extraction of a separate set of qualified names for each internal package via the engine, enabled easy removal of the external dependencies from the DSM in the view layer. Furthermore, the addition of an updateMatrix function, negated the need for a separate MatrixView implementation, allowing both matrix and graph to be generated together from the same view implementation instance. An additional feature, colour coding the DSM as a heatmap, was not completed within the timebox, although progress was made towards understanding some of the different swing java GUI components which might prove helpful in displaying a more graphical DSM.

```
public interface View {
    interface ForLogic {
        void updateGraph(TreeMap<Dependency,Integer> dependencies, TreeSet<String> qualifiedNameSet);
        void updateMatrix(TreeMap<Dependency,Integer> dependencies, TreeSet<String> qualifiedNameSet);
        void showErrorMessage(String error);
    }
}
```

Figure 32: The updated View.ForLogic interface, indicating changes to the DSM creation process

Finally, two features which were essentially defect oriented, removal of the static keyword from the extracted dependencies and ensuring that there was a lower limit on edge thickness within the graphviz view, were completed. The former was completed through the introduction into the engine of a trim function which ensured only the keyword was removed, not possibly part of the class name which happened to overlap with it. The latter was a simple matter of extracting the maximum value between a set minimum visible edge thickness value and the graduated edge thickness proportional to the relative coupling strength of a given dependency within the graph.

4.4.3 Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Refactor code to ensure both graph and matrix can be viewed simultaneously	MatrixView class removed and functionality added in as method	MatrixView class removed and functionality added in as method		
2	Display graph automatically instead of requiring online paste	Install graphviz and invoke graph of dot file from code to create and open image	Graphviz installed and invoked on DOT file via java Process to create and open image		Potential problems of platform specificity, future expansions may require docker
3	Expand package view functionality to allow iterative slicing	Refactor slicing function to slice in increments, as opposed to simple conditional slice, allowing reduction of visual complexity	Slicing function refactored to allow slicing in increments, as opposed to simple conditional slice, reducing visual complexity		Slicing method, logically consistent but unintuitive to user, needs functional changes

4	DSM must contain all internal dependencies	Set of internal packages should be returned from engine and displayed in matrix	Internal package set returned and used to display matrix for both dependent and depended packages		
5	Colour code DSM as heatmap	DSM displayed visually with colour graduation for different dependency strengths		Java GUI window not opening, more work required	
6	Remove static keyword from dependencies	Nodes on graph of complex DVLA codebase should not contain "static" keyword	Nodes on graph of complex DVLA codebase do not contain "static" keyword		
7	Ensure minimum thickness on graph edges	Edges of graph of complex DVLA codebase should have a minimum visible thickness	Edges of graph of complex DVLA codebase have a minimum visible thickness		

Table 15: Timebox 3 test sheet

4.4.4 Evaluation

Of the features taken into this timebox, all were completed as intended, within the time given, with the exception of the colour coded display of the DSM as a heatmap, which was not completed. This was predominantly due to the developer's lack of experience with Java GUI libraries and tooling, which has been somewhat rectified over the course of the cycle. The feature from the new functional requirement, expanding the slicing capabilities, was completed as intended, however, the nature of the requirement has been rethought over the course of the timebox. To this end, it is intended that a new feature be included in the following timebox, reworking the slicing feature to fulfil a more intuitive functionality than the current implementation.

It should be noted that at this point in time, the majority of the "must" and "should" requirements from the MoSCoW functional requirements list have been addressed within the code. Furthermore, two of the "could" requirements have also been addressed within this timebox: the requirement for automatic calling of the graphing tool and for the addition of zoom functionality (by default of image presentation) as discussed within this timebox overview. The meaning and significance of this is that the overall MVP has been secured, in addition to further updated iterative and incremental releases of the system. The sole remaining "should" requirement is that of extracting strong java dependencies between classes within the same package. As such, it is recommended that, in addition to addressing the two features mentioned in the preceding paragraph, the next timebox begins the process of investigating and addressing the possibility of utilising JavaParser and JavaSymbolSolver for this purpose, with the intention of creating a new implementation of the extraction engine.

4.4.5 Requirements Addressed

- The graphing tool should be called automatically sequential to the extraction process
- The graphing tool could include zoom functionality to present different views of the modules
- A DSM should be created representing the dependency structure of the codebase
- New requirement (not covered by MOSCOW approach) for slicing to view different depths of the package structure

4.5 Timebox 4

Date – 18th November 2019

Duration – 7 Days

Completion Date – 25th November

4.5.1 Features

1. DSM should be displayed as colour-coded heat map using java GUI
2. Refactor and rework slicing functionality to allow the resolution at which the package structure is viewed to be decreased via a set parameter, which denotes maximum depth in terms of number of viewable package layers, as opposed to number of layers removed from the view
3. Begin practical investigation into JavaParser and JavaSymbolSolver libraries to allow future construction of a new extraction engine implementation which will be capable of extracting dependencies between classes which occupy the same package

4.5.2 Design and Implementation

4.5.2.1 Use Case Description

Use Case Name: Allows max package depth slicing and heatmap GUI DSM
Stakeholders & Interests: End User
Brief description: End user is required to set a maximum package depth for returned dependencies and view in a more intuitive colour coded DSM
Trigger: A specific maximum package resolution view of dependencies needs to be accessed, and relative dependency strengths need to be clarified as a heatmap
Normal flow of events: 1. The parameter to determine level of slicing is set 2. The user enters a directory to analyse 3. Dependencies are extracted according to package structure, being resolved to a maximum package depth according to the set parameter 4. Number of instances of each package to package dependency passed through in map 5. The numerical dependencies between only internal packages are selected and displayed to user via java GUI in the form of a heatmap colour coded DSM
Sub-flows: S-1 1. The user may quit the application by terminating the GUI window
Alternative / Exceptional flow of events: S-1a: 1. The user does not change the slicing parameter, so dependency structure is displayed at level of maximum package depth across each layer

Table 16: Use case description for allowing max package depth slicing and heatmap GUI DSM

4.5.2.2 Use Case Diagram

The graphic shown in Figure 33 displays the user's interaction with the updated system. As shown, a maximum package depth is set, to which all extracted dependencies will be sliced. After this, a number of different display forms can be used, one of which is a colour coded heatmap DSM via a GUI.

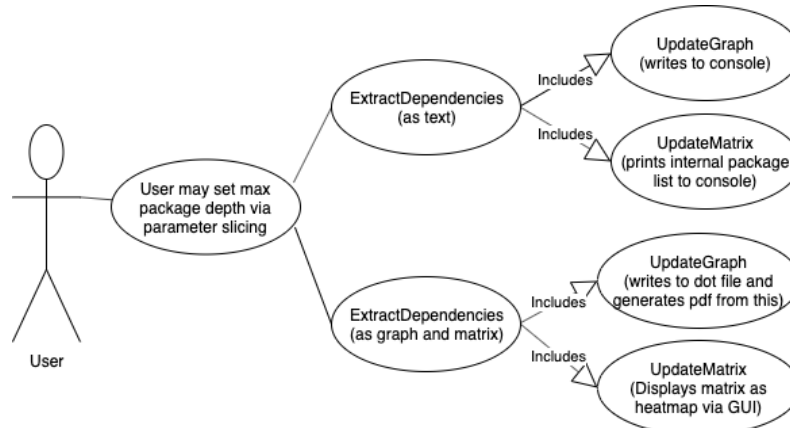


Figure 33: Use case diagram for max package depth slicing and heatmap GUI DSM display

4.5.2.3 Class Diagram

Figure 34 shows a UML class diagram of the updated system. As usual, modifications to existing functions (e.g. the trim function) and/or additions to the entity package (in which the matrix JTable class for the GUI is stored) are not displayed and changes to the class diagram (indicating exploratory JavaParser implementation) are highlighted in green.

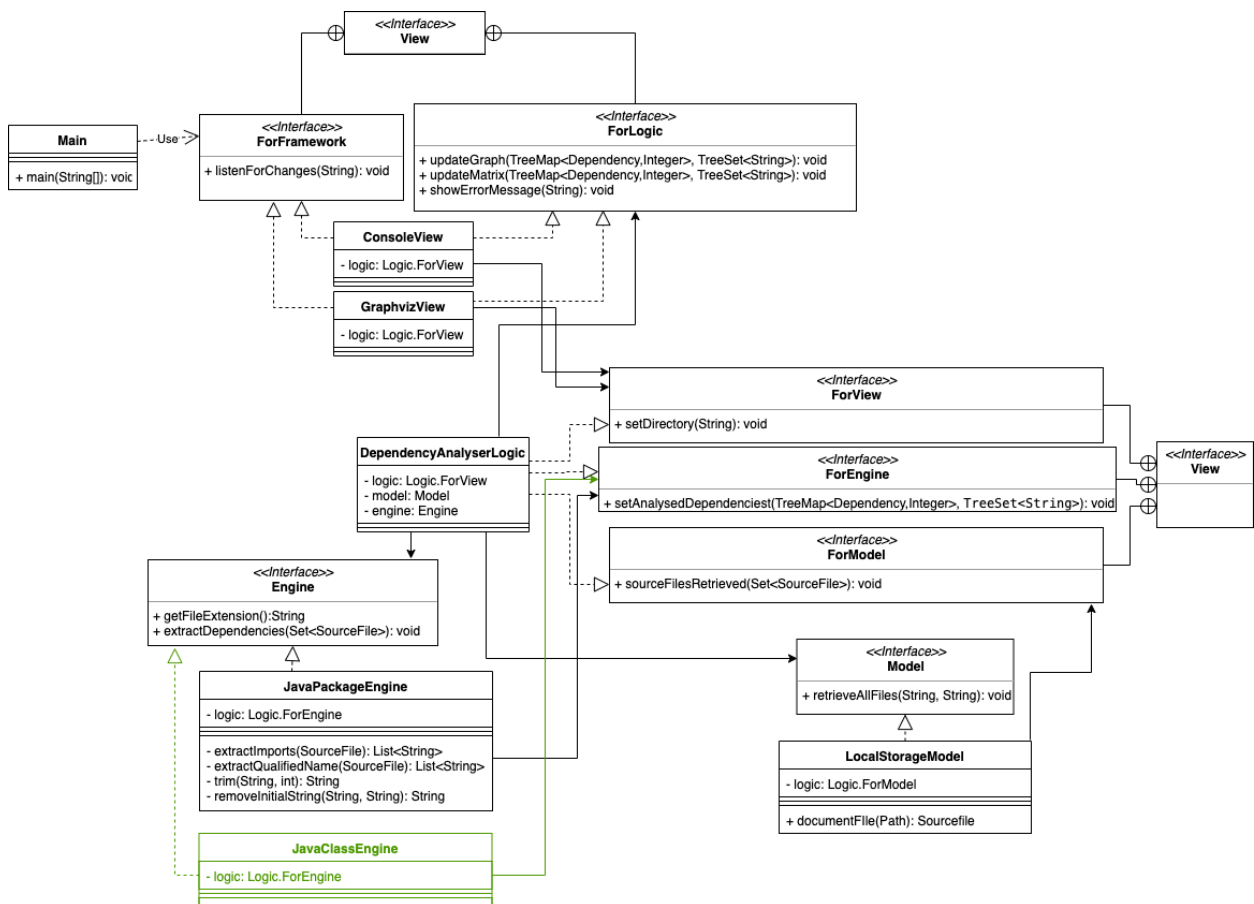


Figure 34: UML class diagram for max package depth slicing and heatmap GUI DSM display

4.5.2.4 Timebox Overview

The primary targets of this timebox can be split into three areas, improving the DSM display, changing the package depth slicing methodology and exploring dependency extraction via JavaParser. The DSM, whilst representing a more quantitative, less illustrated view of the dependency substructure than the graphical representation, can still be presented in a manner which allows easy visual consumption. One of the factors which was quickly brought to light with regards to the DSM was the possibility of presenting a more easily comprehensible format than the initial version, shown in Figure 25, through the use of a heatmap structure. This was one of the features introduced in the previous timebox, with the aim of using a Java GUI to produce the new DSM format. Work on this feature continued during this timebox, with the preceding cycle serving as an introduction to the java Swing GUI toolkit.

The heatmap style DSM was implemented using java's JTable and JFrame GUI components, with a 4-stage colour code system utilising different shades of the colour green. Dependencies were split into fourths according to numerical value (number of instances of a particular package dependency) with the bottom quarter corresponding to the lightest shade of green and the top quarter, representing the strongest dependencies, corresponding to the darkest shade of green. This is illustrated in Figure 35, which displays the DSM created for this dependency analysis system, without package slicing.

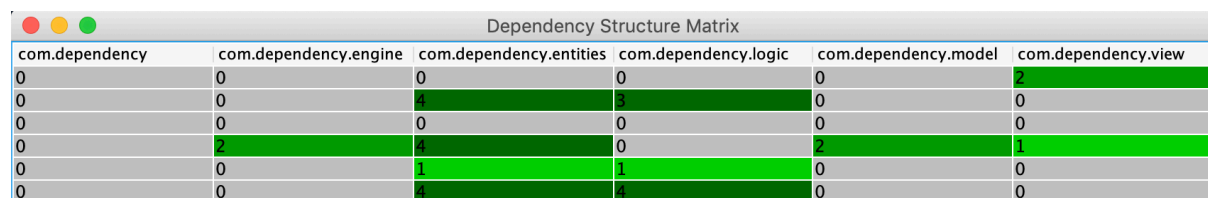


Figure 35: Colour coded DSM in heatmap form, automatically produced via Java GUI

One potential issue which was revealed as part of the testing process related to scale. For the image displayed in Figure 35, resizing of the GUI window was possible, allowing representation of the full package name for each column. However, when extracting dependencies for a much larger codebase, such as some of the DVLA workspaces, the DSM presents as shown in Figure 36. Due to its size, the entire DSM is not represented in the image below, however, as can be seen, the package labels for each column are not displayed, due to size limitations. Although it is possible to manually scale individual columns to a larger size, making the column labels readable, consideration should be given to scaling to this view. Possible improvements may involve some combination of, rotation of column labels by 90 degrees, trimming of shared strings between all package names, text wrapping on column labels, altering the horizontal scaling of the GUI window, etc.

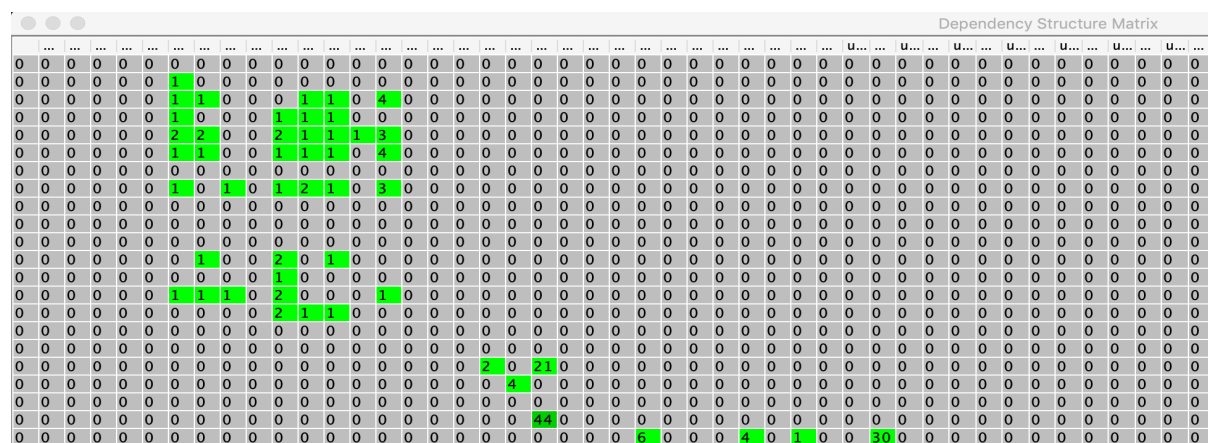


Figure 36: Segment of the DSM produced for large DVLA codebase

With regards to slicing package strings so as to present layered views of dependencies at different package depths, the previous implementation, one of setting a numerical parameter representing a maximum number of layers which would be sliced from each package identification string, was replaced by a parameter representing the maximum number of layers which would be displayed, for each package string. Therefore, with a parameter of 6, for example, the package strings displayed would reach a maximum of 6 layers in length, whereas before, this would imply that up to six layers would have been sliced off each package string. This presents a much more intuitive method for the user to interact with and understand the system and its outputs. This feature is demonstrated in Figure 37, which compares the dependency structure of the dependency analysis system before vs after being exposed to a slicing parameter which reduces each package string to a max depth of 2 layers. Note that the `javax.swing` dependencies from the additional GUI DSM feature are represented as a single node on the graph, as opposed to having been sliced to a separate `javax.swing` and `javax.swing` dependency, as the previous iteration of the slicing feature may have done.

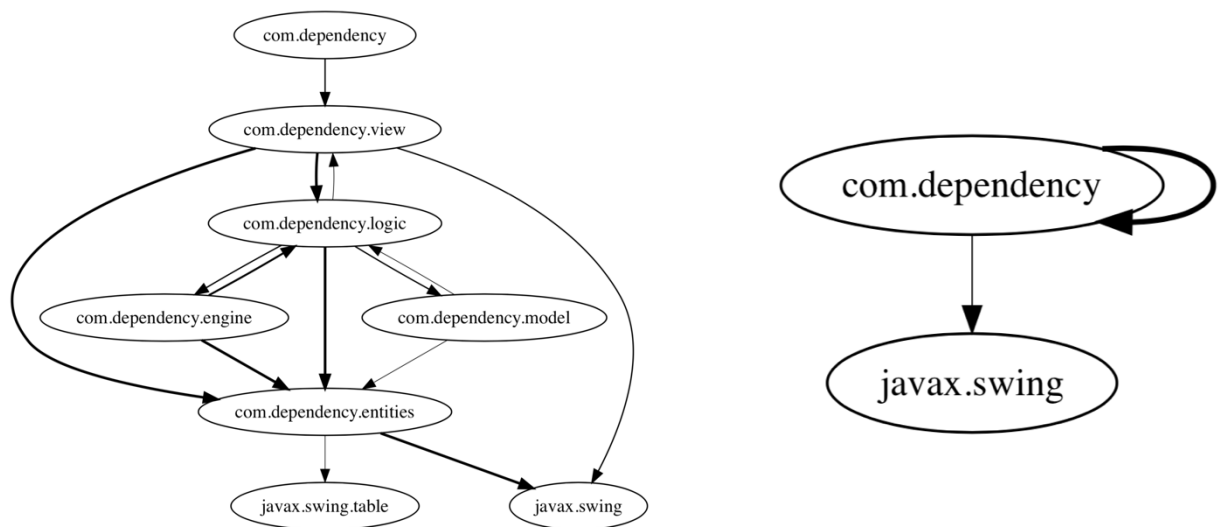


Figure 37: Dependency structure for system codebase displayed without slicing (left) vs with max package depth of 2 (right)

Finally, some initial exploration of the functionality behind the `JavaParser` and `JavaSymbolSolver` libraries was undergone, with the dependencies being added to the Maven Project Object Model or `.pom` file. This allowed some preliminary investigation of some of the features of this library, such as extracting the list of classes imported by a given class using `JavaParser` as opposed to textual analysis. A new implementation of the `JavaEngine` interface was also created, with the aim of using the `JavaParser` functionality to extract inter-class dependencies as opposed to inter-package.

4.5.3 Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Display DSM as a heatmap via a GUI	Colour coded DSM should automatically display in GUI upon extraction	Colour coded DSM displays automatically in GUI upon extraction	Scalability when viewing large projects	Scalability features need to be improved
2	Max package length determined by slicing depth parameter	Package identification strings should be sliced to length of numerical parameter	Package identification strings were be sliced to length of numerical parameter		
3	Preliminary exploration of JavaParser and JavaSymbolSolver features for the purpose of extracting class to class dependencies	JavaParser library added on and some of the features explored	JavaParser library was added on and some of the features were explored		Ideally more in-depth practical investigation would be carried out, however this can be addressed over the course of future timeboxes

Table 17: Timebox 4 test sheet

4.5.4 Evaluation

This timebox was set at a length of 7 days, with all features required within the goal of the cycle being delivered to at least an acceptable level. Exploration of Java GUI features from the previous timebox proved useful in producing a viable heatmap DSM in this cycle. As with all UI features, to some extent, there is room for improvement in the method of presentation, however, this should be limited to the most key aspects to prevent excessive time consumption in future sprints. The package depth slicing requirement can now be considered completed, other than the need to allow the user to edit the slicing parameter via command line. A number of sprints have been used for the purpose of addressing this requirement, predominantly because of a realisation, during the previous sprint, of a re-implementation which would handle user expectations in an improved fashion. This was addressed during this timebox, however, it initially occupied more time than expected, due to a convoluted approach being taken to the slicing parameter. Essentially this was chasing the possibility of slightly refined functionality at the expense of re-working a good portion of the codebase and the overall efficiency of the program. Successful completion of this timebox was contingent upon addressing the sunk cost fallacy of that implementation and instead delivering a workable feature via a less convoluted approach. The class to class dependency extraction functionality has not been delivered as part of this timebox, however practical investigation of the JavaParser libraries has begun. This is necessary, in addition to the theoretical research performed during the literature review, to better comprehend the forms and limits of functionality provided by the library. Although this exploration revealed new means of extracting package dependencies, it also emphasised the importance of writing a textual analyser from the ground up, to allow in depths understanding of the system itself.

4.5.5 Requirements Addressed

- A DSM should be created representing the dependency structure of the codebase
- New requirement (not covered by MOSCOW approach) for slicing to view different depths of package structure
- The system should extract strong dependencies between classes in the same package

4.6 Timebox 5

Date – 25th November 2019

Duration – 11 Days

Completion Date – 6th November

4.6.1 Features

1. Colour coding nodes based on number of other nodes with which they interact
2. Introduce functionality to allow removal of external packages from the graph view, with the user able to choose this option via the command line interface (CLI)
3. Package slicing depth parameter should also be possible to set via a CLI
4. Rework DSM view to have colour consistency with the new colour coded graph view
5. Introduce tooltips to DSM GUI to allow easy identification of dependency elements

4.6.2 Design and Implementation

4.6.2.1 Use Case Description

Use Case Name: Front end set to facilitate visual consumption
Stakeholders & Interests: End User
Brief description: End user is required to set a maximum package depth for returned dependencies and view in a more intuitive colour coded DSM
Trigger: A specific maximum package resolution view of dependencies needs to be accessed, and relative dependency strengths need to be clarified as a heatmap
Normal flow of events: 1. The user enters a directory to analyse 2. The user is prompted to enter set maximum package depth if they wish 3. The user is prompted to select visualisation features (here, external package removal) via entering a CLI code string 4. The dependencies returned to the view are decreased/unified in accordance with the values that the user has entered 5. The graphical dependency display will be colour coded, with increased saturation representing increased connectivity of a given node. 6. The DSM will be colour coded in a consistent manner with increased saturation for higher dependency values 7. Individual dependency elements of DSM will be identified by tooltip box
Sub-flows: S-1 1. The user may choose not to enter a package slicing parameter or remove external dependencies or may enter incorrect values 2. The default views will be created, without reducing number of visible dependencies
Alternative / Exceptional flow of events: S-1a: 1. GUI error after generation of DSM could confuse entry of subsequent input criteria to the visualisation system

Table 18: Use case description for rendering system output more visually consumable

4.6.2.2 Use Case Diagram

The graphic shown in Figure 38 displays the user's interaction with the updated system. As shown, the user has control over which type of view they wish to be displayed on an extraction by extraction basis. The graph produced will be sliced and have its external dependencies removed in accordance with whatever parameters the user may choose to enter at that stage, node colour saturation increases with connectivity of said node. Slicing parameter choice will also affect the DSM (from which external dependencies are removed by default). The DSM is colour coded in a manner consistent with the graph and allows the user to identify dependencies by hovering over the numerical elements on the graph to produce a tooltip.

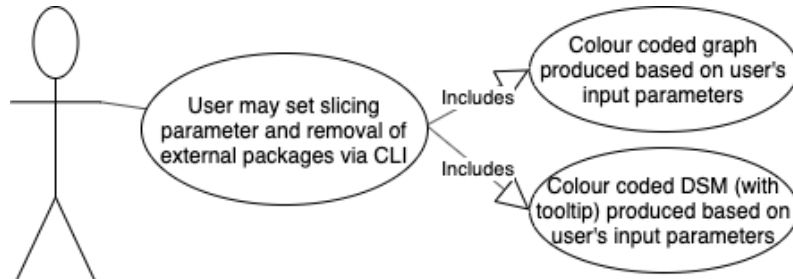


Figure 38: Use case for generating system outputs in a more visually consumable fashion

4.6.2.3 Class Diagram

Figure 39 shows a UML class diagram of the updated system. Input parameter additions, and their influences, are emphasised in green within the diagram. Changes which are not represented, due to not causing

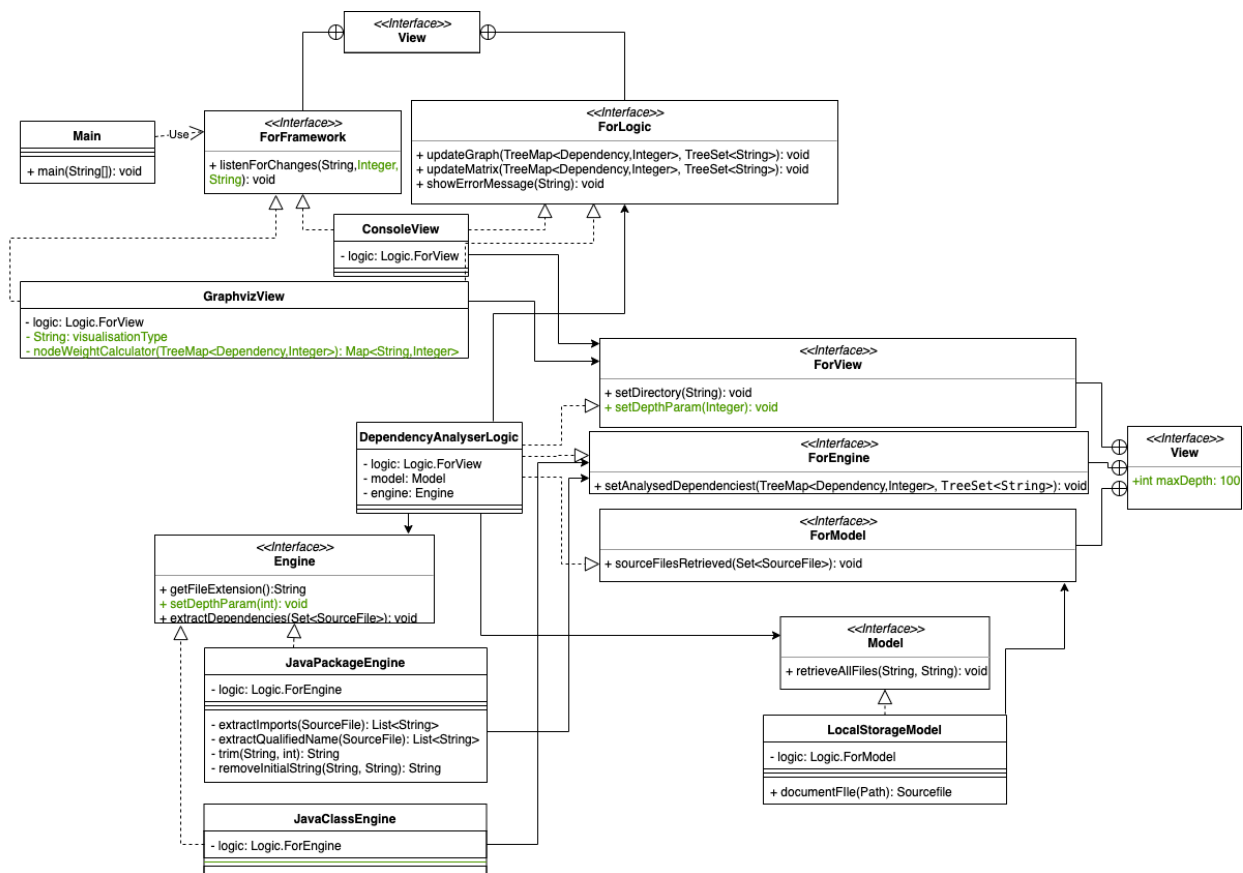
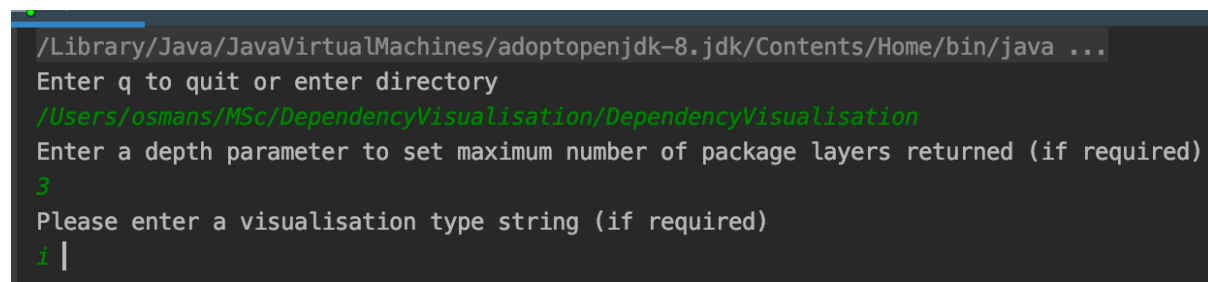


Figure 39: UML class diagram for generating system outputs in a more visually consumable fashion

a diagrammatic change or entity package changes include introducing a DSM table model entity and modifications to the graphviz DOT file generation.

4.6.2.4 Timebox Overview

The features of this timebox can essentially be split into 3 areas: granting the user CLI control of the format of returned dependencies, colour coding of visual output and addition of tooltip functionality to the DSM GUI. These were tackled in the order listed, with prompts for both the slicing parameter and the option of removing external dependencies (which was a new requirement which logically precipitated from the previous timebox) being set up for the user to input. Key concerns here included architectural issues, such as choosing the layer of the app in which these parameters should be handled. In addition, given the nature of user input, it was important to ensure these parameters were held in a suitable format and properly sanitised. The package slicing parameter was brought to the logic layer as an Integer (so as to be nullable in case where user chose not to slice in which case it was set to the maximum package depth value within logic) and passed on to the engine. The removal of external dependencies was intended to be entered as part of a potentially composite command string, with unique character codes for each function, i.e. “i” to only show internal dependencies as represented in Figure 40. Note that any string, or an empty string may be entered by the user, but external dependencies will only be removed when that particular character is present in the input. The presence of this code was then checked for in the subsequent view layer, based on which, external dependencies would either be removed from, or displayed with, the internal dependencies in a graph



```
/Library/Java/JavaVirtualMachines/adoptopenjdk-8.jdk/Contents/Home/bin/java ...
Enter q to quit or enter directory
/Users/osmans/MSc/DependencyVisualisation/DependencyVisualisation
Enter a depth parameter to set maximum number of package layers returned (if required)
3
Please enter a visualisation type string (if required)
i |
```

Figure 40: Representing command line entry of visualisation parameters

format.

Colour coding of the nodes of the graph was deemed necessary in order to emphasise “node importance” in an easily comprehensible manner. A function was created to assign a weight to each node by determining the number of other nodes it was connected to. The nodes were then assigned a colour based on an HSV (hue, saturation, value) numerical representation. The hue and value of the colour was set at those of a standard green (being a mid-spectrum colour to which the human eye is most receptive), whilst the saturation was varied to provide a graduation of shades [95]. In this way, a scale could be created whilst keeping node contrast constant, maintaining visibility, with the node of highest relative connectivity being possessing maximum saturation and the unconnected nodes having a saturation value of zero. This was facilitated by the GraphViz application, which includes functionality for setting node colour via an HSV code. The colour coding of the DSM was adapted, from its previous method of splitting into four colours, to follow the exact same process as the graph, with the same ‘H’ and ‘V’ values of 0.33 and 0.63 respectively, with the only difference being that in the DSM maximum saturation represents maximum number of instances of a single dependency. The colour coding for the graph and DSM views are shown in Figure 41 and Figure 42 for a sample large open source codebase.

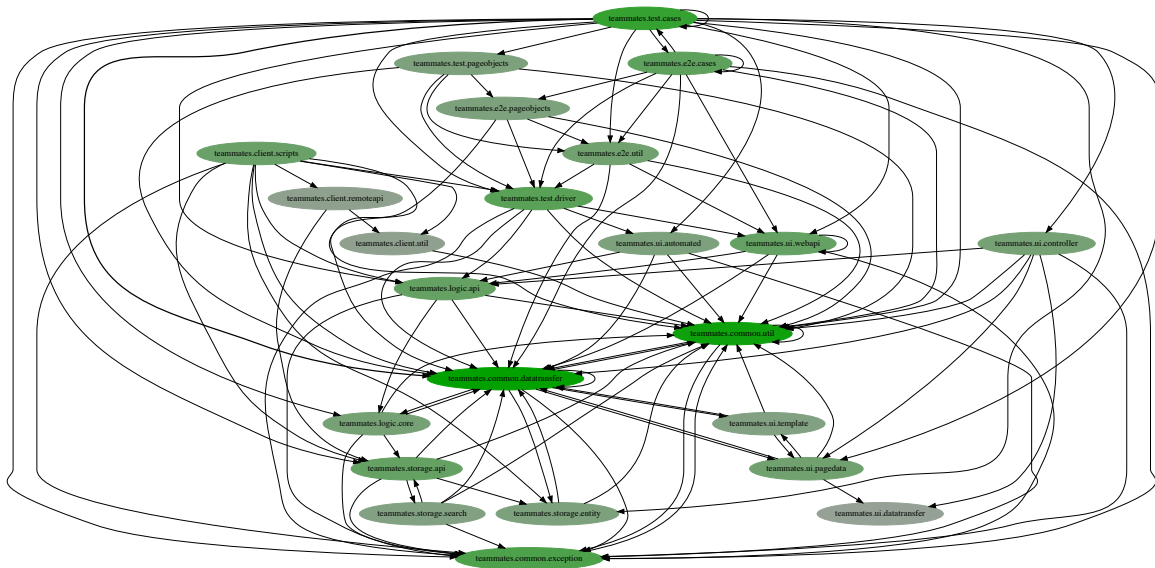
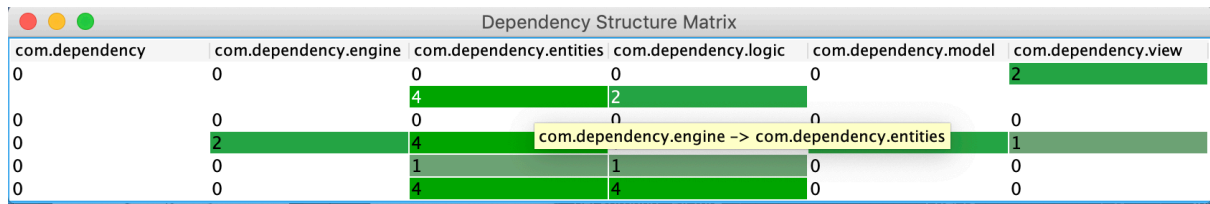


Figure 41: Internal dependencies of free open source peer evaluation tool 'Teammates' graphed to max package depth of 3 layers, showing colour graduations for connectivity

teammates...	teammates...	teammates...	teammates...	teammates...	teammates...	teammates...	teammates...	teammates...
0	0	1	0	0	0	0	0	0
3	0	2	7	2	10	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	113	2	166	0	0	0
0	0	0	0	0	1	0	0	0
0	0	0	8	8	5	0	0	0
0	0	0	27	4	23	1	14	13
0	0	0	1	0	3	0	0	3
0	0	0	13	0	4	0	0	0
0	0	0	37	8	23	0	0	0
0	0	0	88	31	44	0	0	0
0	0	0	24	23	28	0	0	0
0	0	0	4	0	6	0	0	0
0	0	0	22	1	18	0	0	0
0	0	0	521	107	396	33	0	19
0	0	0	13	2	14	0	0	0
0	0	0	4	0	29	0	34	3
0	0	0	20	19	43	0	0	0
0	0	0	22	10	42	0	0	0

Figure 42: Section of DSM for free open source peer evaluation tool 'Teammates' graphed to max package depth of 3 layers, showing colour graduations for dependency strength

Lastly, it was observed in the previous timebox that during the process of graphing large projects, it was easy for the column names on the DSM to become obscured. This is essentially a scaling issue, presenting all package the information for every dependency on a single screen becomes increasingly infeasible with increased matrix size. One solution to this is to provide dependency identification information on a discretionary basis, through the use of a tooltip. This enables the user to view the identity of any given dependency element as shown in Figure 43. The ordered nature of the dependency storage structures also helps the user interact with the DSM elements, by alphabetically ordering rows and columns.



4.6.3 Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Test colour coding for node connectivity on graph display	Dependency graph for codebases should exhibit graduation in colour saturation based on node connectivity improving visual comprehension	Dependency graph for codebases exhibited graduation in colour saturation based on node connectivity, improving visual comprehension		
2	Introduce functionality to allow user to remove external packages from returned dependencies	If user enters correct CLI code, generated dependency graph should only contain packages internal to the project	When user enters correct CLI code, generated dependency graph only contains packages internal to the project		Tech debt, logic for this should be moved from view to logic layer
3	Allow package depth parameter to be set via the CLI	Entering integer should slice packages to that depths, not entering should give max depth view as default	Entering integer sliced packages to that depths, not entering gave max depth view as default		GUI error can result in confusion of CLI
4	Test colour coding for dependency strength on DSM	DSM should exhibit graduation in colour saturation based on number of dependency instances, improving visual comprehension, consistent with graph	DSM exhibited graduation in colour saturation based on number of dependency instances, improved visual comprehension, consistent with graph		
5	Introduce tooltips to allow identification of DSM dependency elements	Hovering over one of the elements of the DSM should produce a tooltip showing the dependent package and package depended on	Hovering over one of the elements of the DSM produces a tooltip showing the dependent package and package depended on		

Table 19: Timebox 4 test sheet

4.6.4 Evaluation

This timebox was slightly longer than average at a length of 9 days, with all features required within the goal of the cycle being delivered to an acceptable level. The timebox was primarily front end focused, with emphasis on providing more control to the user via the CLI and improving the comprehensibility of output visuals. This is relevant and understandable, as the forthcoming phase of

the project will focus on usability testing and direct user research. Initial preparation for this phase was intended to occur in parallel with this timebox, however, due to conflation of workflows, this was not completed as planned, partially contributing to the increased length of this timebox. One of the features brought into this sprint, the removal of external dependencies, represented a new requirement which was identified during preceding timeboxes. This can in some ways be regarded as conceptually similar to the package depths slicing function, as it aims for the same goal, reducing visual complexity to improve user comprehension of codebase structure. Further features were drawn into the timebox in an organic fashion, for example, altering the DSM colour coding to be consistent with the graph node colour coding was a natural logical extension. Similarly, the feature to introduce a tooltip to the DSM, which was added at the end of the cycle, fits well with the ethos of this timebox in terms of seeking to improve user comprehension of outputs.

One matter which should be born in mind is the presence of a number of 'technical debt' items which have arisen over the course of this sequence of timeboxes, which involve the less apparent elements of the codebase, such as future extendibility, long term maintainability at scale etc. A number of architectural improvements can be made to the model with some refactoring, such as ensuring all non-language related and non-display related logic is handled in the logic layer, as opposed to the view or engine layers, introducing 'utility' classes for each layer to abstract functionality and prevent future code duplication etc. Issues such as these have been noted, however and will not be allowed to advance to the point of affecting the system. Code clean up and refactoring related to non-functional issues like this will be addressed on a rolling basis in parallel with future system updates/expansions, as has been the practise throughout previous timeboxes.

4.6.5 Requirements Addressed

- A DSM should be created representing the dependency structure of the codebase
- New requirement (not covered by MOSCOW approach) for slicing to view different depths of package structure
- Colour coding or UML class diagram notation could be introduced to identify dependency types (will have to bear in mind multiple dependency contributions)
- New requirement (not covered by MOSCOW approach) for removing external packages and only graphing internal if user selects this option

4.7 Timebox 6

Date – 6th December 2019

Duration – 12 Days

Completion Date – 18th December

4.7.1 Features

- Design list of tasks simulating those that would be completed by user in real-world version of system
- Perform cognitive walkthrough of tasks to ensure user may interact with all system controls whilst completing tasks
- Construct questionnaire from set of context-based questions to obtain data on user interaction and impressions
- Perform run-through of questionnaire with 2 volunteers, who will complete tasks whilst giving live qualitative feedback, complete specific questionnaire and more general SUS questionnaire.
- Pending successful run-through with first participants, stage interviews and questionnaires with a further 6 volunteers to collect further data

4.7.2 Design and Implementation

This section addresses the manner in which the content and format of the user task list and questionnaire were developed and subsequently administered. The completed version of each can be found in Appendix A – User Task List and Appendix B – User Questionnaire.

4.7.2.1 *User task list and cognitive walkthrough*

The planned structure of the user task list was a series of four sections, the first three being highly structured tasks with detailed, step-by-step instructions, which take the user through each of the key elements of functionality of the system. The fourth task would be more free-flowing and unguided, allowing assessment of the users understanding of the system as a whole, whilst also opening up a more exploratory method of testing, possibly allowing new qualitative data to be brought to light.

The format of the first three tasks was to guide the user through each view they have of the system, starting with the memory location of 3 separate java code repositories on the local machine used to execute the tasks. It should be noted that, although the tasks attempt to simulate normal user interaction, there are inherently artificial elements to them, as, in a real-world scenario, the user would be entering the memory address of a known, pre-existing repository on their machine, which they wish to probe the structure of.

The 3 tasks were designed to increase in both the complexity of user interaction and complexity of examined codebase. The cognitive walkthrough of the tasks was used to ensure that each feature was being introduced in the most intuitive fashion. For this reason, the first task examined the smallest codebase (which was the visualisation system's own source code), requiring minimal interaction and displaying the default view of the codebase visualiser. The second task introduced a section of the significantly larger DVLA tachograph-driver-card repository, whilst prompting users to remove external packages from the visualisation. The following task then scaled up again to the entire driver card

workspace, at which point the user was advised to introduce the slicing feature to visualise the codebase at a lower package resolution.

As a follow on from this, it was decided that, during each task, the user would be prompted to identify key structural features of the codebase, i.e. packages with particularly strong dependencies on each other, packages with high overall popularity and so on. This followed as a result of the cognitive walkthrough, ensuring that users would understand key functions that the software would enable them to perform and confirm that they are given appropriate feedback from the system to inform their subsequent comments. All task run-throughs were designed to be recorded, either through video, audio, or hand-written notes, dependent on user agreement, to ensure clear documentation of user interaction. In addition to the live questions on structural features, users would be encouraged to talk through each task aloud, in order to retain more qualitative data. Subsequent to their completion of all four tasks, the users would complete two questionnaires on the system.

4.7.2.2 User Questionnaires

Two forms of questionnaire were selected to achieve a larger range of possible data. The standard SUS questionnaire, as shown in: Appendix C – System Usability Scale (SUS), would be used to provide a more generic usability grading, enabling more quantitative comparison with other systems. This would then be complemented by a questionnaire focusing on the features of this visualisation system specifically.

With respect to the system-specific survey, questions were essentially split into two main types. “How much” type questions, using Likert scales to allow users to quantify a particular feature more easily, tended towards assessing the potential of the system and its overall usefulness. “What” type questions, which were probing and qualitative in nature, would allow the user to provide suggestions for system improvement in a more free-form fashion, paving the way for further system features and requirements. It was also decided that users should enter their role and experience (both inside and outside of the DVLA) as part of the questionnaire, to indicate the range of users tested, as well as providing context for their ratings and the more information-rich responses they provided.

4.7.3 Findings

This subsection will provide an overview of the interview and questionnaire results for each participant, in order of date. This represents a summary of information contained in both the recording/handwritten notes taken by the researcher and the questionnaire form filled out by the user.

4.7.3.1 Participant 1

The first participant was a year in industry student with 3 months experience as a developer and at the DVLA. Their inclusion in the data sample can be viewed as particularly appropriate, as the early motivations for developing the system lay in the desire of a novice developer for a more intuitive way to get to grips with an unfamiliar codebase in a new place of work. As such, the data received via a participant from a comparable context has particular value as project feedback.

The participant was able to successfully identify elements such as the most and least connected nodes, as well as the strongest individual dependencies. In addition, they were able to understand the distinction between internal and external packages and different layers of packages and successfully navigate between them using the functionality provided by the system. One point mentioned during the walkthrough was that the participant’s initial assumption had been that an increase in the “depth parameter” would signify a decrease in package depths, i.e. with a lower number indicating visibility of lower levels. However, in the current system increasing the depth parameter increases the number

of layers of packages visible, with a smaller depth parameter corresponding to a shallower visualisation. This should be considered in the context of intuitive understanding, with respect to the interpretations of other participants

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are shown in Figure 44. As shown, the only non-maximal score was attributed to the format of the dependency graph, which can be addressed through the use of clear documentation, or an in-system guide or key.

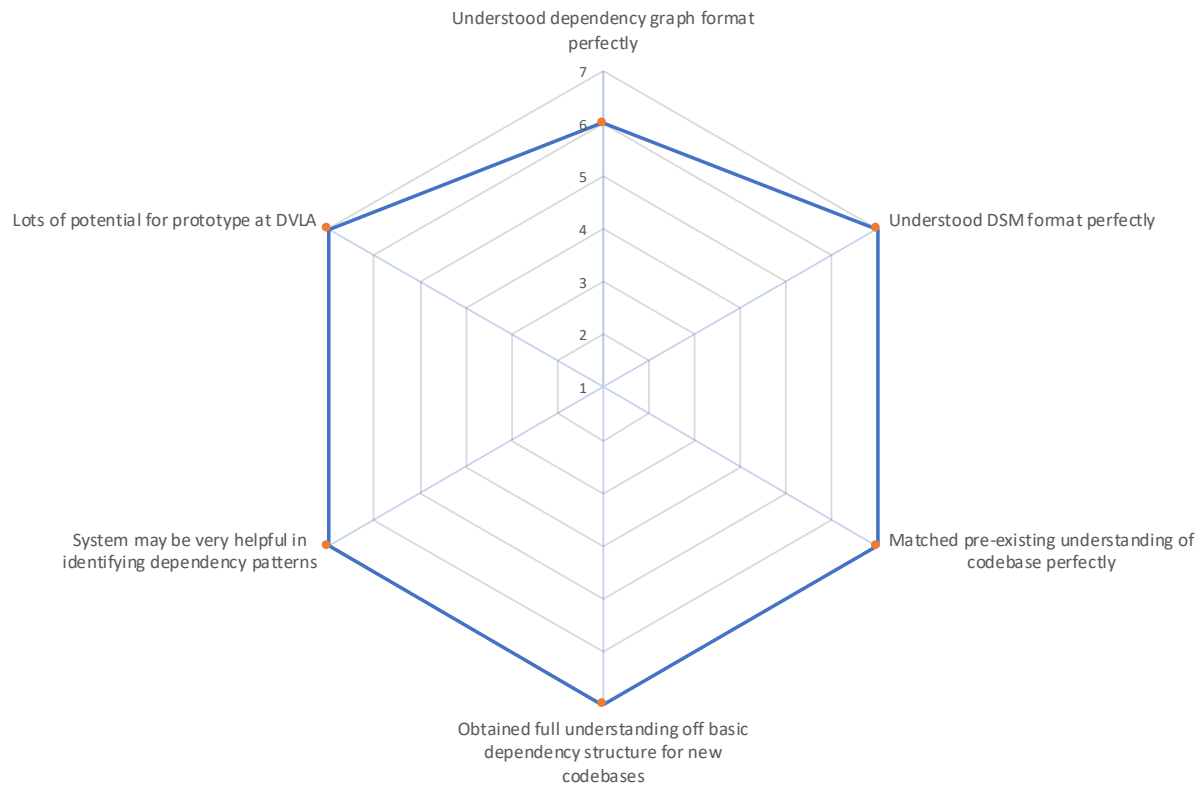


Figure 44: Graph illustrating quantitative data for system specific questionnaire responses of participant 1

The participant's completion of the SUS questionnaire produced a usability score of 92.5. This represents a grade A SUS score, with average systems receiving SUS scores of around 68 in general. Points of improvement from the SUS questions were: system ease of use, cumbersome and confidence whilst using the system. It should be noted that these issues should be at least partially addressed by continued use of the system, as opposed to this rapid introduction

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant submitted two main suggestions on the questionnaire. With regards to improving the understandability of the dependency graphs and DSM formats, the participant advised that the system should indicate that the depth parameter represents the maximum number of package levels shown in the view, as opposed to the numeric depth of an individual layer being viewed. In terms of future features, the participant suggested introducing the ability to select and highlight a particular dependency, in order to more easily distinguish it from other elements. This degree of interactivity also introduces the possibility of selecting an element in order to produce more detailed visualisations of its context. The participant felt that no improvement was necessary to increase similarity to familiar codebases or impart further understanding of new ones.

4.7.3.2 Participant 2

Participant 2 was a senior test engineer with more than 5 years' experience working as a tester at the DVLA. Although not possessing software development experience, they interact with various DVLA codebases on a regular basis during exploratory testing and are known to be extremely thorough and meticulous in their approach to software evaluation. It is hoped that their procedural rigour, combined with the fact of having no programming experience, will provide a bottom line of usability for a trained engineers' interaction with the system.

The participant was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. In addition, they were able to link the DSM to the graph in the correct manner and correctly interpret the significance of node popularity, strength of dependency and self-dependency (i.e. shown by a single edge which begins and ends on the same node). The participant struggled to interpret the meaning of disconnected packages during task 3 and also encountered some difficulty in producing sliced visualisations using the depth parameter via the command line interface during task 4.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 45. It should be noted that the participant felt unable to claim any pre-existing understanding of the DVLA codebases, and as such could not provide an indication of how closely they matched with the visualisation, this element is therefore excluded from the graph. Key issues appeared to be getting to grips with the new DSM and edge node graph format displayed by the system, whilst the user still felt that there was significant merit in the prototype itself.

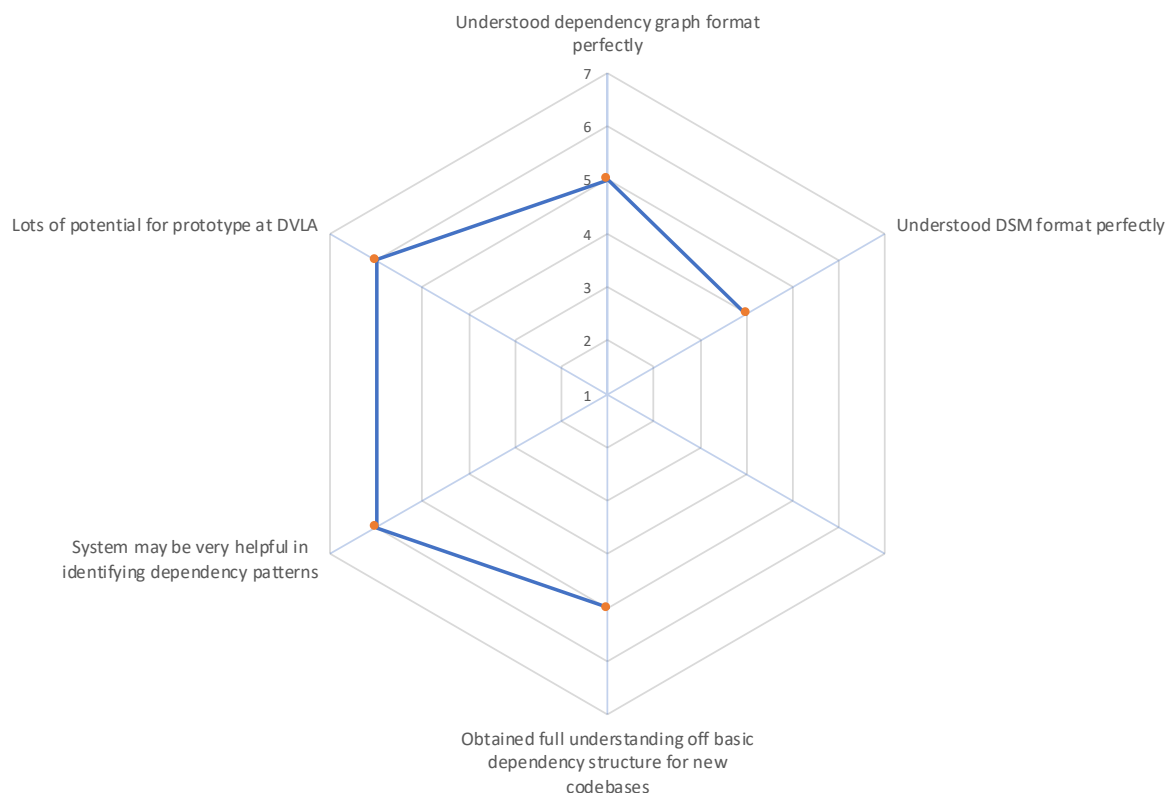


Figure 45: Graph illustrating quantitative data for system specific questionnaire responses of participant 2

The participant's completion of the SUS questionnaire produced a usability score of 55. This represents a failing grade SUS score. The explanation for this low score can to some extent be attributed to the

participant's lack of development experience, in addition to punctilious nature of their role, as tester they must attempt to break the system wherever possible. Key points of improvement from the SUS questions were: the participant felt they wouldn't use the system frequently and also felt they would require the support of a technical person in order to use the system. The former can be attributed to the participant not working directly with code and therefore not needing to be aware of the dependency structure of each codebase. The latter may bring to the appropriateness of a CLI when the user is not a developer, an alternative might be a GUI or similar, with more visual options for package layering and depth slicing.

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant submitted one key suggestion on the questionnaire. With regards to making the DSM more understandable, the participant suggested that the direction of the dependency should be made clearer in the tooltip when hovering over the DSM. They suggested producing a separate, linear list of all package-package dependencies for increased clarity. Besides this, the user could think of no further necessary features, and no further improvements to increase understanding of basic dependency structure of new codebases.

4.7.3.3 Participant 3

Participant 3 possessed approximately 1 year of experience working as a software developer at the DVLA. This individual is a participant on the same part-time MSc scheme as the researcher, with similar previous levels of programming experience and has worked on the same DVLA codebases. It is hoped that their perspective will therefore provide valuable confirmation or contradiction of the researcher's assumptions in the completion of this project.

The participant was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. They also attained a quick and intuitive grasp of the depth slicing parameter and its use, being able to identify key features of importance in the free-form fourth task. They misinterpreted the dependency direction, having assumed that the arrow should point from independent to dependent package. They also mentioned the issue of scaling the DSM, with large codebases leading to overly condensed column title fields. The participant strongly suggested the inclusion of a key to the system display, so as to clarify structure and format.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 46. As shown, the key area of improvement from their perspective was understanding the format of the dependency graph. This is one aspect which could be addressed through use of an informative key in tandem with the visualisation tool.

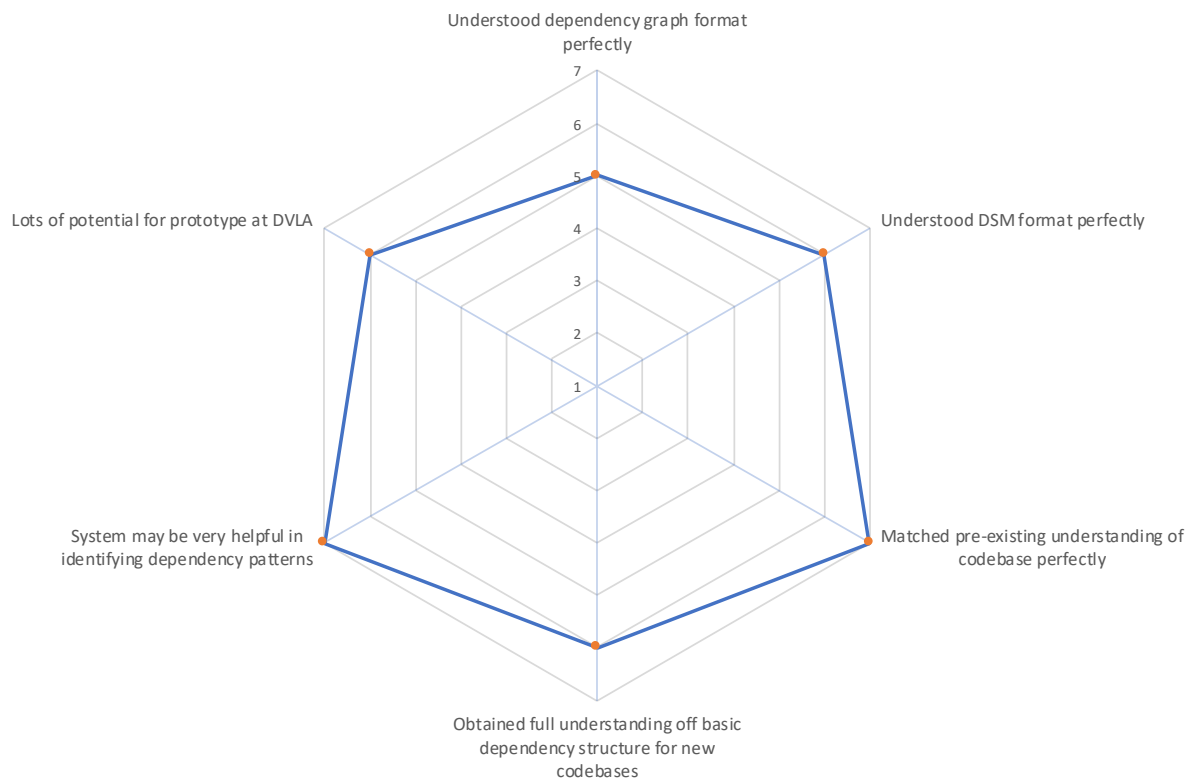


Figure 46: Graph illustrating quantitative data for system specific questionnaire responses of participant 3

The participant's completion of the SUS questionnaire produced a usability score of 70. This represents an above average or "passing" SUS score. Key points of improvement from the SUS questions were: need of support from a technical person to use the system, confidence in using the system and integration of the system's functions. The first two can be addressed by introducing the changes that the participant mentioned, in particular introducing a key to illustrate the graph format. System integration may be approached by unifying the manner in which the different display windows pop up, i.e. displaying DSM and graph in the same container as opposed to separately

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant submitted several suggestions on the questionnaire. In order to impart increased understanding of dependency structure, the participant suggested introducing colour coding different packages to give a better idea of system makeup and function. In terms of making both the DSM and dependency graph more understandable and as an additional feature for future releases, the participant strongly suggested the inclusion of a key/legend illustrating arrow direction meaning etc. In addition, they suggested addressing the scalability problems of the DSM, for example introducing text wrapping or scroll bars to avoid "squishing" of DSM fields, and also introducing a more drastic colour gradient to increase distinction between packages etc.

4.7.3.4 Participant 4

Participant 4 possessed approximately 3 year of experience working as a software developer, with one year at the DVLA. This individual is a participant on the same part-time MSc scheme as the researcher, and has worked on the same DVLA codebases, but had more pre-existing coding experience and was more specialised towards the User Interface side of development.

The participant was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. After correctly interpreting the meaning of the DSM, one comment made by the participant was that the tooltip produced for the DSM disappears too quickly. The participant mentioned that a web-based solution might be preferable here.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 47. As shown, the key area of improvement from their perspective was imparting understanding of basic dependency structures for new codebases. Their suggested methods are summarised along with the features they recommended adding.

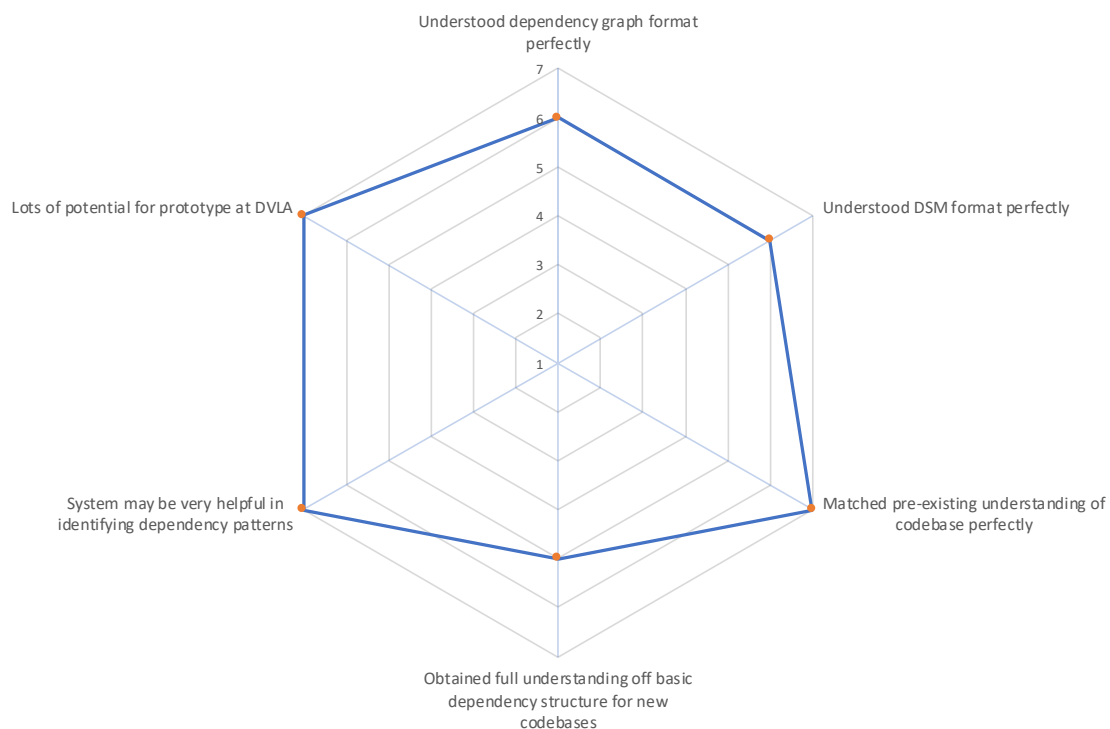


Figure 47: Graph illustrating quantitative data for system specific questionnaire responses of participant 4

The participant's completion of the SUS questionnaire produced a usability score of 82.5. This represents a grade "A" SUS score. The key area of improvement from the SUS questionnaire was that the participant felt they needed to learn a lot of things before they could get going with the system. This is something which can primarily be attributed to the length of time required to explain new concepts, such as the dependency graph or matrix. However, the introduction of a key/legend will mean that every detail of the system need not be known before beginning the visualisation process.

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant two main suggestions on the questionnaire. In order to impart increased

understanding of dependency structure as well as the format of the DSM and graph, the participant suggested making the zoom function more obvious and allowing the graph and the DSM to be displayed side by side. This may be viewed as linked to participant 3's responses on the integration of the system's features.

4.7.3.5 Participant 5

Participant 5 had 6 years' experience as a developer and had worked at the DVLA for 6 months as an SDET. It is hoped that the combination of their different role, SDET, with their early experiences at the DVLA will provide an insight which is valuable for this project.

The participant was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. The participant used the tooltip to identify package names in an unprompted and in a very natural fashion.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 48. The participant's completion of the SUS questionnaire produced a usability score of 80. This represents a high "B" grade SUS score. The key areas of improvement from the SUS questionnaire were ease of use of the system and likelihood of the participant to use the system frequently. This matches with the other quantitative data obtained from the participant, as shown.

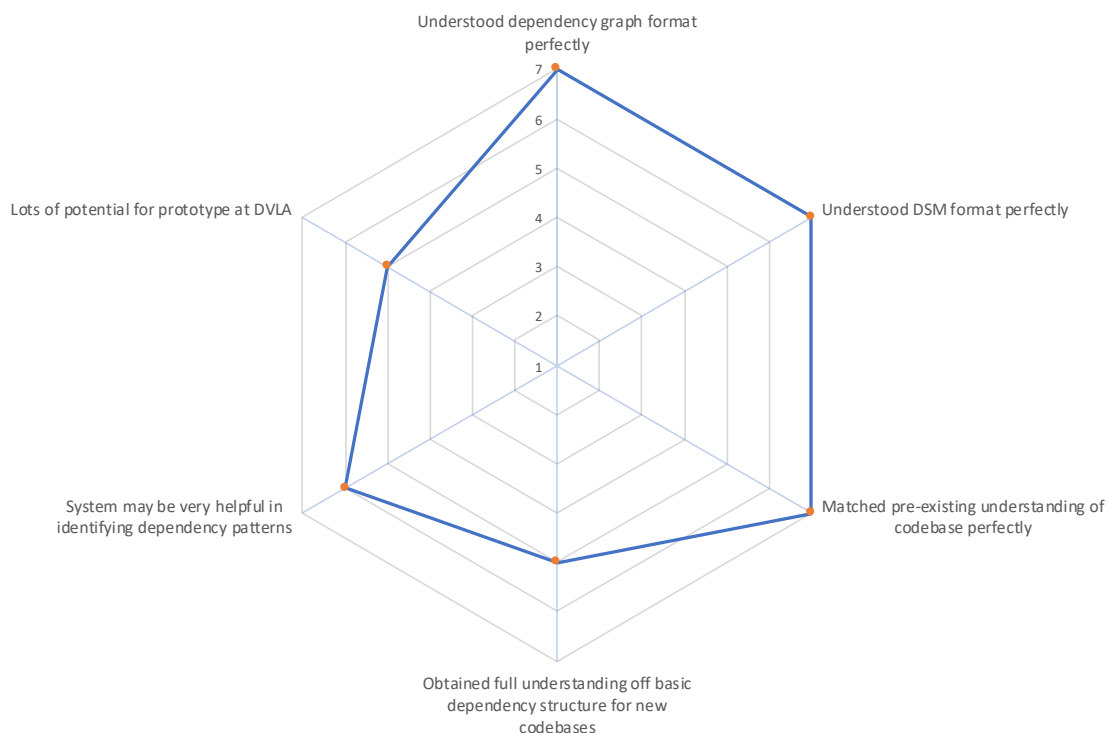


Figure 48: Graph illustrating quantitative data for system specific questionnaire responses of participant 5

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant had a number of points, which they addressed separately for the graph and the DSM. For both, however, it was suggested to give only the partial name of the package, as opposed to the full package string, i.e. trim the package names to remove shared starting strings. For the DSM it was suggested to bring up the name of a given element in a separate window when clicking on it. For the graph, it was suggested to add functionality to remove packages which don't possess dependencies. Also, bi-directional arrows were suggested to replace loops where two nodes each

depend on each other. It was also suggested that new colours should be introduced, such as red and blue, as opposed to just saturation, to help distinguish between packages.

4.7.3.6 Participant 6

Participant 6 has 1 years' experience working as a test analyst at the DVLA. Again, the inclusion of this participant offers the opportunity to get more range rather than purely a developer perspective. Additionally, his relatively shorter experience contrasts with the other non-developer included in this data sample.

Excluding the codebase for task 2, where they picked the second strongest dependency, the participant was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. They mentioned some possible confusion over the differing meanings behind the use of saturation in the graph and the use of saturation in the DSM. They also were confused by the external dependencies being coloured grey, which is similar to how low saturation, unconnected internal nodes would appear. They also suggested highlighting the rows of the DSM in a different colour to white, to prevent confusion, as the zeros disappear upon highlighting. Lastly, they wished to note that they felt they understood the system and felt that it was intuitive in nature.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 49. It should be noted that, although the users scoring might suggest some discrepancy between the code and the visualisation, they explained that their current understanding of the DVLA tachograph projects as a whole were limited, but that through using the tool they got a good idea off what the structure was exactly.

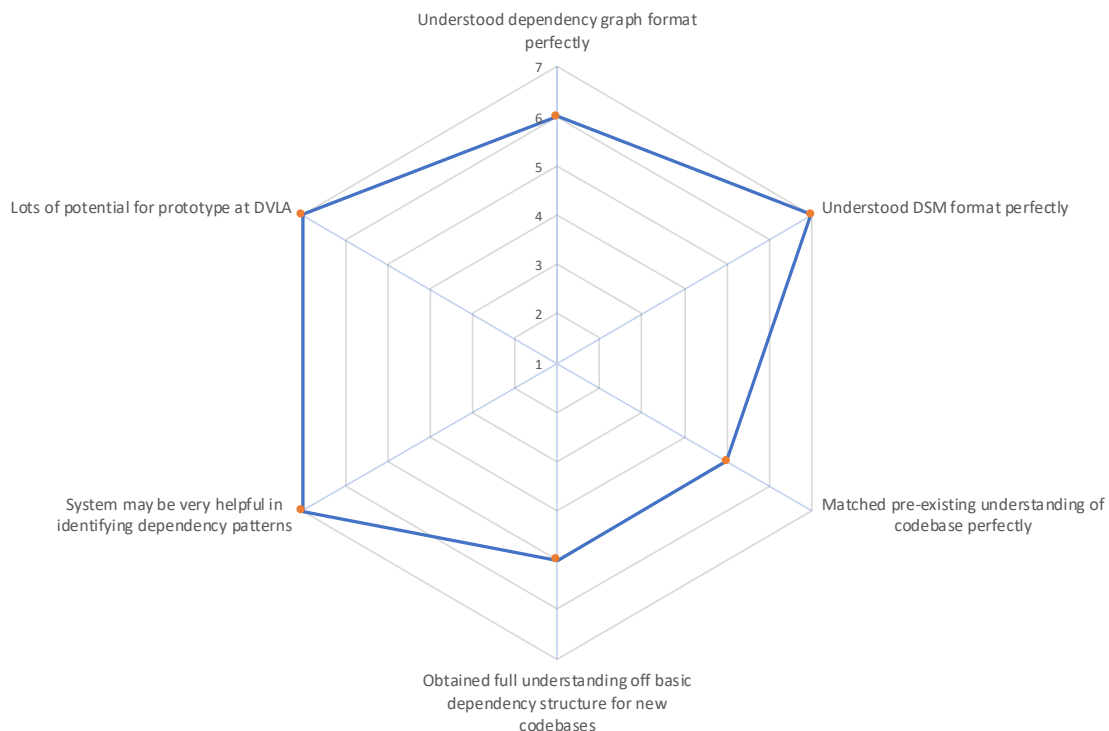


Figure 49: Graph illustrating quantitative data for system specific questionnaire responses of participant 6

The participant's completion of the SUS questionnaire produced a usability score of 95, representing a high "A" grade SUS score. In order to increase system ease of use and decrease learning overhead before using system, they suggested highlighting individual rows/columns in the DSM according to the package with the most dependencies. A further feature they suggested was the introduction of different line colours on the graph to show nodes with significance in the dependency structure, line saturation could also be varied, this would allow users to focus more easily on the key connections without getting lost in the network.

4.7.3.7 Participant 7

Participant 7, a software developer with 11 months experience at the DVLA was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies. As with the previous user, they mentioned the highlighting of empty rows in the JTable DSM, with the zeroes disappearing due to the white highlight colour. They also suggested that straight lines be used for the arrows as opposed to graphviz's default curving lines. They also suggested the use of different textures of line, such as hashed arrows, to separate different levels of dependency. Lastly, they addressed the possibility of increasing colour contrast, for example by transitioning from blue to turquoise to green, as opposed to only scaling via saturation.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 50. It should be noted that, when explaining any difference between the codebases they have worked on and the visualisation, the participant made clear that the tool had actually revealed new information about the source code to them. Any decrease in imparted understanding of dependency structure was attributed by the participant to the limitations of the graphing software (graphviz).

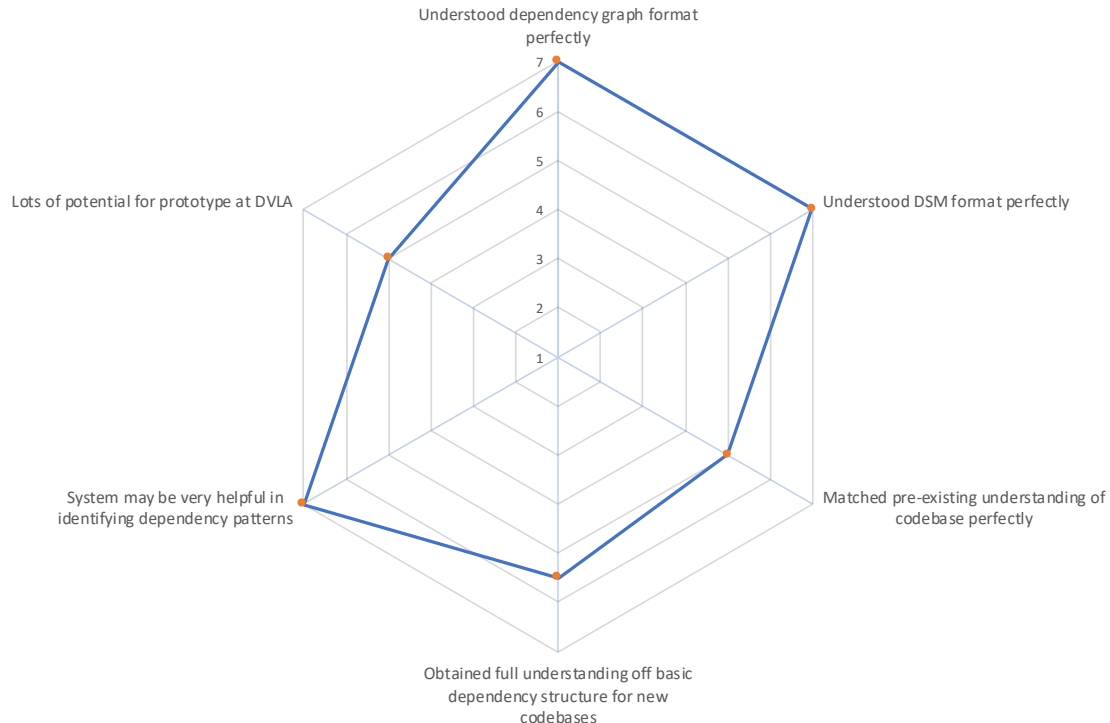


Figure 50: Graph illustrating quantitative data for system specific questionnaire responses of participant 7

The participant's completion of the SUS questionnaire produced a usability score of 75, representing a low "B" grade SUS score, with the lowest relative score being attributed to the degree of integration of the system's functions. Suggestions for future features from the participant included a clearer graphing system, perhaps using straight lines, to fill in the diagonal on the DSM showing the line of self-dependence and the possibility of showing row labels on the DSM as well as column labels.

4.7.3.8 Participant 8

Participant 8, a software developer with 4 years' experience overall and 2 years with the DVLA, was able to successfully identify, for each repository, elements such as the most and least connected nodes, as well as the strongest individual dependencies.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 51. An immediate observation is that the potential for adoption of future expansions of the prototype by the DVLA has received the minimal score. The participant made it clear that he does not view this as a reflection on the software itself, but rather it reflects his view of the civil service, and the low likelihood of any specific piece of new technology being adopted in that environment. With regards to ways to impart fuller understanding of dependency structure, the participant had 2 suggestions: highlighting common components more prominently, e.g. perhaps with colour, and having set designs for certain patterns, such as MVC, Bridge, Visitor etc, which the system would then recognise.

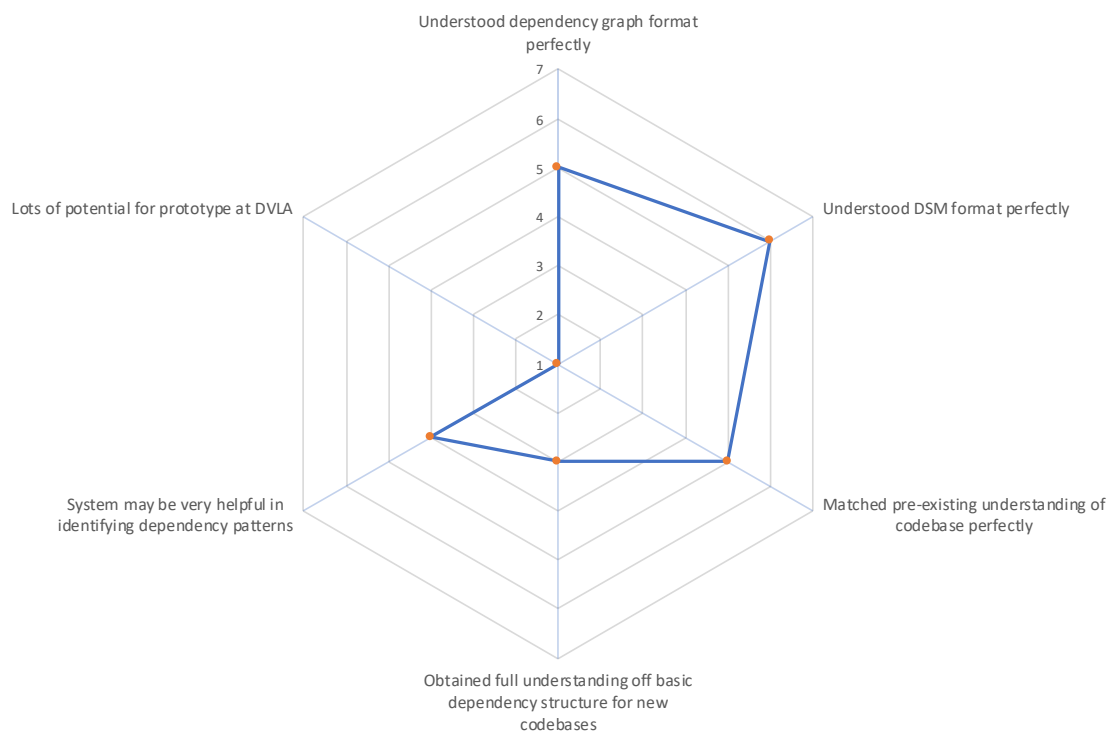


Figure 51: Graph illustrating quantitative data for system specific questionnaire responses of participant 8

The participant's completion of the SUS questionnaire produced a usability score of 67.5, representing an approximately average "passing" grade SUS score, with the lowest relative score being attributed to the number of things that needed to be learnt before using the system. Again, this is one of the aspects which it is hoped can be at least partially resolved through the introduction of a key/legend. The participant had 3 suggestions of features for future releases of the system. In order of priority,

design pattern recognition and classification, changing the colour of the nodes on the dependency graph to indicate different package types and lastly, for releases in the more distant future, interactivity, such as the ability to move nodes about on the graph, etc.

4.7.4 Summary

The average SUS score across all 8 participants was 77.2. This is a B Grade SUS score, comfortably above the average score of 68, in the top 30% of scores [96]. As such the system can be said to have good levels of usability in the general sense. The graphic representing the more system-specific quantitative results from the user questionnaire are shown as averages of the values supplied by each participant is shown in Figure 52.

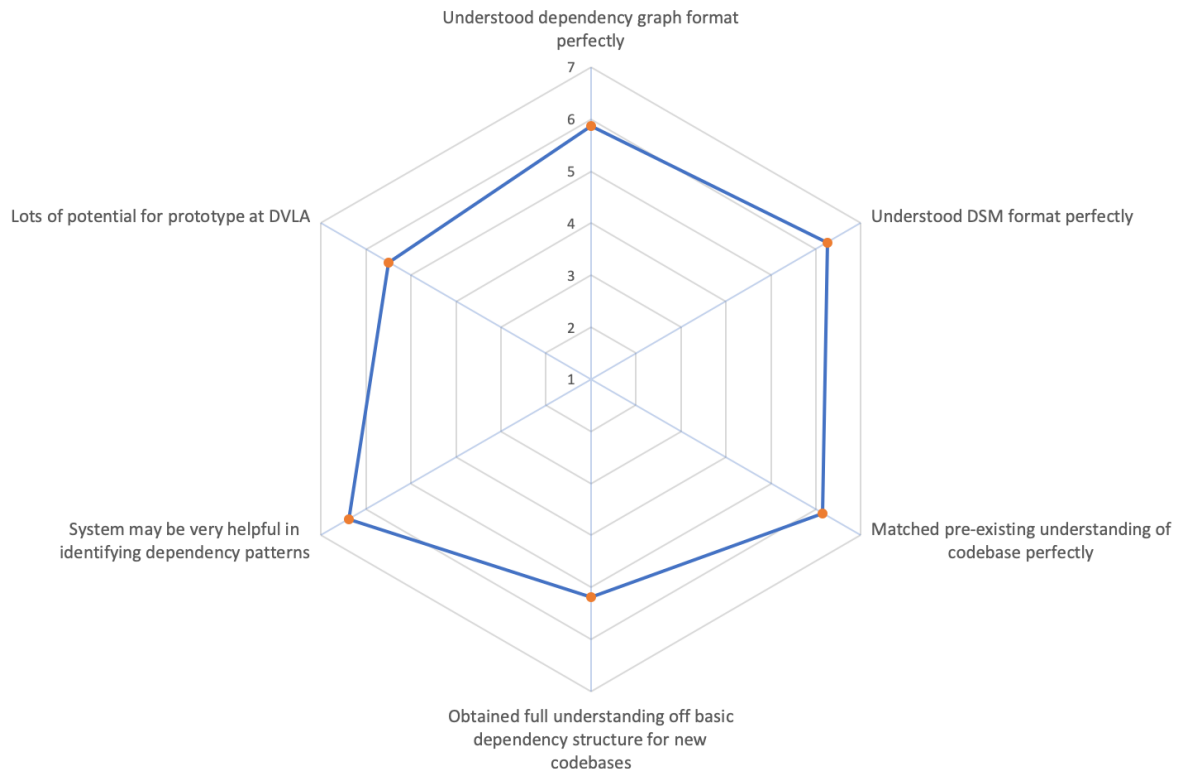


Figure 52: Graph illustrating average values of quantitative data for system specific questionnaire

Ultimately, it should be noted that all participants except one indicated significant levels of potential for such a system at the DVLA, with a Likert scale score of at least 5/7. Furthermore, all except one of the participants were able to identify key features, such as strongest package-package dependency, and those packages connected to the highest and lowest number of other classes, for every single codebase they were shown, using this tool. In addition to the SUS survey data and the more qualitative positive comments supplied by participants during the interview process, there is sufficient evidence to indicate significant potential for this static analysis tool.

In addition, a range of suggestions for possible additional features and solutions to potential problems/inefficiencies were provided as a result of the documented interviews. Predominant amongst these are the need for a key to be displayed as part of the visualisation process to increase comprehension, also the use of increased colour verification to enhance contrast between packages of different types and some method of shortening package names to prevent distortion due to long strings where avoidable. A number of other key suggestions, such as the need to introduce interactivity to enhance viewing, integrate all of the view elements off the system into a web front

end to increase integration and increase separation of packages into multiple different types using intelligent recognition, go beyond the scope of the remaining timebox. However, they are extremely appropriate suggestions in terms of future work.

4.7.5 New requirements to be addressed in the following timebox

- Integrate key/legend with visualisation system
- Colour coding of internal and external packages to distinguish them more easily
- Trimming of shared strings from package names where possible

4.8 Timebox 7

Date – 6th January 2019

Duration – 7 Days

Completion Date – 13th January 2019

4.8.1 Features

1. A key or legend should be designed which will be displayed in tandem with the dependency graph
2. External dependencies changed in colour to better contrast with normal dependencies
3. When displaying package identification strings in DSM or graphview, any pre-fix string shared by all packages (such as a list of enclosing packages) will be removed
4. Minor code refactoring and change tooltip arrow length

4.8.2 Design and Implementation

4.8.2.1 Use Case Description

Use Case Name: Display legend with graph of trimmed package strings and colour contrasted external packages
Stakeholders & Interests: End User
Brief description: End user wishes to understand the format of the displayed graph with shorter package names and more clearly differentiated package nodes
Trigger: A dependency graph needs to be easily understood
Normal flow of events: 1. The user enters a directory to analyse 2. Dependencies are extracted according to package structure, then any shared prefix between package names is trimmed 3. Legend detailing format of dependency graph is displayed 4. Dependency graph is displayed with shortened package strings and blue coloured external packages
Alternative / Exceptional flow of events: S-1a: 2. There is no shared package string to be removed across all packages, so full package names are shown

Table 20: Use case description for key, package name trimming and colour change

4.8.2.2 Use Case Diagram

The graphic shown in Figure 53 displays the user's interaction with the updated system. As shown, there are two key pieces of added functionality: the additional display of a legend before the dependency graph is shown and some changes to the format of the subsequently displayed dependency graph

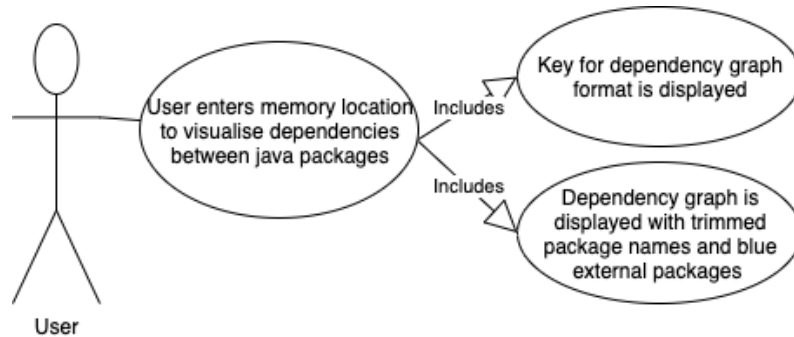


Figure 53: Use case diagram for displaying key, trimmed package names and colour contrasted external packages

4.8.2.3 Class Diagram

Figure 54 shows a UML class diagram of the updated system. As displayed, the structural changes from a UML perspective are restricted to the addition of two functions to the java package engine. As usual, modifications to existing functions and/or additions to the entity package are not displayed and changes to the class diagram are highlighted in green.

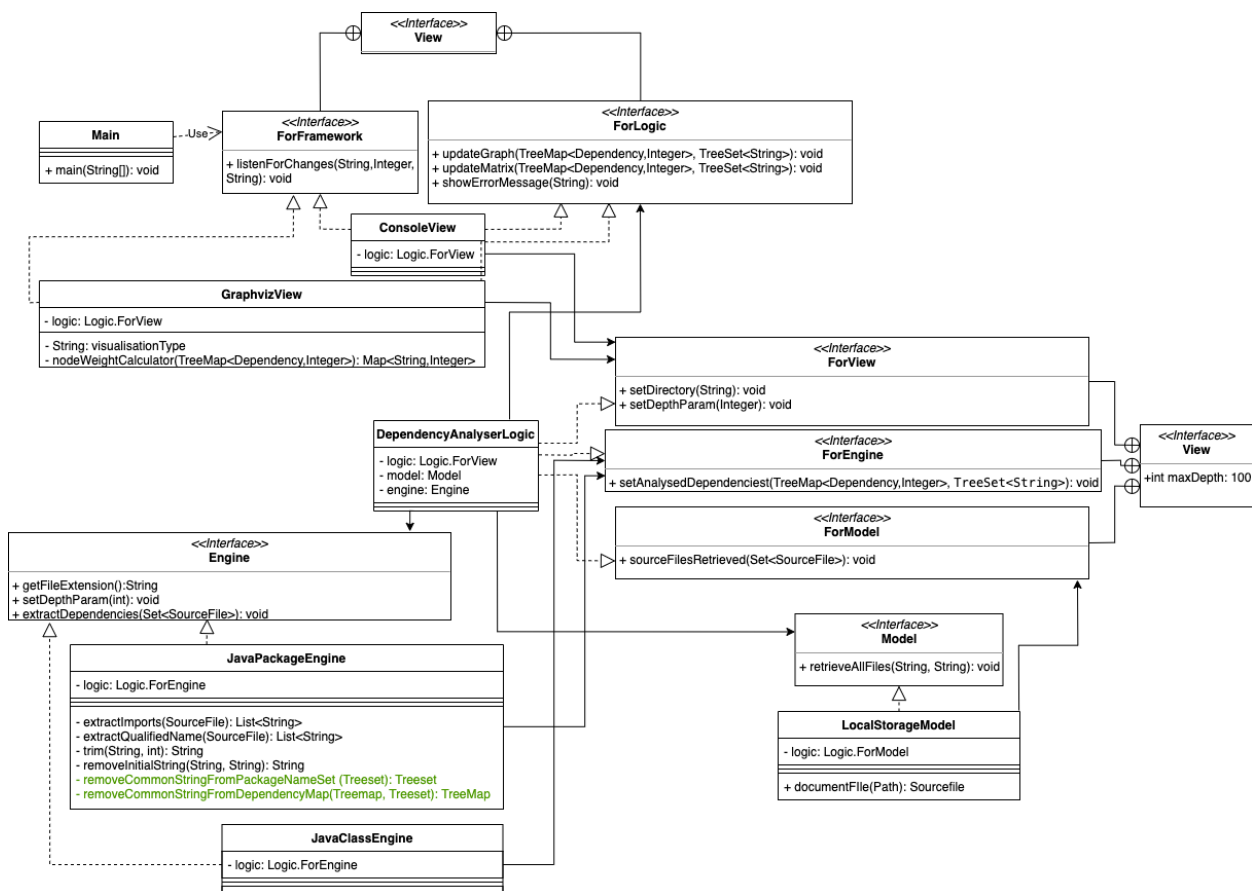


Figure 54: UML class diagram for displaying key, trimmed package names and colour contrasted external packages

4.8.2.4 Timebox Overview

The changes introduced during this timebox were largely cosmetic in nature, as they originated in user feedback from the first testing phase. It is viewed as a good indicator of code robustness that view changes are made possible through minimal refactoring. It indicates that separation of concerns has been effective in preventing linkage between, for example, the view and engine elements.

The addition of the key can be covered quite simply. The key, shown in Figure 55 illustrates the four key areas of the graph: edge direction, edge weight, node colour type and node saturation, meaning, respectively: direction of dependence, number of instances of a particular package to package dependency (i.e. dependency strength), internal or external nature of package and popularity of package. Three of the participants in the first testing phase were consulted informally during the construction of this key to ensure that it was clear in conveying element meaning.

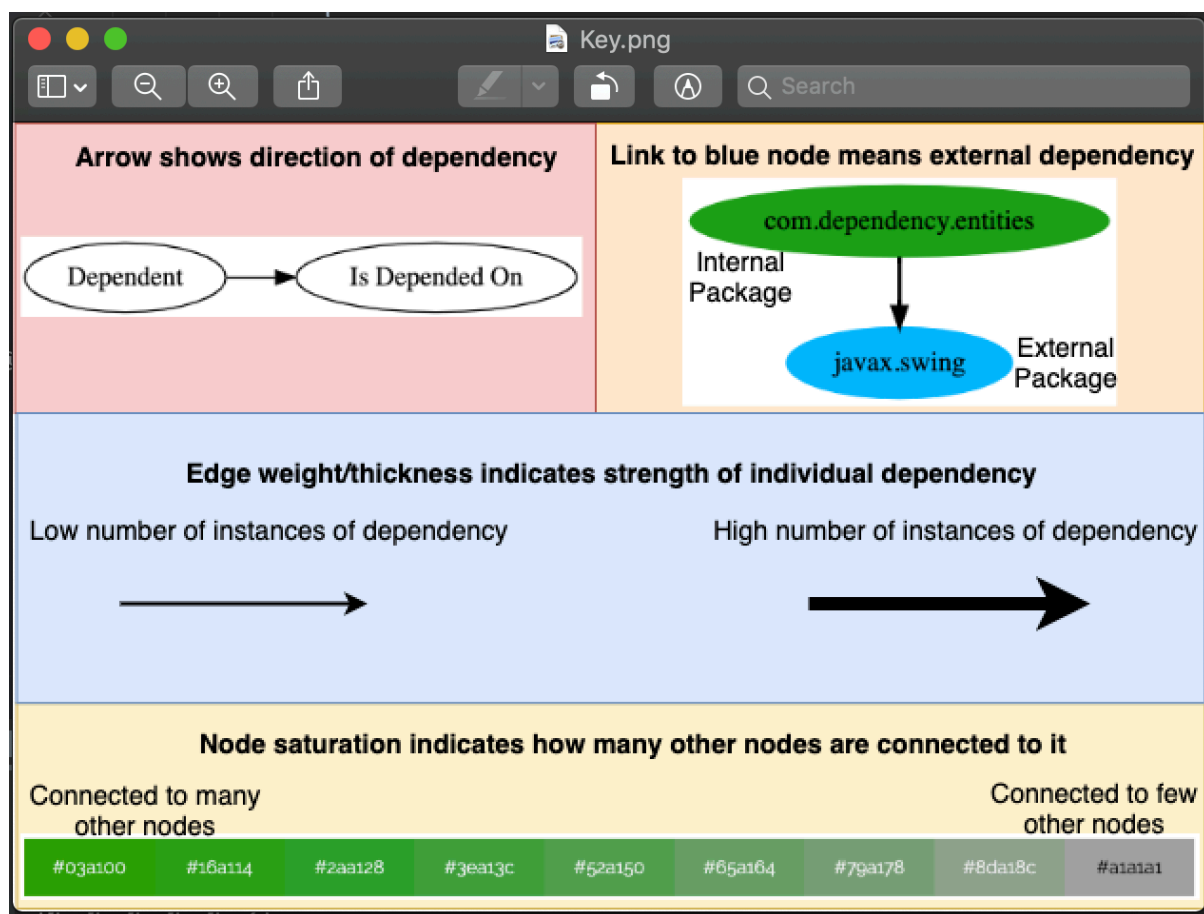


Figure 55: Key/Legend displayed to illustrate key aspects of the dependency graph format

The graph of the visualisation system with trimmed internal packages (the “com.dependency” package substring has been removed) and blue external packages can be seen in Figure 56. The trimming of package names feature was addressed through back-end changes. Here two functions were added to ensure that any shared string would be removed, whilst also ensuring that packages would not be shortened beyond a shared prefix, and also that shared strings in the middle or end of the package name would be ignored, to guarantee that only superfluous detail would be removed. The function responsible for trimming the set of package names, prior to trimming from the dependency map is shown in Figure 57. In addition to these features there was a minor refactor to ensure robustness.

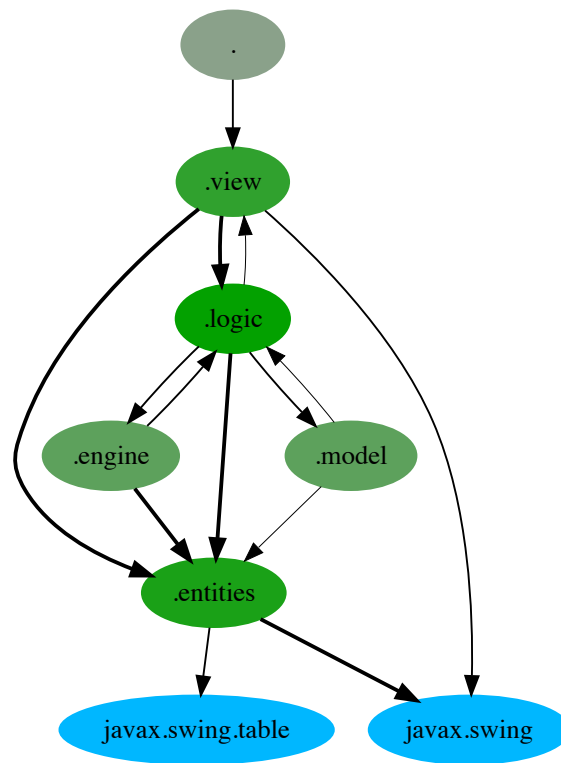


Figure 56: Dependency graph for visualisation system, showing trimmed package names and blue external package nodes

```

private TreeSet<String> removeCommonStringFromPackageNameSet(TreeSet<String> packageQualifiedNameSet){
    TreeSet<String> shortenedPackageQualifiedSetName = new TreeSet<>();
    sharedPackageString = packageQualifiedSetName.first();
    for (String packageName: packageQualifiedSetName) {
        if (sharedPackageString.length()>0){
            int sharedCharNumber=0;
            for (int i = 0; i < sharedPackageString.length(); i++) {
                if (sharedPackageString.charAt(i)==packageName.charAt(i)) sharedCharNumber++;
                else break;
            }
            sharedPackageString = sharedPackageString.substring(0,sharedCharNumber);
        }
        else break;
    }

    for (String packageName:packageQualifiedSetName) {
        if (packageName.substring(sharedPackageString.length()).isEmpty()){
            shortenedPackageQualifiedSetName.add(".");
        }else
            shortenedPackageQualifiedSetName.add(packageName.substring(sharedPackageString.length()));
    }
    return shortenedPackageQualifiedSetName;
}

```

Figure 57: Code used to remove prefixes shared by all packages from package set

Testing

Task ID	Task Name	Planned Results	Actual Results	Problems	Comments
1	Display DSM in tandem with dependency graph	Understandable key should be displayed upon running the visualisation command	Understandable key was displayed upon running the visualisation command		Understandability of key was tested informally with prospective users
2	External packages should be more colour contrasted with internal packages	External packages will be displayed as blue nodes instead of grey/green	External packages were displayed as blue nodes instead of grey/green		
3	Package strings should be trimmed to remove superfluous detail from package names, reducing length	If and only if part of the package string is a shared prefix across all classes, it will be removed from the name of each package	The name of each package was trimmed if and only if there was a shared prefix across all packages		For large packages or workspaces this feature loses relevance as the probability of a shared prefix decreases significantly

Table 21: Timebox 7 test sheet

4.8.3 Evaluation

This timebox was set at a length of 7 days, with all features required within the goal of the cycle being delivered to an acceptable level. Changes were primarily cosmetic in nature, focussing on the view and display format related suggestions from the initial testing timebox. The introduction of the key should prove useful in combatting confusion on the dependency graph structure, whilst also allowing users to begin visualising with the software without having to completely apprehend the nature of the display first. Informal feedback on the key is already positive and indicates success in this area, suggesting that perhaps a key for the DSM would also be beneficial.

In terms of graph display, package name shortening works well for smaller packages, or sections of packages, but tends to lose its effect over large workspaces. However, this is a much preferable scenario to the possibility of accidentally removing necessary package string data in the interests of shortening the name. This cautious approach to trimming may be easily supplemented in the future by a condition such as “if a string is shared by 80% of the packages it should be trimmed” instead of the current 100%. Introduction of blue colouring for external package nodes visibly increases contrast and can thus be viewed as a success.

Ultimately, there are many more features and potential requirements that could still be addressed in the code. For example, introduction of interactivity in the graph and/or the DSM. However, following the changes made in this final timebox, it can be said with some confidence that we have secured a functional prototype of the dependency visualisation system.

4.8.4 Requirements Addressed

- Integrated key/legend with visualisation system
- Colour coded external package nodes to distinguish them more easily from internal packages
- Shared prefixes are trimmed from package names where possible
- Slight refactor of view section to ensure robustness and lengthening dependency arrow displayed in tooltip of DSM

4.9 Timebox 8

Date – 13th January 2020

Duration – 7 Days

Completion Date – 20th January 2020

4.9.1 Features

- Select two participants from the first testing timebox
- Perform run-throughs of the same 4 exercises found in Appendix A – User Task List with 2 participants from the first testing timebox, recorded as video, audio or in handwritten format
- Obtain responses to the questionnaires: Appendix B – User Questionnaire and Appendix C – System Usability Scale (SUS) as before

4.9.2 Findings

The participants in this second testing timebox correspond to participants 1 and 3 in the first survey. The reasons for this are threefold. Firstly, they both made suggestions which had relevance to the modifications made to the software system. Furthermore, they are both relatively junior developers, not too accustomed to the DVLA codebases, linking them to the essential target of this project. In addition, the issue of availability due to a strict civil service deadline meant that participants had to be selected with some care to avoid jeopardising their government work. Implementation of the tests was kept as identical as possible to the first run-through with the only difference being a final verbal question of whether they felt the system had been improved.

4.9.2.1 Repeat of participant 1

The participant's software engineering experience at the point of questioning was in the region of 4-5 months. All of this time had been spent at the DVLA as a year in industry student from Swansea University. They were able to identify key features of the dependency structure, such as very popular packages and unpopular packages, as well as particularly strong dependencies, for each of the four different codebases presented in the user tasks. They were also able to attribute the correct meaning and significance to the different features using the DSM and the dependency graph.

The participant quickly mentioned the addition of the key as having a positive impact on the system. They were able to successfully use the key during their tasks to attribute meaning for example linking saturation to connectivity etc. They were also quick to note the added benefit of changing the colour scheme of external packages to blue to increase contrast, mentioning that it allowed them to know what they wanted to focus on more quickly. Overall, the user successfully engaged with the system in a fluid and natural fashion.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 58. On the scale of no understanding to complete understanding of dependency graph format the participant's score has jumped moved from 6 to 7. This can be attributed to a combination of the addition of the key to increase understanding and perhaps some familiarity from their previous run-through of the system in the preceding month. On the scale of potential at the DVLA their score dropped from 7 to 6, this can perhaps be attributed to their increased experience at the DVLA, with the new knowledge that currently no such visualisation tools are used

there. The participant's completion of the SUS questionnaire produced a usability score of 95. This represents a high "A" grade SUS score, indicating high system usability.

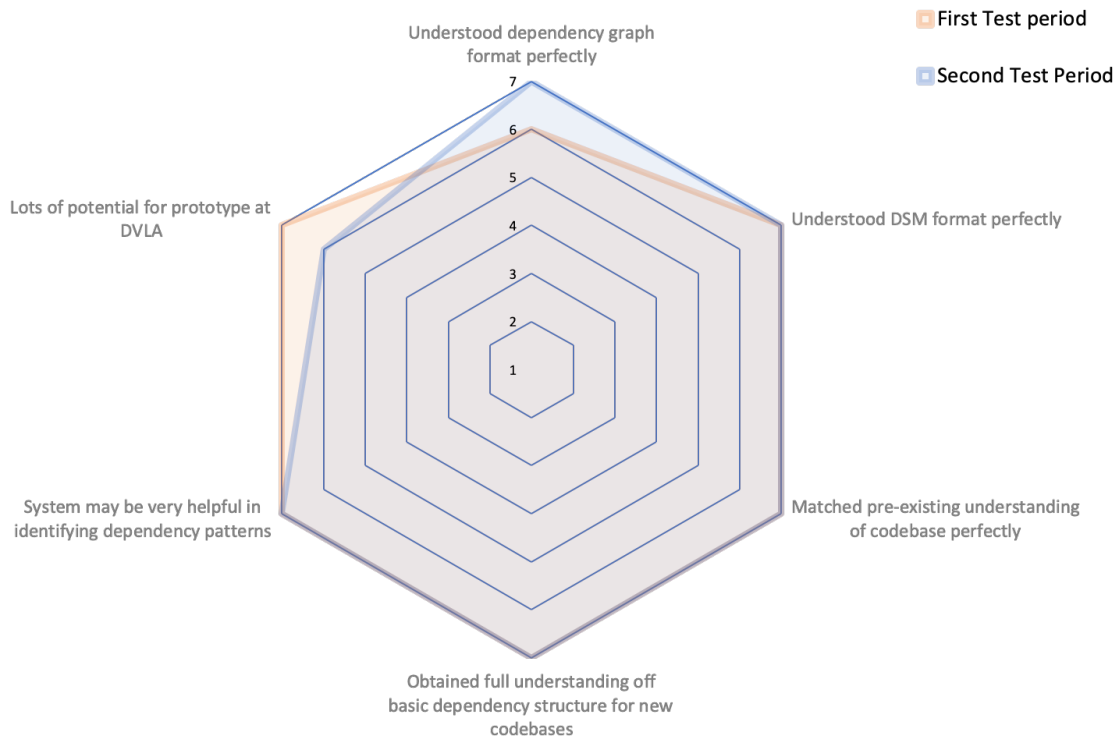


Figure 58: Graph illustrating quantitative data for repeat system specific questionnaire with participant 1

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant raised two points. Firstly, that linear scaling of edge weight on the graph and saturation on the DSM to represent dependency strength, begins to deteriorate for large workspaces. The reason for this is primarily that the number of instances of specific individual dependencies is simply too high, skewing the representation. The participant suggested the introduction of a logarithmic scaling method for this situation. Secondly, the participant again referred to interactivity, mentioning that the possibility of clicking on nodes to reach subsequent package levels would significantly enhance the visualisation system.

4.9.2.2 Repeat of participant 3

The was able to identify key features of the dependency structure, such as very popular packages and unpopular packages, as well as particularly strong dependencies, for each of the four different codebases presented in the user tasks. They were also able to attribute the correct meaning and significance to the different features using the DSM and the dependency graph and use functionality such as the zoom feature effectively.

The participant also used the key very effectively in each of the four tasks. They mentioned that the key made the different elements of the graph much "clearer", which stands to reason as they had suggested the inclusion of the key during their first run-through. During the fourth task they mentioned that the use of blue colouring made the separation of internal and external nodes much more obvious. The participant also noted the package name trimming function as being beneficial.

The participant's responses to the quantitative, Likert scaled elements of the system-specific questionnaire, are displayed in Figure 59. On the scale of no understanding to complete understanding of dependency graph format the participant's score has jumped moved from 5 to 7. This can be attributed largely to the addition of the key to increase understanding, which was a feature specifically requested by the participant in the previous run-through. Other contributing factors may include some familiarity from their previous run-through of the system in the preceding month, as well as the increased distinction through colour change of external packages. On the scale of how closely the tachograph visualisations matched their understanding of the codebase the DVLA their score dropped from 7 to 6. The participant mentioned that they were simply slightly surprised at the sheer number of dependencies present. The participant's completion of the SUS questionnaire produced a usability score of 95. This represents a high "A" grade SUS score, representing a significant jump from their previous score of 70, indicating a significant perceived increase in system usability.

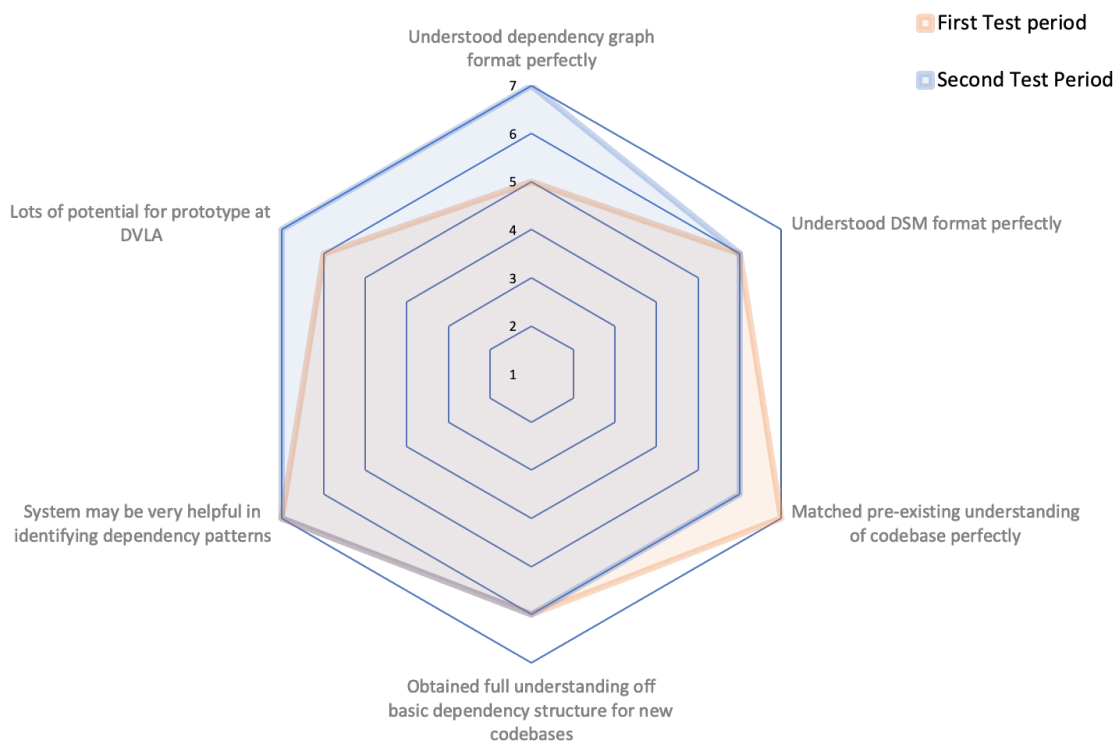


Figure 59: Graph illustrating quantitative data for repeat system specific questionnaire of participant 3

In terms of qualitative suggestions and potential guidance on features to include in subsequent releases, the participant raised one point. As with the other repeat participant, the key suggestion they made was to make the graph interactive, so that clicking on a node would enable you to interactively "zoom in on its dependencies". In the other probing improvement questions the user indicated no possible improvements, suggesting satisfaction with the system.

4.9.3 Summary

The average SUS score of the two repeat participants was 95. This represents a significant improvement from the first test timebox average of 77.2, moving from a "B" grade to a high "A" grade on the scale. The indication is that the usability of the system has considerably increased, however it should be noted that this evidence alone is not sufficient to make a statistical claim. The evidence for system improvement is also backed up by the combined radar diagram of the quantitative Likert scale

responses of the two participants shown in Figure 60, which can be compared to Figure 52 in section 4.7.4, a combined overlay graphic displaying the first and second test period is displayed in Figure 61 in section 5: Conclusions and Recommendations.

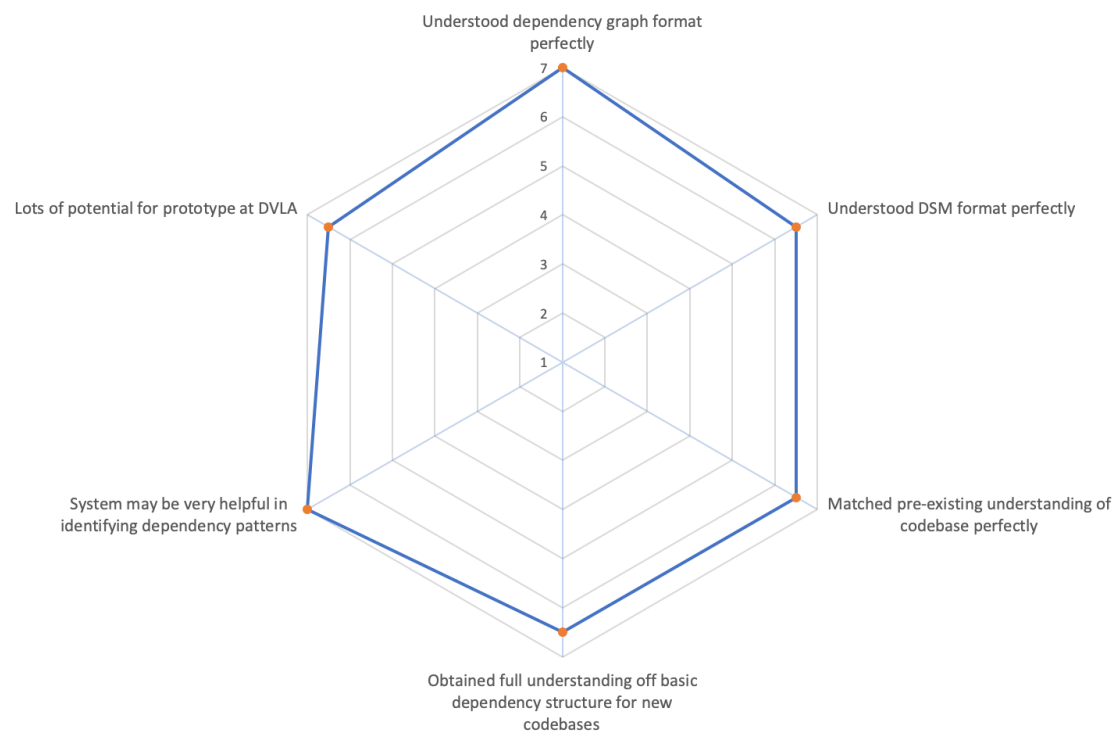


Figure 60: Graph illustrating average quantitative data for the second testing timebox

Both repeat participants confirmed that they had observed significant improvement in the visualisation system. They interacted with the added features in the intended fashion and gave positive confirmation of their beneficial impact on the project. The combined fact that this qualitative evidence was reinforced by their quantitative scorings on the questionnaire provides significant evidence that the system has been meaningfully enhanced by the addition of the features from the previous timebox. Lastly, in this final timebox, it should be noted that the key suggestions of the users centred heavily around interactivity. This suggests that, although there are numerous further enhancements that can be made to the current iteration of the system and always will be, the system is approaching the point where a more significant change is required. In order to increase to another level of usability, the system must undergo a more radical change of the kind that would take it beyond the scope of the current prototype project. A summary of this project, along with a discussion of possible expansions can be observed in the following section of the report.

5 Conclusions and Recommendations

This section of the report presents a summary of the project progress and achievements so far, as well as looking ahead to further work, i.e. expansions on the project which will be instrumental for future releases.

5.1 Conclusions

Ultimately, the aims of the project were to develop a prototype codebase visualisation system and determine its potential for implementation within the Driver and Vehicle Licensing Agency. The intention was that this prototype would act as a static analysis tool that would extract and display the basic dependency structure of a java codebase automatically. These aims have been achieved. The submitted software system is capable of taking in a local memory location, parsing through the contained java files and automatically generating an edge node graph visualisation and a DSM of any package dependency structure present within these files. Furthermore, this prototype has been tested, tweaked and then retested with real employees at the DVLA, via a series of interviews and questionnaires, to produce both qualitative and quantitative evidence of its effectiveness and ultimately its potential as a means of improving code quality at that agency. All of this is in line with the stated objectives of the project.

Over the course of this project, it quickly became clear that the developed system could yield some benefit if integrated into the DVLA's code review process as a static analysis tool, as discussed during the literature review [3][6]. Indeed, the inception of this project lay in the question of whether the understanding of code quality indicators such as cohesion and coupling could be enhanced via the use of such a system [13]. There is direct user feedback to support this assertion, with users indicating that they had gained new knowledge of their codebases via the system and indicating surprise at the degree of coupling revealed. This has also gone some way towards proving the effectiveness of including a DSM view in the display, confirming the postulation from section 2.3.1 in the literature review, that it would facilitate easier analysis. The fact that no user was able to identify any discrepancies between the dependency visualisation and the reality of the source code, also acts as good evidence of the effectiveness of the integrity of what was essentially a parser for the java language. This was created in accordance as the textual analyser which discussed in section 2.2.2.1, allowing lexical and syntactic analysis [47]. The success of this engine has highlighted the effectiveness the strategy of focussing on package dependencies first and foremost. It also illustrates the potential of the project to possibly begin to bridge the gap between software engineers and architects, by providing technical accuracy from its reliable extraction of the source classes, whilst producing visualisations at the less granular package level favoured by architects.

The software system utilises a graph format to visualise the dependency structure in a more intuitive fashion, drawing from the concept of directed cyclic and acyclic graphs from mathematics, interfacing the extracted dependencies with the displayed graph through the use of the DOT graph markup language [69][71]. In order to generate the edge-node graphs, the software system integrates with the graphviz tool package as discussed in section 2.4.2, as this was found to be more efficient than utilising java's own graphing libraries [75]. The ability of the user to select visualisation depths, essentially slicing off layers of packages depending on which resolution they wish to view the structure at, is at its heart a matter of abstraction, the removal of extraneous information, which is a fundamental aspect of information visualisation [30].

The software system bears some inspiration from existing manual systems of graphing dependencies, such as the UML and C4 diagrams which are the predominant models used by architects within the

DVLA, when visualisations are deemed necessary [84][86]. A number of features are also shared to some degree by existing technology, for example, the system uses colour saturation to indicate dependency strength, whilst in the NDepend package, colour is used in the DSM to identify cyclic dependencies [89]. The way this prototype separates internal and external package nodes using different colours, and illustrates this via a key is comparable to the manner in which CodeMR program represents module complexity using colour [93]. Lastly, the use of the edge-node layout has been proved successful through its utilisation by the sonargraph codebase visualisation tool [92].

Ultimately, all of this research and each of the requirements addressed as a result of it have contributed to a complete and functional codebase visualisation system, which goes beyond satisfying the MVP requirements laid out in section 4.1. The software system underwent an initial test with 8 software-facing DVLA employees, with roles including software engineer, tester and SDET. The results of this test period were positive, with an average SUS score of 77.2, significantly above the required average of 68, indicating high system usability. In addition, the system-specific questionnaires yielded an encouraging outcome, supporting the claim that the system is an effective tool, with the potential to increase code quality as shown in Figure 61. Furthermore, the qualitative data gathered was overwhelmingly positive in nature, with a number of participants indicating that they liked the system, that it brought new aspects of the codebase to light and that there was high potential for the prototype. After adding the final features based on survey feedback and retesting with a sample of the original participants, the SUS score increased to 95, and the system-specific data increased as shown in Figure 61, indicating a successful system update. The remaining suggestions from user feedback primarily fall into the range of further work and future expansions of the prototype.

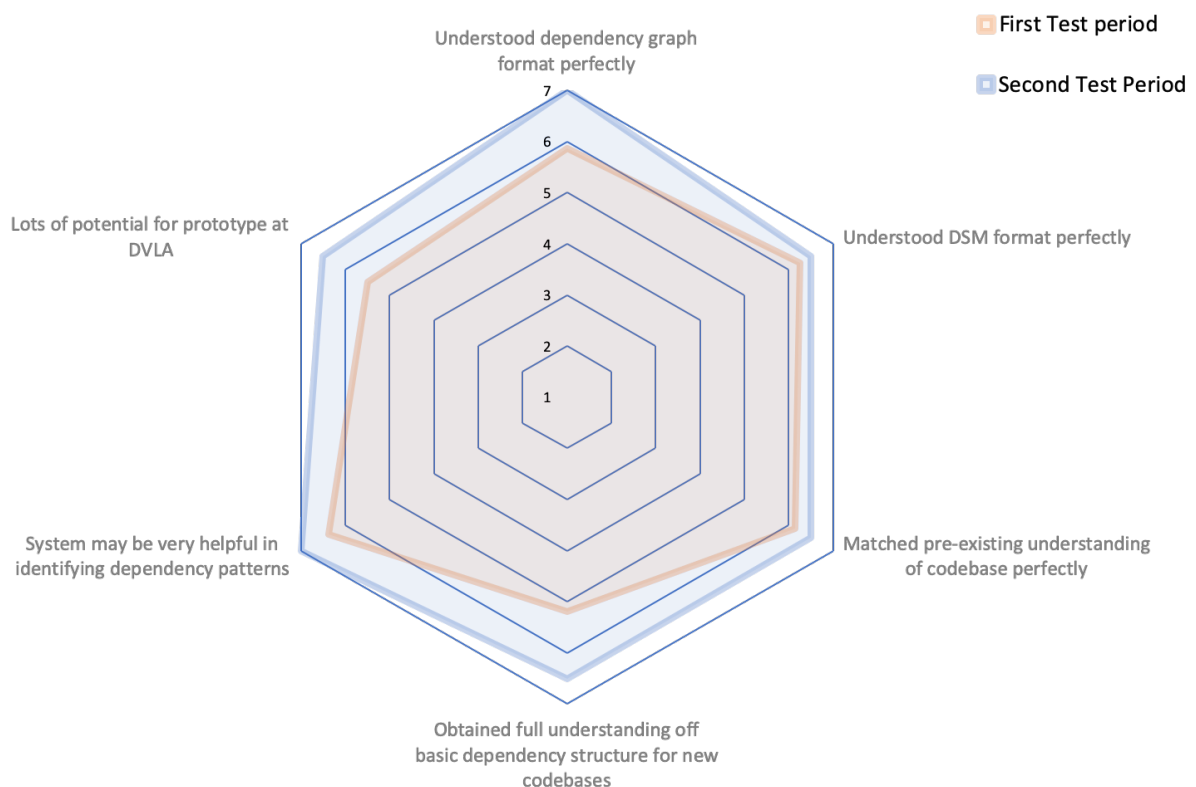


Figure 61: Graph illustrating average quantitative data for the second testing timebox

5.2 Further Work

There are a number of possible directions of future system expansions which the user feedback suggests are of particular merit and significance. Foremost amongst these must be the inclusion of interactivity into the visualisation display. This could allow, for example, the ability to select package nodes on the dependency graph in order to zoom in on their components. Selection of a given dependency could also indicate its numeric strength and/or reveal its constituent class-to-class dependencies. Clicking on a package could also highlight the other packages to which it is connected. In particularly advanced iterations, nodes and edges could even be set to light up in the order that the relevant modules and dependencies would be invoked during runtime. The depth slicing mechanism would also benefit from this upgrade, as one could simply use a toggled slider to increase or decrease granularity of the package structure, instead of having to enter a separate parameter to generate static images.

A suggestion from the testing timeboxes which may be particularly compatible with the possibility of interactivity was the concept of displaying the visualisation via a web front end. This would provide an effective way of consolidating the different components of the system together, improving on the scores received for system integration in the SUS questionnaires. Furthermore, this naturally leads to the idea of making each element in the visualisation a clickable hyperlink. New web pages would then be loaded to present each different view. Of course, all operation would be happening at a localhost level, allaying any security concerns that would come with the idea of loading, for example, DVLA code onto an external website.

Related to this feature is the concept of making the visualisation tool cross-system functional. The use of docker containerisation would be particularly effective in making the software platform independent. This would enable an image of the software to run up on essentially any OS, with the necessary tools such as, for example, graphviz, being bundled with it to prevent users from needing to download anything beside the system itself. As the system in its current prototype form, is only designed to work on one specific laptop, this form of update would be a key step towards a publicly releasable product.

One aspirational goal which was mentioned in the first user survey period was the possibility of making the system more intelligent through the introduction of pattern recognition. For simpler patterns this could be a standard set of rules, such as the formula for identifying cyclic paths within a network. This would enable the visualiser to highlight potentially negative dependency sub-structures within the codebase. However, in the more distant future, machine learning could be introduced, with a neural net capable of creating its own definition of bad code smells or formulating its own methods for recognising them.

Lastly, although this particular expansion was not raised as part of the test feedback and although there is often little need to go beyond the package level of visualisation, at least from an architectural perspective, it is the hope of the researcher that future iterations will be capable of extracting class dependencies. Currently, the system remains focused on package level dependency visualisation, with class dependencies marking a significant increase in the complexity of the extraction process as discussed in section 2.2.2.1. It is the view of the researcher that the JavaParser library will prove particularly valuable as means of simplifying the class dependency extraction process [52].

There are numerous other possible expansions of the system, such as introducing compatibility with other languages, interfacing with git, etc. However, the above, combined with the current prototype system, are deemed to have the greatest chance of increasing the potential of the product.

6 Reflections

I feel that a number of significant personal achievements have been made over the course of this research project. First and foremost, going from zero experience of java and practically zero experience of object-oriented programming (OOP), to producing not just a working object-oriented java system, but one which aims to simplify java OOP for others, just over a year later, is not a small undertaking. In addition, the successful collection of user data, despite all previous experiments being purely mechanical in nature, without room for ambiguity, is also something I view as a personal accomplishment.

Another key area which should be considered is the incorporation into the project of new knowledge from the MSc course which accompanied it. An example of this might include an understanding of coupling and cohesion, part of the key basis for this whole project, which originated during the Programming and Software Testing module in Autumn term 2018. An understanding of the UML classification of dependencies, and the importance of structural visualisation of code was also gleaned from the Agile Software Engineering Principles course in the same term. Both of these concepts have been fundamental in establishing the base understanding necessary to properly formulate this project. Even courses which may originally seem unrelated to this project, such as the Web Development course in Spring Term 2019, have had an impact. The focus on design patterns, in particular the MVC structure, provided the inspiration for the structure of the source code of the system. The idea of maintaining separation of concerns by ensuring that the model, view and engine packages only interacted through the interface of the logic module was only possible because of the new knowledge absorbed from the module mentioned.

However, as well as achievements, a number of challenges arose over the course of the project. Surprisingly, many of these were not related to programming. For example, at one point whilst recording an interview with a test participant, it was suddenly brought to my attention that certain work-colleagues had quite understandably been made uncomfortable by the idea of being unintentionally recorded during work hours. In order to rectify this, it was necessary to change location when conducting tests, and/or record via handwritten notes as opposed to video/audio. Another key challenge was effectively balancing the requirements of DVLA work with project work. Engaging in user testing whilst at work has the potential to raise questions amongst the development squad or worse, negatively impact achievement of the sprint goal. However, operating outside of work hours significantly increases the difficulty of securing enough participants for the survey. These sorts of dilemmas require middle of the road solutions which were not immediately intuitive to me and thus posed a challenge. One possible solution to these issues would be to work more closely with the relevant agile delivery managers and student coordinators to help avoid misunderstandings.

Possible changes I might make to my approach would include more user testing during the main process of development. Small groups could be informally questioned before or after key timeboxes to modulate project direction. Perhaps a focus group which ran prior to commencing the design process would have been particularly beneficial. I also would have brought in experimentation with tools such as JavaParser earlier on in the process, perhaps outside of a formal timebox. The reason for not doing this is that so many personally new areas were already being introduced during this project, as something of a personal challenge, that it seemed gratuitous to hurry to jump into yet another. However, again, doing so would have introduced new information which might have better informed project direction. That being said, I am not unhappy with the path that was taken and ultimately view the project as a success.

7 Works cited

- [1] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: a case study of the qt, VTK, and ITK projects," in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 2014, pp. 192–201.
- [2] I. García-Ferreira, C. Laorden, I. Santos, and P. Garcia Bringas, "Static analysis: a brief survey," *Log. J. IGPL*, vol. 24, no. 6, pp. 871–882, Dec. 2016.
- [3] T. Baum, O. Liskin, K. Niklas, and K. Schneider, "A Faceted Classification Scheme for Change-Based Industrial Code Review Processes," in *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2016, pp. 74–85.
- [4] A. Bacchelli and C. Bird, "Expectations, Outcomes, and Challenges of Modern Code Review," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 712–721.
- [5] H. Uwano, M. Nakamura, A. Monden, and K. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Proceedings of the 2006 symposium on Eye tracking research & applications - ETRA '06*, 2006, p. 133.
- [6] I. Dillig, T. Dillig, and A. Aiken, "Reasoning about the unknown in static analysis," *Commun. ACM*, vol. 53, no. 8, p. 115, Aug. 2010.
- [7] D. Distefano, M. Fähndrich, F. Logozzo, and P. W. O'Hearn, "Scaling static analyses at Facebook," *Commun. ACM*, vol. 62, no. 8, pp. 62–70, Jul. 2019.
- [8] M. Power, "Top Free Static Code Analysis Tools - Maxpower - Medium," *Medium*, 2018. [Online]. Available: <https://medium.com/@maxpowersii/top-free-static-code-analysis-tools-7d39abe36e5c>. [Accessed: 14-Aug-2019].
- [9] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why Don't Software Developers Use Static Analysis Tools to Find Bugs?," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 672–681.
- [10] R. C. Martin and M. C. Feathers, "Clean code : a handbook of agile software craftsmanship," Prentice Hall, 2009, pp. 32–48.
- [11] J. A. N. Lee, *International biographical dictionary of computer pioneers*. Fitzroy Dearborn, 1995.
- [12] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.
- [13] E. Yourdon and L. L. Constantine, "Structured design : fundamentals of a discipline of computer program and systems design," Prentice Hall, 1979, pp. 95–126.
- [14] D. L. Parnas, "Information distribution aspects of design methodology," Mar. 2012.
- [15] T.-H. Chen *et al.*, "An Empirical Study on the Practice of Maintaining Object-relational Mapping Code in Java Systems," in *Proceedings of the 13th International Conference on Mining Software Repositories*, 2016, pp. 165–176.
- [16] T. Syed and S. Rabani, "A Novel Temporal Cohesion Complexity Metric for Object-Oriented

- Design,” pp. 2456–3307, Sep. 2017.
- [17] S. Kalantari, M. Alizadeh, and H. Motameni, “Evaluation of reliability of object-oriented systems based on Cohesion and Coupling Fuzzy computing,” *J. Adv. Comput. Res.*, vol. 6, no. 1, pp. 85–99, Feb. 2015.
 - [18] H. B. and S. Hari, “CCOM – A Communicational Cohesion Metric for Object-Oriented Programming,” *Int. J. Comput. Appl.*, vol. 155, no. 5, pp. 1–6, Dec. 2016.
 - [19] H. Razafimahatratra, T. Mahatody, J. P. Razafimandimby, and T. M. B. Andriamahazomandimby, “Automatic detection of the cohesion type in the UML sequence diagram,” in *2018 19th International Carpathian Control Conference (ICCC)*, 2018, pp. 462–467.
 - [20] Y. ; Kim and E. A. Stohr, “Software Reuse: Survey and Research Directions.”
 - [21] R. C. Martin and M. C. Feathers, *Clean code : a handbook of agile software craftsmanship*. Prentice Hall, 2009.
 - [22] J. Al Dallah and Jehad, “Improving the applicability of object-oriented class cohesion metrics,” *Inf. Softw. Technol.*, vol. 53, no. 9, pp. 914–928, Sep. 2011.
 - [23] E. Okike, “A Pedagogical Evaluation and Discussion about the Lack of Cohesion in Method (LCOM) Metric Using Field Experiment,” Apr. 2010.
 - [24] B. B. Agarwal and S. P. Tayal, *Software engineering*. University Science Press, 2008.
 - [25] B. Stiglic, M. Heričko, and I. Rozman, “How to evaluate object-oriented software development?,” *ACM SIGPLAN Not.*, vol. 30, no. 5, pp. 3–10, May 1995.
 - [26] J. Eder, J. Eder, G. Kappel, and M. Schrefl, “Coupling and Cohesion in Object-Oriented Systems,” 1992.
 - [27] M. Page-Jones and Meilir, “Comparing techniques by means of encapsulation and connasence,” *Commun. ACM*, vol. 35, no. 9, pp. 147–151, Sep. 1992.
 - [28] J. Bertin, *Semiology of graphics*. University of Wisconsin Press, 1983.
 - [29] B. B. Bederson and B. Shneiderman, *The craft of information visualization : readings and reflections*. Morgan Kaufmann, 2003.
 - [30] J. Kramer, “Is Abstraction the Key to Computing?,” *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007.
 - [31] R. Spence, “Information Visualisation,” in *Information Visualisation*, Pearson Education, 2001, pp. 134–153.
 - [32] E. R. Tufte and E. R., *Visual explanations : images and quantities, evidence and narrative*. Graphics Press, 1997.
 - [33] W. P. Rogers, “Presidential Commission on the Space Shuttle Challenger Accident,” Orlando, 1986.
 - [34] I. C. Society, P. Bourque, and R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.
 - [35] R. Miles, *Learning UML 2.0 Russ Miles, Kim Hamilton: Books*. O’Reilly Media, 2006.
 - [36] M. Fowler, *UML distilled : a brief guide to the standard object modeling language*, 3rd ed.

- Boston. MA.: Addison-Wesley, 2004.
- [37] D. Berardi, D. Calvanese, and G. De Giacomo, "Reasoning on UML class diagrams," *Artif. Intell.*, vol. 168, no. 1–2, pp. 70–118, Oct. 2005.
 - [38] L. Efrizoni, W. M. N. Wan-Kadir, and R. Mohamad, "Formalization of UML class diagram using description logics," in *2010 International Symposium on Information Technology*, 2010, vol. 3, pp. 1168–1173.
 - [39] Y. Zhou, Y. Yang, B. Xu, H. Leung, and X. Zhou, "Source code size estimation approaches for object-oriented systems from UML class diagrams: A comparative study," *Inf. Softw. Technol.*, vol. 56, no. 2, pp. 220–237, Feb. 2014.
 - [40] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *ACL 2017 - 55th Annual Meeting of the Association for Computational Linguistics, Proceedings of the Conference (Long Papers)*, 2017, vol. 1, pp. 1139–1149.
 - [41] M. Appeltauer and G. Kniesel, "Towards Concrete Syntax Patterns for Logic-based Transformation Rules," *Electron. Notes Theor. Comput. Sci.*, vol. 219, no. C, pp. 113–132, Nov. 2008.
 - [42] JetBrains, "PSI Files," 2019. [Online]. Available: https://www.jetbrains.org/intellij/sdk/docs/basics/architectural_overview/psi_files.html. [Accessed: 27-Sep-2019].
 - [43] K. Bulenkov, "IntelliJ IDEA PSI Viewer," *IntelliJ IDEA Blog*, 2009. [Online]. Available: <https://blog.jetbrains.com/idea/2009/11/psi-viewer/>. [Accessed: 27-Sep-2019].
 - [44] P. Hanks, *Lexical analysis : norms and exploitations*. The MIT Press, 2013.
 - [45] J. Daintith and E. Wright, *A Dictionary of Computing*. Oxford University Press, 2008.
 - [46] A. Palanisamy, A. Pop, M. Sjölund, and P. Fritzson, "Modelica based parser generator with good error handling," in *Proceedings of the 10 th International Modelica Conference; March 10-12; 2014; Lund; Sweden*, 2014, no. 096, pp. 567–575.
 - [47] A. Leung and S. Lerner, "Parsimony: An IDE for example-guided synthesis of lexers and parsers," in *ASE 2017 - Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 815–825.
 - [48] N. Chomsky, "Three models for the description of language," *IRE Trans. Inf. Theory*, vol. 2, no. 3, pp. 113–124, 1956.
 - [49] B. Öttl, G. Jäger, and B. Kaup, "Does formal complexity reflect cognitive complexity? Investigating aspects of the Chomsky Hierarchy in an artificial language learning study," *PLoS One*, vol. 10, no. 4, Apr. 2015.
 - [50] T. Parr, *The Definitive ANTLR 4 Reference*. Pragmatic Programmers, LLC, The, 2013.
 - [51] N. Smith, D. van Bruggen, F. Tomassetti, and C. Maximilien, "JavaParser – For processing Java code," *javaparser.org*, 2019. [Online]. Available: <https://javaparser.org/>. [Accessed: 02-Jul-2019].
 - [52] N. Smith, D. van Bruggen, and F. Tomassetti, *JavaParser: Visited*, 1st ed. Learnpub, 2019.
 - [53] D. R. White, "GI in no time," in *GECCO 2017 - Proceedings of the Genetic and Evolutionary Computation Conference Companion*, 2017, pp. 1549–1550.
 - [54] E. Fauzi, B. Hendradjaya, and W. D. Sunindyo, "Reverse engineering of source code to

- sequence diagram using abstract syntax tree,” in *Proceedings of 2016 International Conference on Data and Software Engineering, ICoDSE 2016*, 2017.
- [55] D. V. Steward, “DESIGN STRUCTURE SYSTEM: A METHOD FOR MANAGING THE DESIGN OF COMPLEX SYSTEMS.,” *IEEE Trans. Eng. Manag.*, vol. EM-28, no. 3, pp. 71–74, 1981.
 - [56] S. D. Eppinger and T. R. Browning, *Design structure matrix methods and applications*. MIT Press, 2012.
 - [57] A. MacCormack, J. Rusnak, and C. Y. Baldwin, “Exploring the structure of complex software designs: An empirical study of open source and proprietary code,” *Manage. Sci.*, vol. 52, no. 7, pp. 1015–1030, 2006.
 - [58] NDepend, “Dependency Structure Matrix,” *NDepend Documentation*, 2019. [Online]. Available: <https://www.ndepend.com/docs/dependency-structure-matrix-dsm>. [Accessed: 04-Mar-2019].
 - [59] P. Smacchia, “Identify Code Structure Patterns at a Glance,” *CodeBetter*, 2009. [Online]. Available: <http://codebetter.com/patricksmacchia/2009/08/24/identify-code-structure-patterns-at-a-glance/>. [Accessed: 13-Oct-2019].
 - [60] R. Cardin, “Calculating Dependency - DZone Java,” *DZone*, 2017. [Online]. Available: <https://dzone.com/articles/dependency>. [Accessed: 14-Sep-2019].
 - [61] A. Kumar and S. K. Khalsa, “Determine cohesion and coupling for class diagram through slicing techniques,” *Int. J. Autom. Control Eng.*, vol. 4, no. 1, pp. 19–24, 2012.
 - [62] V. Krishnapriya and K. Ramar, “Exploring the difference between object oriented class inheritance and interfaces using coupling measures,” in *ACE 2010 - 2010 International Conference on Advances in Computer Engineering*, 2010, pp. 207–211.
 - [63] B. Jonsson, S. Perathoner, L. Thiele, and W. Yi, “Cyclic dependencies in modular performance analysis,” in *Proceedings of the 7th ACM international conference on Embedded software - EMSOFT '08*, 2008, p. 179.
 - [64] B. Foote and J. Yoder, “Big Ball of Mud,” in *Fourth Conference on Patterns Languages of Programs*, 1997.
 - [65] W. Playfair, *The Commercial and Political Atlas, Representing, by means of Stained Copper-Plate Charts, The Progress of the Commerce, Revenues, Expenditure, and Debts of England, during the whole of the Eighteenth Century*. London: T. Burton, 1786.
 - [66] E. Kruja, J. Marks, A. Blair, and R. Waters, “A Short Note on the History of Graph Drawing,” 2002, pp. 272–286.
 - [67] A. Bondy and U. S. R. Murty, *Graph Theory: Graduate text in mathematics*. London: Springer, 2010.
 - [68] N. Beerenwinkel, S. Beretta, P. Bonizzoni, R. Dondi, and Y. Pirola, “Covering Pairs in Directed Acyclic Graphs,” 2014, pp. 126–137.
 - [69] M. Janota, R. Grigore, and V. Manquinho, “On the quest for an acyclic graph,” *arXiv Prepr. arXiv1708.01745*, 2017.
 - [70] E. Kerzner, “Graffinity: Visualizing Connectivity in Large Graphs.,” *Comput. Graph. Forum*, vol. 36, no. 3, pp. 251–261, Jun. 2017.
 - [71] R. Marik, “Efficient genealogical graph layout,” *Stud. Comput. Intell.*, vol. 693, pp. 567–578,

- 2017.
- [72] R. Tamassia, *Handbook of Graph Drawing and Visualization*, 1st ed. Chapman & Hall/CRC, 2016.
 - [73] N. Schonning, G. Hogenson, G. Warren, S. Cai, M. Jones, and M. B, “Directed Graph Markup Language (DGML) reference - Visual Studio | Microsoft Docs,” *Microsoft Visual Studio Docs*, 2016. [Online]. Available: <https://docs.microsoft.com/en-us/visualstudio/modeling/directed-graph-markup-language-dgml-reference?view=vs-2019>. [Accessed: 17-Oct-2019].
 - [74] K. Chandrakant, “Graphs in Java | Baeldung,” *Baeldung Courses*, 2019. [Online]. Available: <https://www.baeldung.com/java-graphs>. [Accessed: 20-Jun-2019].
 - [75] E. Gansner and J. Ellson, “Welcome to Graphviz,” *Graphviz*. [Online]. Available: <https://www.graphviz.org/>. [Accessed: 04-Apr-2019].
 - [76] J. Ellson, J. Ellson, E. R. Gansner, E. Koutsofios, S. C. North, and G. Woodhull, “Graphviz and dynagraph – static and dynamic graph drawing tools,” *GRAPH Draw. Softw.*, pp. 127–148, 2003.
 - [77] E. R. Gansner and S. C. North, “Improved force-directed layouts,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1999, vol. 1547, pp. 364–373.
 - [78] A. Urbán and R. Hirz, “UMLet - Free UML Tools for fast UML diagrams,” *UMLet*, 2018. [Online]. Available: <https://www.umlet.com/>. [Accessed: 20-Oct-2019].
 - [79] M. Auer, T. Tschurtschenthaler, and S. Biffl, “A flyweight UML modelling tool for software development in heterogeneous environments,” in *Conference Proceedings of the EUROMICRO*, 2003, pp. 267–272.
 - [80] M. Auer, L. Meyer, and S. Biffl, *Explorative UML Modeling - Comparing the Usability of UML Tools*. 2007.
 - [81] M. Bastian, S. Heymann, and M. Jacomy, “Gephi: An Open Source Software for Exploring and Manipulating Networks,” *ICWSM*, vol. 8, pp. 361–362, Jan. 2009.
 - [82] B. Naveh, J. Sichi, J. Kinable, and D. Michail, “JGraphT README,” *JGraphT*, 2019. [Online]. Available: <https://github.com/jgrapht/jgrapht/blob/master/README.md>.
 - [83] J. O’Madadhain, J. O’Madadhain, D. Fisher, S. White, P. Smyth, and Y. Boey, “Analysis and Visualization of Network Data using JUNG.”
 - [84] M. J. Chonoles, “What is UML?,” in *OCUP Certification Guide: UML 2.5 Foundational Exam*, Elsevier, 2018, pp. 17–41.
 - [85] M. Adams, “5 of the best software for UML diagrams to use on Windows 10,” *WindowsReport*, 2018. [Online]. Available: <https://windowsreport.com/uml-diagram-software/>. [Accessed: 23-Oct-2019].
 - [86] S. Brown, “The C4 Model for Software Architecture,” *InfoQ Architecture and Design*, 2018. [Online]. Available: <https://www.infoq.com/articles/C4-architecture-model/>. [Accessed: 23-Oct-2019].
 - [87] N. Schonning, G. Warren, G. Hogenson, S. Cai, M. B, and R. McAllister, “Generate code from UML class diagrams,” *Visual Studio 2015 | Microsoft Docs*, 2016. [Online]. Available: <https://docs.microsoft.com/en-gb/visualstudio/modeling/generate-code-from-uml-class-diagrams?view=vs-2015>. [Accessed: 24-Oct-2019].

- [88] A. Fronk, D. Gude, and G. Rinkenauer, "Evaluating 3D-Visualisation of Code Structures in the Context of Reverse Engineering," Jan. 2006.
- [89] E. Dietrich, "5 Tips to Help You Visualize Code," *NDepend Website*, 2017. [Online]. Available: <https://blog.ndepend.com/5-tips-help-visualize-code/>. [Accessed: 08-Aug-2019].
- [90] "Sourcetrail Documentation for Version 2019.3," *Sourcetrail*, 2019. [Online]. Available: <https://www.sourcetrail.com/documentation/>.
- [91] Hello2Morrow, "Sonargraph Product Family," *SonarGraph Products*, 2019. [Online]. Available: <https://www.hello2morrow.com/products/sonargraph>.
- [92] E. Wagner and A. von Zitzewitz, "How BMW Improved Software Quality and Agility of a Mission Critical Software System," 2015.
- [93] "CodeMR User Guide v2019," *CodeMR documentation*, 2019. [Online]. Available: <https://www.codemr.co.uk/documents/>.
- [94] D. Clegg and R. Barker, *Fast-track : a RAD approach*. Addison-Wesley Pub. Co., 1994.
- [95] R. A. Serway and C. Vuille, *College physics. Volume 2*. 2014.
- [96] J. Sauro, *A Practical Guide to the System Usability Scale: Background, Benchmarks & Best Practices*. 2011.

8 Appendices

8.1 Appendix A – User Task List

Task 1

1. Run the application
2. Upon first input prompt – *‘Enter q to quit or enter directory’* – enter the first memory location string, corresponding to the dependency project repository
3. Upon second input prompt – *‘Enter a depth parameter to set maximum number of package layers returned (if required)’* – press enter
4. Upon third input prompt – *‘Please enter a visualisation command code (if required)’* – press enter
5. Observe the automatically generated graph image
6. Observe the automatically generated Dependency Structure Matrix, which represents numerical dependency strength between different packages.
7. Mouse over different elements of the DSM to see which dependencies they represent
8. Close the table and image

Task 2

1. Upon first input prompt – *‘Enter q to quit or enter directory’* – enter the second memory location string, corresponding to the driver-card service
2. Upon second input prompt – *‘Enter a depth parameter to set maximum number of package layers returned (if required)’* – press enter
1. Upon third input prompt – *‘Please enter a visualisation command code (if required)’* – enter ‘i’ to remove external libraries
3. Observe the automatically generated graph image
4. Observe the automatically generated Dependency Structure Matrix, which represents numerical dependency strength between different packages.

5. Mouse over larger elements of the DSM to see which dependencies they represent
6. Close the table and image

Task 3

2. Upon first input prompt – *'Enter q to quit or enter directory'* – enter the third memory location string, corresponding to the driver-card workspace
3. Upon second input prompt – *'Enter a depth parameter to set maximum number of package layers returned (if required)'* – enter 7
4. Upon third input prompt – *'Please enter a visualisation command code (if required)'* – enter 'i' to remove external libraries
5. Observe the automatically generated graph image
6. Zoom in on the different sections of the graph using the magnifying glass icon in the image preview
7. Observe the automatically generated Dependency Structure Matrix, which represents numerical dependency strength between different packages.
8. Mouse over larger elements of the DSM to see which dependencies they represent
9. Close the table and image
10. Quit the application via the input terminal

Task 4

Run the application

Investigate the project stored within the memory location represented by the fourth path string, i.e. the open source teammates project. Enter different depth parameter values (based on the number of dot separated elements in each package name) to examine the dependency structure at different resolutions. View only internal dependencies using the 'i' command or leave external dependencies in by entering nothing during that step.

8.2 Appendix B – User Questionnaire

- How long have you worked as a developer?
- How long have you worked as a developer with the DVLA?
- What's your general position at the DVLA?
- On a scale of 1 to 7, where 1 is "Did not understand at all" and 7 is "Understood perfectly", circle the number representing how much you understood the format of the dependency graph.

Did not understand at all	1	2	3	4	5	6	7	Understood perfectly
---------------------------	---	---	---	---	---	---	---	----------------------

- On a scale of 1 to 7, where 1 is "Did not understand at all" and 7 is "Understood perfectly", circle the number representing how much you understood the format of the dependency structure matrix.

Did not understand at all	1	2	3	4	5	6	7	Understood perfectly
---------------------------	---	---	---	---	---	---	---	----------------------

- What can we do to make these more understandable?
- Thinking only of tasks 2 and 3, on a scale of 1 to 7, where 1 is "Did not match at all" and 7 is "Matched perfectly", circle the number representing how closely the model matched any pre-existing understanding you might have of the tachograph driver-card packages.

Did not match at all	1	2	3	4	5	6	7	Matched perfectly
----------------------	---	---	---	---	---	---	---	-------------------

- If anything did not match, what was it?
- Thinking only of tasks 1 and 4, On a scale of 1 to 7, where 1 is "Imparted no understanding" and 7 is "Imparted full understanding", circle the number representing how much understanding of the basic dependency structure of the new project was imparted to you?

Imparted no understanding	1	2	3	4	5	6	7	Imparted full understanding
---------------------------	---	---	---	---	---	---	---	-----------------------------

- What can we do to improve this?
- On a scale of 1 to 7, where 1 is “Not at all” and 7 is “Very helpful”, circle the number representing how useful this prototype, or future expanded and refined versions of it, might be to help identify areas of high dependency within a project, or unwanted dependency patterns?

Not at all	1	2	3	4	5	6	7	Very helpful
------------	---	---	---	---	---	---	---	--------------

- What, if any, modifications to the user interface or additional features of the system, might you want to see in subsequent releases (prioritised based on feasibility and importance)?

- On a scale of 1 to 7, where 1 is “Not at all” and 7 is “Lots of potential”, circle the number representing the potential of this prototype, or future expanded and refined versions of it, might be for use at the DVLA?

Not at all	1	2	3	4	5	6	7	Very helpful
------------	---	---	---	---	---	---	---	--------------

8.3 Appendix C – System Usability Scale (SUS)

	Strongly disagree						Strongly agree
1. I think that I would like to use this system frequently	☐	☐	☐	☐	☐		
	1	2	3	4	5		
2. I found the system unnecessarily complex	☐	☐	☐	☐	☐		
	1	2	3	4	5		
3. I thought the system was easy to use	☐	☐	☐	☐	☐		
	1	2	3	4	5		
4. I think that I would need the support of a technical person to be able to use this system	☐	☐	☐	☐	☐		
	1	2	3	4	5		
5. I found the various functions in this system were well integrated	☐	☐	☐	☐	☐		
	1	2	3	4	5		
6. I thought there was too much inconsistency in this system	☐	☐	☐	☐	☐		
	1	2	3	4	5		
7. I would imagine that most people would learn to use this system very quickly	☐	☐	☐	☐	☐		
	1	2	3	4	5		
8. I found the system very cumbersome to use	☐	☐	☐	☐	☐		
	1	2	3	4	5		
9. I felt very confident using the system	☐	☐	☐	☐	☐		
	1	2	3	4	5		
10. I needed to learn a lot of things before I could get going with this system	☐	☐	☐	☐	☐		
	1	2	3	4	5		