

Prototyping a codebase visualisation
system for use in the software
engineering field

Research Proposal

Salih Osman

Contents

Contents

Contents.....	2
Figures.....	3
Introduction and Background to the Proposed Research	4
Designation of a working title for proposed research	4
Description of the background to the subject area of study	4
Object oriented programming and related structural concepts.....	4
The visualisation of source code dependencies	5
Provision of an understanding of the field in which the study lies and an idea of where and how the study fits into this field	7
Introduction of the relevant the industry/organisation/context that is being investigated	8
Overarching aim and series of objectives	8
Expected outcomes and/or deliverables	8
Research and Development Methods	9
Research Methods	9
Philosophy.....	9
Strategies	9
Study sample and data collection approaches	9
Development methods	10
Agile methodology	10
Object oriented programming and separation of concerns	10
Identification of risks and limitations	10
Data Evaluation	11
Resources & Planning	12
Resources Required	12
Gantt Chart	12
7. Reference List.....	13

Figures

Figure 2: The demarcated sections demonstrate internal cohesion	5
Figure 2: An illustration of the concept of coupling between classes or packages	5
Figure 3: pre-abstraction and post abstraction representations of the London Tube Map.....	6
Figure 4: CodeMR example dependency graphs	8
Figure 5: Gantt chart representing project plan	12

Introduction and Background to the Proposed Research

Before attempting to engage in more detailed discussion of the research project and its basis, it is first necessary to designate a title for said project, with appropriate justification.

Designation of a working title for proposed research

The current title for the research project detailed subsequently is: “Prototyping a codebase visualisation system for use in the software engineering field”. The basis for this title lies in the motivation for the project, which originates in observation, within an industry setting, of common issues regarding the understanding and display of source code developed by software engineering squads. More specifically, as suggested by the term “visualisation”, the project seeks to explore and analyse a new technique or system by which the collected body of source code may be represented and viewed. As made clear within the title, this system must be developed in prototype form, in accordance with the iterative and incremental principles of agile software development, in order to prevent failure of scope.

Description of the background to the subject area of study

Having established the title of the proposed project and given a brief introduction to the reasoning behind it, the next step is to establish a more conceptual or theoretical background for the general topics addressed by the project.

Object oriented programming and related structural concepts

The advent of object-oriented programming has been hailed as one of the most significant innovations within the field of computing, regarded by some as the next stage in the “evolution” of programming languages [1]. This is because it yielded a change in structure from the previously accepted format of procedural programming. Instead of a unified code script being run in sequential order, object-oriented programming relies upon the separation of code into discrete units, conventionally known as classes, which are intrinsically correlated, and extrinsically distinct [2]. These characteristics, which designate the strength, type and complexity of relationships between and within these discrete units, are defined by the terms brought into use in the 1960s by Larry L. Constantine: cohesion and coupling [3].

The meaning of the word cohesion in relation to the discrete units within an object-oriented program, as delineated by Constantine, essentially represents the strength of the relationship between the different constituent elements, or subsections of a given class or package [4]. Put another way, high cohesion implies a class or package, the constituent elements of which are unified in purpose or function towards a particular goal, as shown in Figure 1 [5]. Low cohesion, on the other hand, would signify that there is significant variance in functionality within the class or package, with multiple disparate aims being served. In a parallel manner, coupling refers to the degree of linkage, or interdependence between different classes or packages, see figure 2 [6]. The more closely dependent a particular class is on another class, the more “tightly coupled” those two classes are said to be, the more self-contained and independent each class or package is of its fellows, the more “loosely coupled” it is said to be. As such we arrive at a generalised pair of statements which concern the quality of object-oriented source code, in relation to the previously defined terms. In other words, “high cohesion and low coupling is desirable”, “low cohesion and high coupling is detrimental” a codebase which exhibits the former is known to be superior, structurally and functionally, to a codebase which exhibits the latter [7]. This clear criterion for assessing the quality of the code based on its structure, is a concept of significant importance within the proposed research project.

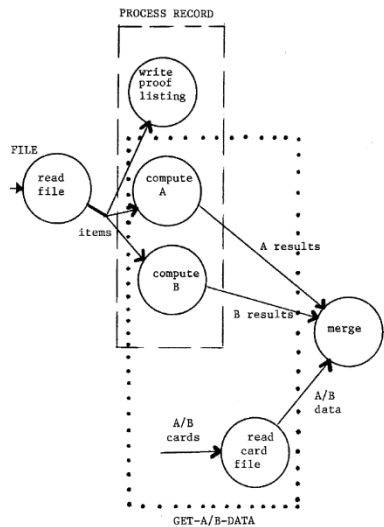


Figure 2: The demarcated sections demonstrate internal cohesion

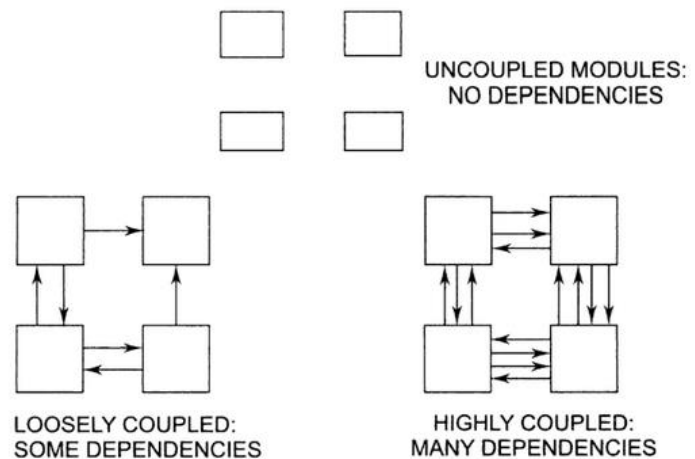


Figure 2: An illustration of the concept of coupling between classes or packages

In addition to the concepts of coupling and cohesion, some mention should be made of the numerous different methods by which the classes or packages within a given codebase can reference and relate to each other. Within the Java programming language, which is the language chosen for the development of the visualisation system, two modules may be linked by: inheritance, implementation of interfaces, direct instantiation within one class of the object representation of another class, or direct access to the variables or functions of one class by another [8].

Concordantly, it is important to bear in mind that the choice of implementation of any of these methods within a codebase might lead to either desirable or detrimental levels of dependency within that system, with certain species of dependency being favourable over others [9]. It is as a result of this that the visualisation of intra-codebase dependencies attains significance.

The visualisation of source code dependencies

It has been established that the nature of the dependency structure within a codebase is one of the essential features that must be taken into consideration when establishing, and attempting to effect change on, the quality and scalability of the system as a whole. From this, it can be interpreted that any technique which will allow a more clear understanding of, and appreciation for, the fundamentals of this dependency structure, is a technique of significant utility in the development of effective code.

The most basic understanding of this aspect of the source code can be obtained simply by direct inspection, or the most fundamental “static analysis”, of the code modules and the in-line invocation separate modules have for each other. However, this can immediately be seen as both intrinsically unreliable, due to the unintuitive nature of the interface with respect to achieving an understanding of the large scale structure, and also requiring a significant degree of familiarity with the source code, which, quite simply, cannot be ensured. This point is well summarised by Bacchelli and Bird [10], in their statement “code review is understanding”, highlighting the impossibility of doing away with the need for deep understanding of the motivation and context for specific features of, and changes to, the codebase. Furthermore, research presented by Hidetake Uwano, Masahide Nakamura, Akito Monden and Ken-ichi Matsu, in their paper: “Analyzing individual performance of source code review using reviewers' eye movement”, exposes that subtle intrinsic factors, such as briefer “scan” period, can lead to significant decreases in effectivity of manual code review [11].

Drawing on a number of the points made previously, the argument can clearly be made that a more intuitive, high level technique for visualising source code dependencies would be of significant benefit. It is essential that, in order to avoid the issues mentioned above, such a technique must correspond with sufficient implementation of the principle of abstraction.

Abstraction, as laid out by Jeff Kramer, is the word which describes the practise of extracting the specific features and elements of a system so as to decrease its overall intricacy, enabling a clearer and more immediate understanding of the fundamental essence, or attribute of the system which is being analysed [12]. This concept is shown more tangibly in Figure 3, with displays the example of the London Tube Maps, before and after the abstraction of a number of unnecessary details from their contents by Harry Beck in 1931 [12]. In essence, the indispensable details of the London tube map, at least where the general public is concerned, can be reduced to simply the structure of the connections between different stations.

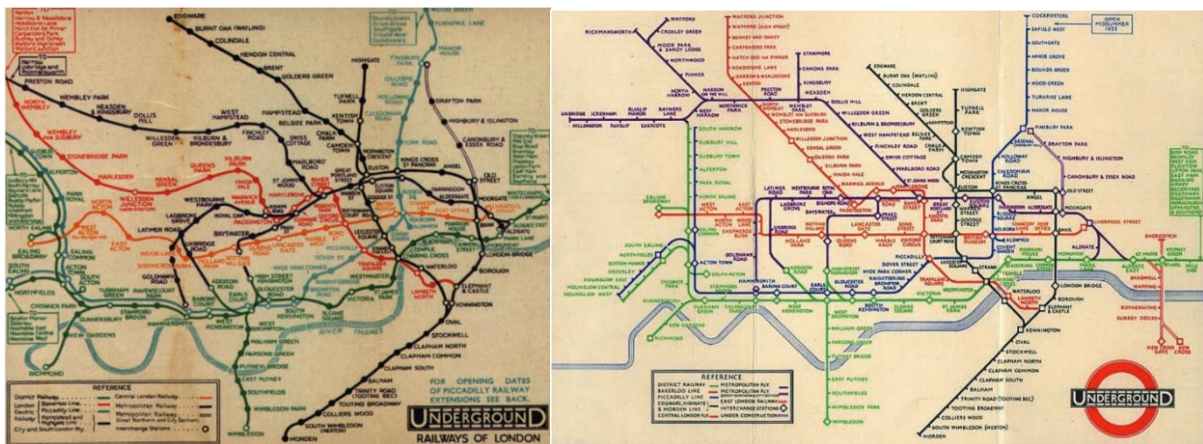


Figure 3: pre-abstraction and post abstraction representations of the London Tube Map

The logical analogy can be drawn between the presence of extraneous detail within the tube map, and the presence of extraneous detail within the basic Integrated Development Environment view of an object-oriented codebase. This analogy can be further expanded when one takes into consideration the fact that the fundamental and essential detail of each of the two systems is the same, in other words: the connectivity. A natural conclusion would be to represent the codebase in a similar, minimalistic fashion, that is via the application of graph theory, as discussed by R. Spence in “Information Visualisation” [13].

The “Beck-like” solution to the code representation problem, then, might be to simply characterise the codebase as a mathematical graph. That is, each module of the object-oriented source code would represent a node, and each connection between modules would represent an edge. The resulting graph would be directional, due to the directional nature of each dependency. This is in accordance with the details laid out by F. Balmas with regards to a hierarchical approach to dependency visualisation [14].

A common alternative and comparable form of visualisation for codebase structure involves the use of the Unified Modelling Language (UML). A UML class diagram separates out the individual classes as the fundamental elements of the diagram and uses different connectors to illustrate variant forms of relationship between classes, for example composition, inheritance or aggregation [15]. The concept of using UML diagrams to represent code structure is a well-known and well-established one, within the field of Software Engineering, as it has long been used within the software design process [16].

Provision of an understanding of the field in which the study lies and an idea of where and how the study fits into this field

As addressed in the section immediately previous to this, the use of diagrams to represent code dependency structures is not a new one. Most traditionally, techniques of this nature have been used prior to development and deployment of the software, as a design technique to enable visualisation of the “skeleton” around which the code would be constructed. However, examples of the benefits of using codebase visualisation techniques, as being proposed in this project, as a post-hoc method of understanding an existing codebase, are evident as far back as 1993. Eick and Wills indicated the beneficial impacts of an edge and node graph representing the relationships between functions, in their paper: “Navigating large networks with hierarchies” [17].

One of the aspects which serves to make the proposed system somewhat more unique in comparison to the aforementioned practises, is that the generation of the graph structure would be handled in an entirely automated fashion. A proposed reason for the lack of adoption and use of source code dependency visualisation techniques is the difficulty and relatively significant time expenditure required to manually create a useful graph. There exist numerous websites whose purpose is to somewhat ease this process, and enable a developer to manually enter their codebase structure in simple syntax form, such as the browser version of the open source graphviz software visualisation system: webgraphviz [18].

However, there are a number of packages which deal with automatic generation of dependency information and graphical representations of codebase structure from input source code. Perhaps the most well-established and prolific of these would be Ndepend, a software visualisation tool which incorporates the building of directional dependency graphs with the generation of a design structure matrix (or dependency structure matrix, DSM) which provides a more mathematical, statistics oriented codebase representation [19]. However, this analysis tool is only applied to the .Net languages, such as C, C++, C# etc., excluding Java, which is the base programming language used within a large number of organisations, including the Driver and Vehicle Licensing Agency (DVLA), around which this project is based.

When it comes to Java, there are a number of different options available with respect to automatic source code dependency visualisations. Examples of these include a number of plugins for the IntelliJ integrated development environment, with the most well reviewed currently being the CodeMR plugin for Java. As shown in Figure 4, the visualisations produced by the CodeMR plugin follow the graph abstract graph structure described previously [20].

However, as appealing as the imagery shown in Figure 4 is, there is a caveat to the use of the CodeMR plugin, just as with many of the other dependency visualisation systems currently available. The majority are priced, and although CodeMR has a non-corporate individual use plugin version which is free, it is limited in scope. Attempting to extract more than 50 source files will result in a message indicating that this is only possible on the licensed version of CodeMR, which is far from inexpensive.

Thus one of the key attributes which motivates this project is the prototyping of a source code dependency visualisation system which can be taken up by developers in any environment or scenario. The advantage of ease of use and ease of access is one of the key features in the uniqueness of the proposed project. Another salient point is that, in order for a piece of technology such as the system being developed as part of this project to take effect, it may need to be adopted by a non-trivial proportion of the developers within a squad, in order to reach a form of “market penetration” point, becoming widely recognised and understood.



Figure 4: CodeMR example dependency graphs

Introduction of the relevant the industry/organisation/context that is being investigated

The proposed project is essentially being carried out under funding from the DVLA. As such, the project is targeted at providing a functional insight into possible technologies which may provide a tangible benefit to the aims of that organisation. The context, then, is predominantly one of improving the codebase comprehension and management of Agile scrum style development squads, working at scale with the Java programming language.

Overarching aim and series of objectives

The overarching aim of the proposed project is: to explore the effectiveness of a prototype system which will automatically generate a representative visualisation of an input codebase. In order to realise this goal, the following objectives must be addressed:

1. To analyse and determine best practise techniques for the implementation of a codebase visualisation system written in java
2. To develop, in an agile manner, a software engine capable of parsing a codebase, extracting dependency information and storing it in a suitable memory structure.
3. To similarly develop a platform-agnostic visualisation system capable of displaying codebase dependency data in an understandable format.
4. To determine the potential of the prototype system in facilitating the software development process via practical research

Expected outcomes and/or deliverables

There are two primary outcomes which can be anticipated from the completion of this project:

1. A prototype of an automatic system for generating dependency visualisations based on an input codebase will have been designed and built
2. Research will have been done to get feedback from developers as to the current and future potential of such a system.

Research and Development Methods

The motivation of the proposed project lies in a novice programmer's initial attempts to comprehend some of the sizeable codebases within the DVLA. The open question asked was "What might be a better way to understand this other than simply looking at the codebase?". This question was then refined and subdivided relative to the concepts of automatic generation of dependencies, and graph theory, until it became "Is it possible to prototype a dependency graphing application which will display the source code structure more intuitively?". It was in this vein that this primary, qualitative research project was proposed.

Research Methods

Philosophy

Although the proposed project will exhibit features of both the positivist and phenomenological research philosophies, it is weighted towards the phenomenological side, with a more inductive than deductive approach. This is because, although the project will rely on the generation of a real-world, non-subjective, software system, many of the defining attributes of that system are inherently subjective. This is because concepts such as "usability", "usefulness", "intuitiveness" etc, all characteristics of a visual representation of structure, are not intrinsically objective. As such, it was judged that the most appropriate path to take was one that worked with the fact that the data is more likely to be qualitative rather than quantitative. The phenomenological philosophy and inductive approach match this, with information-rich data being collected via interviews rather than experiment, for the sake of developing a theory, rather than to prove or disprove an existing hypothesis.

Strategies

Three research strategies will be utilised in the execution of this project. A literature or document review will be essential in determining best practice for the development of the software system, as well as for comparison and contrast against contemporary research and choice of direction on issues such as scope. In addition to this the software development strategy will be used, as discussed this has the hallmarks of a more positivist project, but remains more phenomenological due to the nature of the data that it aims to collect. It is hoped that the codebase visualisation software developed as part of this project will yield a valuable insight into potential methods to improve code structure and understanding of code structure at the DVLA. Lastly, the survey strategy will be utilised, with questionnaires and interviews being supplied to members of the DVLA development squads. This will be central to obtaining the volume of subjective data necessary to construct a statement or theory about the effectiveness of automated code visualisation techniques at the DVLA.

Study sample and data collection approaches

The individuals being interviewed will consist predominantly of a mix of technical leads, senior and junior software engineers, with the potential to include some agile delivery managers and business analysts. There will be two main approaches to data collection from these individuals. Towards the beginning of the research project, questionnaires will be carried out to determine data such as role, degree of experience within software engineering etc. (predominantly metadata). Subsequently, in correlation with the development of the visualisation software, a series of interviews will be carried out with the participants in the questionnaire to obtain qualitative data on the potential of the prototype system at the DVLA.

Development methods

As there is no purely experimental component to the research project, the only relevant methods consist of software development methodologies.

Agile methodology

One of the key aspects of this project is that it must be undertaken using an agile approach. In other words, the development process must be not only incremental but also iterative. The use of a more traditional methodology such as waterfall or the V-model would open the door to a number of potential problems. Predominant among these would be the threat of designing the system, having obtained specification etc, developing it and then reaching the deployment stage, only to realise that the project design was misguided or flawed. This would irrecoverably impact the entire project lifecycle.

The use of Agile development, on the other hand, will allow for short, sprint-style development cycles, targeted towards a Minimum Viable Product (MVP), with the ability to change direction depending on input from survey or literature review research. This approach also has the advantage of being concordant with the DVLA's established work practices. However, one potential problem associated with the agile methodology is feature creep. It will be important to maintain strict control of project objectives, in order to avoid going beyond scope and potentially negatively affecting the research process. The development of the system as a prototype is a key aspect of this project, an attempt to broaden into a full commercial product runs the risk of getting bogged down in data visualisation complexities.

Object oriented programming and separation of concerns

Ideally the system developed during this project will be entirely platform independent, and highly scalable and extensible. Such a system can only be developed in conjunction with stringent application of the principle of separation of concerns. Best practices for Object Oriented programming, such as ensuring low coupling and high cohesion, will also be instrumental in achieving this goal. However, it is also possible that too heavy of a focus on the possible future of the system, might limit its effectivity in the short term. As such it will be important to bear in mind that an absolutist approach to separation of concerns and extensibility could subvert the timescale for the prototype to be effective.

Identification of risks and limitations

There are two key risks which can be identified before engaging in the proposed project. Firstly, it is essential that the project be carried out in a stage by stage fashion as mentioned. If this is not the case, a major risk is the possibility of getting overwhelmed by the extent of the project, as it would be difficult to isolate an appropriate starting place. Secondly, it is important that the interview questions are well thought out and well structured, otherwise the research process risks becoming too disjointed without generalizable results. Lastly, one limitation that should be born in mind for the completion of this project, is time. It would be easy to neglect the large proportion of time allocated to work at the DVLA during the same timeframe of this project. As a new entry into the field of software engineering, the researcher cannot allow himself to underestimate the amount of time it will take him to complete basic programme requirements, as this would negatively impact the project timeline.

Data Evaluation

The data must be sourced so as to be fundamentally self-comparable. In other words, interview questions should allow participant engineers to draw comparisons and contrasts between their accustomed methods of understanding codebase dependencies and those offered by the prototype system, as well as allowing them to speculate on potential improvements/expansions that could be made.

Comparisons can also be drawn for reviews of existing visualisation technologies, such as NDepend and CodeMR. This will be useful in analysing scope of future iterations of the prototype system.

Resources & Planning

Resources Required

The resources required for the project are not anticipated to be extensive. A DVLA development laptop will be used for the purpose of building, running and testing the visualisation system. Access to the Java programming language and the IntelliJ development environment will be of significant importance. However, these resources are freely available.

Most crucial, perhaps, will be access to human resources in the form of DVLA engineers to interview and give questionnaires to. This will be possible via access to the DVLA main site, and has been cleared with the relevant line managers at the DVLA.

Gantt Chart

The Gantt chart below represents the current plan for completion of the proposed project.

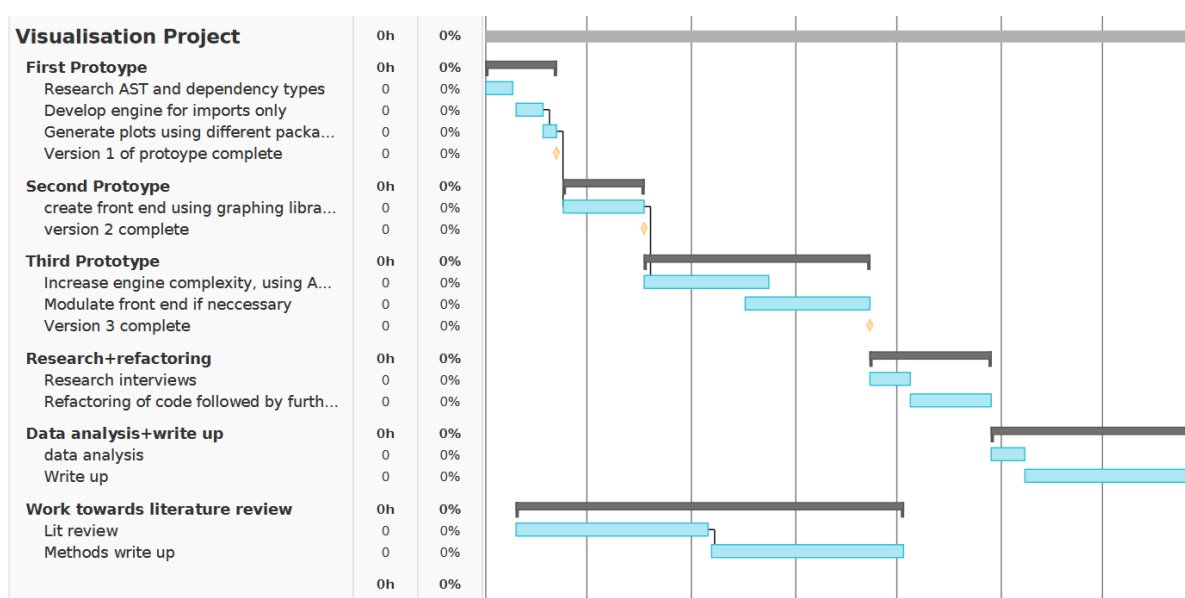


Figure 5: Gantt chart representing project plan

Conclusion

Ultimately, this proposal has attempted to shed light on some of the background features behind the proposed research, such as the intrinsic importance of structure and dependency patterns on key code quality observables such as cohesion and coupling. From this it has made the argument for the important of dependency visualisations, hinging on the principle of abstraction and following established patterns, such as directional graphs of nodes and edges. A background of existing research and contemporary applications has been provided, and contrasted with the overarching aims and objectives of this project. This proposal has discussed the research philosophies, strategies and methods which will be implemented over the course of this project, as well as the practical development methods. Lastly, there has been analysis of potential risks and limitation of the project, as well as resource and planning analysis.

7. Reference List

- [1] S. Hirshfield and R. K. Ege, "Object-oriented programming," *ACM Comput. Surv.*, vol. 28, no. 1, pp. 253–255, Mar. 1996.
- [2] T. Dyke, "Object-Oriented Programming," 1989.
- [3] J. A. N. Lee, *International biographical dictionary of computer pioneers*. Fitzroy Dearborn, 1995.
- [4] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Syst. J.*, vol. 13, no. 2, pp. 115–139, 1974.
- [5] E. Yourdon and L. L. Constantine, "Structured design : fundamentals of a discipline of computer program and systems design," Prentice Hall, 1979, pp. 95–126.
- [6] B. B. Agarwal, S. P. Tayal, and M. (Mahesh) Gupta, *Software engineering & testing : an introduction*. Jones and Bartlett, 2010.
- [7] L. Ouyang, B. Zou, M. Qu, and C. Zhang, "A method of ontology evaluation based on coverage, cohesion and coupling," in *2011 Eighth International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, 2011, pp. 2451–2455.
- [8] N. Walkinshaw, M. Roper, and M. Wood, "The Java system dependence graph," in *Proceedings Third IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 55–64.
- [9] J. Jenkov, "Understanding Dependencies." [Online]. Available: <http://tutorials.jenkov.com/ood/understanding-dependencies.html#summary>. [Accessed: 31-May-2019].
- [10] 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.
- [11] 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*, 2006, pp. 133–140.
- [12] J. Kramer, "Is Abstraction the Key to Computing?," *Commun. ACM*, vol. 50, no. 4, pp. 36–42, Apr. 2007.
- [13] R. Spence, "Information Visualisation," in *Information Visualisation*, Pearson Education, 2001, pp. 134–153.
- [14] F. Balmas, "Displaying dependence graphs: a hierarchical approach," in *Proceedings Eighth Working Conference on Reverse Engineering*, pp. 261–270.
- [15] S. Bennett, *Schaum's outline of UML*, 2nd ed. New York ; London: McGraw-Hill, 2005.
- [16] A. S. Evans, "Reasoning with UML class diagrams," in *Proceedings. 2nd IEEE Workshop on Industrial Strength Formal Specification Techniques*, pp. 102–113.
- [17] S. G. Eick and G. J. Wills, "Navigating large networks with hierarchies," in *Proceedings Visualization '93*, pp. 204–210.
- [18] J. Ellson, Emden Gansner, Yifan Hu, Erwin Janssen, Stephen North, and Et Al, "Graphviz - Graph Visualisation," 2016. [Online]. Available: <http://www.graphviz.org/>. [Accessed: 31-May-2019].

- [19] J. Novak and G. Rakić, "Comparison of software metrics tools for: net," in *Proc. of 13th International Multiconference Information Society-IS, Vol A*, 2010, pp. 231–234.
- [20] CodeMr Development Team, "CodeMR : Code quality measurement and visualisation," 2019. [Online]. Available: <https://www.codemr.co.uk/>. [Accessed: 31-May-2019].

