From Code to Design: A Reverse Engineering Approach

Elliot Varoy*, John Burrows*, Jing Sun† and Sathiamoorthy Manoharan†

*Department of Electrical and Computer Engineering
†Department of Computer Science
The University of Auckland, New Zealand
Emails: *{evar872, jbur236}@aucklanduni.ac.nz, †{j.sun, mano}@cs.auckland.ac.nz

***Abstract*—Understanding existing pieces of software is a challenge faced by many software developers regardless of their experience. This project researches into existing reverse engineering tools used for code comprehension and identifies the limitations of the current approaches. Furthermore, a prototype implementation was developed to extract design models from available source code in order to achieve better program comprehension. The design and implementation of the model extraction tool were defined, with a focus on Java systems and an agile methodology. This tool was realised by extending the existing open source diagrammatic software, UMLet, with a set of features to aid in code comprehension. Finally, the prototype implementation was evaluated against the related tools in the field as well as by a group of professional experts.**

I. Introduction

The ability to comprehend existing codebases is a skill required by software engineers of all levels. However, understanding another developer's software is a difficult task that adds a large amount of overhead when modifying and extending legacy applications. There are often a wide range of dependencies riddled throughout the codebase, and analysing these by reading through multiple source files and lines of code is extremely inefficient. Reverse Engineering [1] is the process of analysing existing software to create representations of the system at a higher level of abstraction. It is an important technique in the software development process, especially during maintenance, refactoring, upgrading, etc.

The aim of this project is to develop a prototype tool for extracting design models from program source code in order to gain better code comprehension. It enables software developers to obtain a much clear understanding of an existing piece of software that they are required to work with. The educational uses of the tool can provide consideration in the design and implementation process [2], [3]. Hence, the objective of the tool is to create a high level overview of the software architecture design, from an existing system, to encapsulate the design patterns used and the flow of control passing through the software [4]. This can be achieved by analysing the state-of-art problem space through research into existing comprehension strategies, tools and the languages used. The initial research can be utilised to identify any potential features that should be included in the development of the proposed tool, or any current limitations with the existing solutions to the code comprehension problem. Finally, the developed tool should be evaluated for effectiveness by comparing against the goals of the project but also against competing tools.

The rest of the paper is structured as follows. Section 2 reviews the existing code comprehension strategies and tools. In Section 3, we present the design and implementation of the proposed extension to UMLet. Section 4 evaluates the prototype system by comparisons to related tools as well as feedback from professional reviews. In Section 5, we present the conclusion and outline the future work.

II. Comprehension Strategies and Tools

The high level artefacts of a software system are usually documented by either notational descriptions or diagrammatic models. Our research began by exploring what code comprehension strategies and tools were currently available.

A. Code Comprehension

The Unified Modelling Language (UML) is a modelling language made specifically for software engineering. The motivation behind its creation was to create a standard across the industry for visualising the design of a system [5]. UML has an international standard specified by the Object Management Group (OMG) as ISO standard (ISO/IEC 19501). It is primarily based around the use of diagrams which are split into two core types, i.e., Structural and Behavioural. Structural diagrams are best used to explain the static architecture of a system and how its components are linked [6]. Behavioural diagrams, however, are focused on the functionality of a system and explain its operation and workflow. Both of these are crucial for gaining an understanding of a software application.

The Control Flow Graph (CFG)[7] is an alternate strategy to display the flow of control in a system in comparison to UML's behavioural tactics. All paths that an application will pass through during its execution are recorded on a node diagram. This diagram will then represent how sections of code are linked during execution along with the dependencies they have on each other. While this is helpful for gaining an understanding of how the system operates, it also offers the benefits of locating code which is not used during execution. This provides the added benefit of not only extending the system, but also modifying the existing code to improve its

testability and maintainability[8]. CFGs also introduce the idea of cyclomatic complexity, which describes the number of independent paths the system execution follows. This is more beneficial when modifying code, as it is a way of quantifying the quality of a built system, where minimising the code complexity can lead to improved readability and structure.

Code Metrics [9] are a set of measures which aim to give developers more clarity over their code. Some of these metrics are, the maintainability index, cyclomatic complexity, depth of inheritance, class coupling and lines of code. There has been some debate in the software community as to whether analysing code using metrics actually provide any value to developers. However they continue to be used by leading organisations such as NASA [10].

B. Existing Tools

In order to assess the usability of these comprehension strategies, an evaluation was conducted on the existing reverse engineering tools available. The goal of this research was to find both the benefits and the downfalls of these technologies in hope to either extend one or use this information for the development of a new application. Our investigation covered primarily the following five applications.

JArchitect is a very expansive and complex reverse engineering tool. It provides over 80 code analysing metrics ranging from counts of types, lines or comments, to calculating method complexities or producing dependency management graphs. This software is aimed at the company-level projects, requiring a deep understanding of code analysis and has a high barrier of learning which can make it difficult for beginners to use. JArchitect is a paid application, so extending the source code is not a viable option, and this is also seen as another barrier to entry for the use of this software. In addition, JArchitect has a lack of focus towards UML.

eUML2 was another tool reviewed during research. The basis for eUML2 is to provide an Eclipse (A common Integrated Development Environment used for Java) Plugin which provides the ability to both create UML diagrams and have them generated from existing code. This is an ideal solution that fits with our intended scope. However, there are some core issues. The installation and use of eUML2 is a frustrating and often futile endeavour. We encountered our own use issues, along with countless other calls for support in online forums. eUML2 is also very invasive software, that will add a large amount of unnecessary commenting to support its parsing algorithms. From our own testing, this commenting resulted in the failure in compilation of our previously functioning codebase. For full application features, a payment is also required.

ArgoUML is built on the same basis as eUML2, looking to provide the generation of UML class diagrams. Its usage was clear and straightforward when generating a diagram for a class. The diagram created also had clearly defined models in line with UML standards which is an important ideal to keep. Ideally, however, to understand a full system this should be applicable at a package level which was not the case with ArgoUML. It also featured heavy text based tags implemented within the diagram. These often obstructed the view, making it difficult to analyse the overall structure of the diagram. ArgoUML is a free, open-source project which should prove useful in utilising its benefits in our own project.

Class Visualiser, another reverse engineering tool, provided various great features for code comprehension. The generation of UML notated class diagrams was very simple, and each section of the model could be selected for further breakdown. This should be a key function in any reverse engineering tool as it allows details to be encapsulated, keeping the high-level diagram in a readable state. A major issue with Class Visualiser is a diagram can only be generated from one class at a time and hence will only show one degree of connection, rather than the system as a whole.

UMLet[11] is a very simply designed tool, with both a standalone and Eclipse plugin format. It is almost a polar opposite of JArchitect, providing a bare bones framework for implementing the core features of UML. We have experience in the use of UMLet through our Software Engineering courses, and it is a great introductory tool into the design of software architecture, providing the necessary tools to create your own software model. UMLet lacks some of the other crucial aspects found in other reverse engineering tools, with the main one being the automatic generation of design models based on existing code. This is vital when developing a system based on UML as it should primarily be used as a way of code comprehension. Instead, UMLet has been developed primarily as a design tool, allowing users to manually generate their own diagrams.

III. Design and Implementation

A. Design Decisions

Research has shown that UML was the most commonly used design model, despite its usage measuring 7.6% against 76.2% for no model at all. Furthermore, while Java is one of the most widely used languages (49%) along with C# (56%) and C++ (45%), only 7.1% of Java programmers use design models[12]. As mentioned earlier, those results can be attributed to the lack of training as well as the obstacles posed by complicated tools.

Our initial research on code comprehension has revealed the limitations of current systems, and led to the identification of key objectives of the proposed tool. We are planning to develop an application that will provide code comprehension features for an existing Java artefact with a focus on simplicity. The proposed reverse engineering tool should ideally present a majority of the information with diagrams and should make use of the UML framework. Along with these features, the tool should be developed with an open-source approach. A decision was made to focus on extending the already implemented software, UMLet [13].

UMLet fits the criteria for the project well. It is developed using Java, and targeted at the comprehension of Java systems. UMLet is completely open source and on a freely available plan, therefore, its functionalities are much easy to be extended. It also exists as both a standalone client, and an

Eclipse plugin which provides a lot of flexibility for extension, development and ease of use. UMLet is already used for educational reasons, gaining experience with it ourselves through a software engineering course. This is because of its simplistic design, while still offering the capabilities to create high-level UML structures to explain code structure. This makes UMLet ideal for an introductory code comprehension tool.

While UMLet has a lot of base features for code comprehension, it is all based around the aspect of designing a system, not understanding one. In doing so, it relies solely on the manual development of diagrams by users. We aim to modify UMLet into a reverse engineering tool which can be used for increasing the users' code comprehension of their input source code. Utilising UMLet's source code of rich UML design, we incorporate the automatic generation of UML features from existing Java systems, rather than the current user-driven construction. Also to consider is the addition of other code comprehension strategies such as the control flow graph or varying code metrics. These additional strategies will be useful but it is important to maintain the simplicity of the software as to not overwhelm potential users.

The extension to UMLet was done using an agile methodology. This is due to extending software being an iterative process, as multiple features being developed leads to an incremental development. It will also allow features to be revisited and improved after new constraints or solutions have been discovered during further implementation. The tool was extended for both the standalone product and the Eclipse plugin version of UMLet as the code used was very similar. The intended design is to have the core UML features be automatically generated from existing source code, using UMLets existing UML diagram drawing features. The diagram representations in UMLets currently use the '.uxf' format. As UML can be broken down into various parts, each feature will be focussed on independently. The initial focus is on the generation of Class Diagrams, being the best understood sector of UML. We will also add functionalities to increase the simplicity of the diagram interface and establish a link to the source code being analysed. Code metrics are also in scope for this project, and will be further explored and implemented following the completion of the automatic generation of the class diagram. This will be followed by the implementation of a behavioural structure, most likely a Sequence or Activity Diagram with evidence showing this combined with Class Diagrams improves code comprehension [14].

B. Key Features Implemented

The following gives a brief summary of the key features implemented in the prototype.

Automatic Generation – UMLet does currently offer a very primitive form of class diagram generation, but requires heavy modification. The current system will only generate an image of each class, with its methods and fields, then lay them out next to each other without any of the defining dependency features that UML offers. Upon breaking down the source code it was discovered that the file handling system is already in place and can be reused for updating the model generation. In order to have a fully functioning class diagram generator, there are two key phases, i.e., further parsing of files is required to locate the dependencies between classes and then structuring them into the diagram.

We have decided to focus specifically on inheritance and interface relationships with regards to the class diagram generated. This is because they are the essential components required for a user to get an overview of the structure of the system being analysed, without involving the more complicated intricacies involved in UML. These relationships were found by utilising the Java source code, and more specifically the *Javaparser* library. We utilised the *CompilationUnit* class, which represents the top level node of the Abstract Syntax Tree that is used to represent the contents within the Java source file being parsed. This allowed us to scan the file for its *TypeDeclaration* nodes, and in particular, the *TypeOrInterfaceDeclaration* nodes. From these, we could establish the list of interfaces that were implemented by this class, along with its inheritance structure.

We developed our structure algorithm for diagram generation , using the concept of inheritance layers and clusters. Each diagram element (which represents a class) was first sorted into an inheritance layer, with the top layer containing all classes with no superclass. The classes which extended those classes in the top layer, were then placed on the layer below. This was repeated until all elements had been assigned a layer. Elements were then sorted into clusters on a layer by layer basis, starting from the bottom layer. A cluster contains all classes which share the same direct superclass, on the layer above. Grouping these elements together allowed them to be evenly separated to ensure a readable diagram could be generated. These separated positions were stored as relative to their parent element, and this process was continued for all elements in the diagram. As the top inheritance layer has no shared superclass, an invisible node was generated at position (0,0), which was utilised as a mock superclass to determine the top layer elements relative positions. After all elements had a relative position generated, they were looped through and drawn in a breadth first approach. This overall algorithm allowed us to generate a clean diagram with no overlapping elements, and minimal line collisions, which included all inheritance and interface relations within the source files analysed. Figure 1 illustrates the UML class digram generated by our tool directly from Java source code with the dependency relationships identified among the classes.

Diagram Customisation – We felt that while adding these relationships helped to clarify users on the structure of the code, adding a more customisable experience would simplify the understanding of the diagram. To do this, we have implemented a core feature of showing and hiding elements. This can be accessed by right clicking any element, and using the drop down menu to select show or hide. This provides quick and easy access to the ability to show or hide a combination of either fields, methods or the responsibilities of each individual class. This allows users to hide information that is not currently

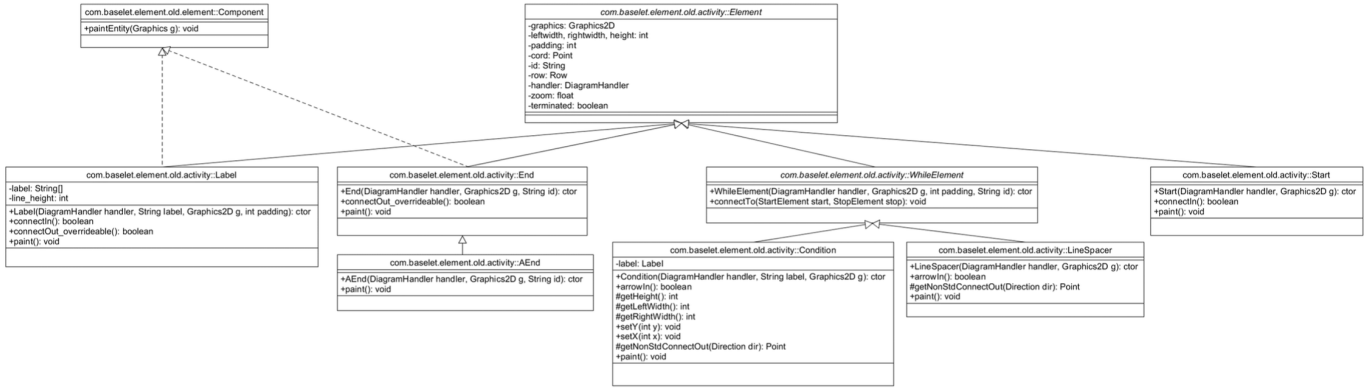

Fig. 1. UML Class Diagram Generated by the Tool

required, at runtime, making it easier to comprehend the information in the diagram which is of greatest importance. This feature also works on multiple elements simultaneously to be rapidly adjusted to user's needs. For example, if a user is studying the overall structure of the codebase, rather than individual classes, they could select all elements and hide all of their individual fields, methods and responsibilities. This will hide all of the unwanted text in the diagram allowing relationships to be easily identified and understood.

Class Information Panel – While hiding certain information can be useful, we also understand that at times it is important to gain an in-depth perspective on individual classes. In order to cater to this aspect of code comprehension, we have developed an information panel to accompany the diagram being generated. This information panel sits to the side of the diagram, to prevent obfuscation of the diagram elements. When an element is selected, the panel will be updated with two core sets of information. Figure 2 illustrates the class information panel generated by our tool from Java source code.

```
/**
* This class consists exclusively of static methods that operate on or return
* collections.  It contains polymorphic algorithms that operate on
* collections, "wrappers", which return a new collection backed by a
* specified collection, and a few other odds and ends.
*
* @author  Elliot Varoy
* @see     Map
* @since   1.2
*/
```

```
Class Information
Dog
Number of Fields: 1
Number of Methods: 3
Number of Lines: 31
Javadoc:
This class consists exclusively of static methods that operate on or return
collections.  It contains polymorphic algorithms that operate on
collections, "wrappers", which return a new collection backed by a
specified collection, and a few other odds and ends.
@author  Elliot Varoy
@see     Map
@since   1.2
```

Fig. 2. Class Information Panel

The first of these is a representation of the Javadoc within the class. This Javadoc will give a brief explanation of the class and its uses, while providing more information than simply a name as is often offered with UML diagrams. This allows for easy access viewing of the class, without having to locate and open the source code for that particular file. The second set shown within the information panel are code metrics derived from the element selected. These code metrics represent the importance of the class, and how it may be used within the entire structure of the system. This has been developed in an extremely extensible way, with future metrics having the ability to be added in a modular format. All that is required, is for future useful metrics to be identified and their relative information collected from the sources files to be easily added to the current list of code metrics. Currently, we have implemented counts for fields, methods and lines. These 3 categories allow the user to garner a quick overview of each class' importance within the code structure. If they are high in number, then a majority of the logic is stored within this class, and it is highly likely that this class is performing vital operations within the codebase and would be a central element to consider when developing an understanding of the system.

Source File Navigation – When using UML, it is important to recognise that the diagram created is a direct representation of the code being analysed. They are not two separate entities, but are instead defined by the links between them. After considering this, we decided to establish a direct link from the diagram to the source files to ensure that users could easily alternate between the two to enhance their overall comprehension. This was done by adding a source file navigation feature which can be accessed by right clicking any element in the diagram, and selecting the open source file button. When using the standalone client, this will open the source code for that class in the operating systems default text editor to alleviate any user confusion with custom formats. If the user is utilising our eclipse plugin version, the source code will open within the eclipse editor in a new editing tab. Both of these scenarios have the code opened in a panel separate to the diagram. This is to maintain the clarity of the diagram, while still allowing users to easily access the more in-depth information in the source code.

C. Technical Details

The project was implemented using agile software development processes, more specifically, a scrum-like process with a feature backlog and varying sprint lengths. Using scrum ensured a "shippable" tool once each feature had been implemented. This is to ensure that by the end of the project there exists a tool which works as opposed to a tool which has had a great deal of time spent on it but does not function. Scrum also made it easy for the project to evolve as the implementation progressed. When working on an existing code base, one of the biggest risk factors was being able to effectively understand and modify it. Although this was a great technical challenge initially, as the project progressed and more time was spent modifying UMLet, this soon sorted itself out. A great deal of the technical choices for the implementation of the tool were decided by default due to the extension of UMLet. The additional features were architected in the way which would make them fit best in to the existing UMLet codebase.

Arranging the class elements in to a structured UML class diagram was by far the biggest technical challenge of the project. A considerable amount of time and refinement went in to getting the algorithm to work as expected. The most well respected algorithm for performing such a task is called the Sugiyama algorithm. This is a NP-hard problem and although very effective, implementing such an algorithm was out of scope for this project. The algorithm that was implemented relies on layering and clustering, a slightly more crude approach than Sugiyama but effective nevertheless. Before it was possible to start arranging the class elements in the diagram, the relationships between the classes must be deduced. To do this the Java compilation unit class was used to observe the generated abstract syntax tree from the source files. It is then possible to query a compilation unit object for what that class extends or implements.

Now the algorithm can sort all the class elements in to inheritance layers, using the relationships deduced in the last step. If a class has no superclass it is on the top layer, classes extending a top level class are on the layer below and so on. The layering is then written to a map data structure. The map is why the same input source files can produce different diagrams, a map is unsorted and will be iterated across in different orders. The algorithm now starts at the bottom level of the diagram (currently stored in the map) and arranges the classes in to clusters. Classes which extended the same parent class will be clustered together. The classes in the cluster will now be assigned positions relative to their parent. This process is repeated for all of the layers of the diagram, working up. Once the algorithm reaches the top layer, where the classes do not have a parent, the classes are formed in to a single cluster and given an invisible parent element at position (0,0). Finally the algorithm will loop through all the layers, working from the top down and draw every class relative to its parent using the position data which was stored previously.

Extracting the code metrics from the source files again required the use of the Java compilation unit. The compilation unit object which is created for each class was required to be exposed to the rest of the application such that the newly created Class Information Panel could have access to it. Any information now stored in the compilation unit object can be used to generate code metrics and display them to the user. Reading the Javadoc was less elegant and required reading in the source file and performing parsing to scrape the Javadoc from the start of the file. This process is done when each class element is accessed, not on every file. This stops the initial generation time from being slowed, especially if a large number of files are being imported.

IV. Evaluation

In order to justify our additions to UMLet, and to assess whether we have developed a code comprehension tool that meets our initial goals, we conducted a series of quantitative and qualitative analyses. Firstly, through qualitative analysis by comparing the solution to the tools looked and in the research phase of the project and evaluating its strengths and weaknesses against these tools. The second method of evaluation would be through the use of expert reviews where experts are given a set of open questions to answer.

A. Tool Comparison

Our first set of evaluations was done through a tool comparison, similar to our initial research phase. The first analysis conducted was on the generation speed of our diagrams after the features had been added, compared the generation speed of the original UMLet. This was performed with both small scale ($<$15 source files) and large scale projects ($>$30 source files). It was found that our additional features slowed the generation of diagrams by a negligible amount. This was a very positive outcome for this project, as the structuring algorithm and development of relationships within the diagram has potential for dramatically increasing the generation time if developed in an inefficient manner.

We performed qualitative analysis by comparing the clarity of our diagram with the previous tools studied. From this, we found that our tool provided well-structured diagrams for small scale projects, which we were specifically targeting. These diagrams also removed a lot of the unnecessary information that was found in other tools, for example the tags found when using ArgoUML which cluttered the diagram and hid vital relationship information. We also found that the reduction in cross class dependencies in the form of aggregation and dependency lines made it easier to grasp an overall outline of the structure in a much shorter time period. This also provided benefits as less knowledge of UML was required to comprehend the code being analysed. We also found that our information panel was useful when trying to understand the importance of individual classes. This was presented in a much more simple form than other competing tools, namely JArchitect. JArchitect has a comprehensive code metric analysis function, but its barrier to learning is extremely high. We found our code metrics simper to understand, while still

providing the core functionality of recognising the importance of the class with respect to the overall structure of the system.

B. Expert Reviews

In addition, we conducted expert reviews on our tool to validate some of the findings we had and to identify any potential weaknesses we may have not accounted for. The experts consists of software engineering year 4 students and professionals from the IT industry. The main being a consensus from all experts that the diagrams generated by our tool excelled in simplicity. Because of this, they said it was an amazing tool for quickly familiarising with the existing code, allowing them to build their code comprehension of the system much faster than with previous tools. There were some issues outlined with the customisation function, with respect to hiding certain areas of information. We were able to rectify this in our final build to provide even more clarity within the diagram. The users found our diagram easy to use, with no experts requiring extra assistance for setup or aid in understanding what was presented. They also found that the amount of UML provided was enough to understand the majority of the structure. However there were some participants who found that there were certain areas of the diagram which required more information for them to gain a full understanding of the system. This has been recognised, however is an intended nature of the system in order to cater for users who do not require these complexities that other tools already provide.

V. Conclusion

Reverse Engineering is an important step towards code comprehension when working with existing software systems. Research into differing code comprehension strategies has found that for high-level structural understanding, the Unified Modelling Language is a very important step for visualising this understanding. Tools already exist for both the generation of UML models, and complimenting code metrics, however many of these are complex and aimed at a company model, rather than beginner or individual developers.

Upon research into existing tools for code comprehension an extension of the UMLet software was chosen. The extension was to fill the void of Java applications that currently only offer complicated high-level structural understanding tools. We added four core features to the UMLet platform which includes automatic generation with relationships, runtime customisation of diagrams, an information panel detailing specific classes and a direct link between the diagram and its source. Overall, we have developed a tool that fits the niche our research has discovered and it has been well received by experts looking to utilise reverse engineering to increase their code comprehension of existing software systems.

While the initial stage of the project has been a success, there remain specific areas for improvement. Firstly, we would like to incorporate the use of the standard UML format '.xmi' by providing export functionality from our generated diagrams. This will provide more flexibility to users, allowing them to utilise the diagrams within more complex systems after gaining an initial understanding with our software. Secondly, further improvements could be made by adding a list of code metrics to the information panel. Thirdly, we would also like to improve our structural generation algorithm. Utilising more of the concepts outlined within the Sugiyama algorithm could help to improve the readability and clarity of the diagrams, offering better code comprehension results. Finally, we plant to further extend the model generation to accommodate the behavioural diagrams such as the sequence and activity.

References

[1] E. J. Chikofsky and J. H. Cross II, "Reverse engineering and design recovery: A taxonomy," *IEEE Softw.*, vol. 7, no. 1, pp. 13–17, Jan. 1990. [Online]. Available: http://dx.doi.org/10.1109/52.43044

[2] H. Zhu, J. Sun, J. S. Dong, and S.-W. Lin, "From Verified Model to Executable Program: the PAT Approach," *Innovations in Systems and Software Engineering*, pp. 1–26, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11334-015-0269-z

[3] C. Eckert, B. Cham, J. Sun, and G. Dobbie, "From design to code: An educational approach," in *Proceedings of the 28th International Conference on Software Engineering & Knowledge Engineering (SEKE 2016)*, July 2016, pp. 443–448. [Online]. Available: http://doi.acm.org/10.18293/SEKE2016-007

[4] S. Wong, J. Sun, I. Warren, and J. Sun, "A scalable approach to multi-style architectural modeling and verification," in *13th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2008)*, March 2008, pp. 25–34.

[5] C. Lange, M. Chaudron, and J. Muskens, "In practice: Uml software architecture and design description," *IEEE Softw.*, vol. 23, no. 2, pp. 40–46, Mar. 2006. [Online]. Available: http://dx.doi.org/10.1109/MS.2006.50

[6] G. Booch, J. Rumbaugh, and I. Jacobson, *Unified Modeling Language User Guide, The (2Nd Edition) (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 2005.

[7] F. E. Allen, "Control flow analysis," *SIGPLAN Not.*, vol. 5, no. 7, pp. 1–19, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/390013.808479

[8] T. J. McCabe, "A complexity measure," in *Proceedings of the 2Nd International Conference on Software Engineering*, ser. ICSE '76. Los Alamitos, CA, USA: IEEE Computer Society Press, 1976, pp. 407–. [Online]. Available: http://dl.acm.org/citation.cfm?id=800253.807712

[9] R. Lincke, J. Lundberg, and W. Löwe, "Comparing software metrics tools," in *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ser. ISSTA '08. New York, NY, USA: ACM, 2008, pp. 131–142. [Online]. Available: http://doi.acm.org/10.1145/1390630.1390648

[10] Y. Jiang, B. Cuki, T. Menzies, and N. Bartlow, "Comparing design and code metrics for software quality prediction," in *Proceedings of the 4th International Workshop on Predictor Models in Software Engineering*, ser. PROMISE '08. New York, NY, USA: ACM, 2008, pp. 11–18. [Online]. Available: http://doi.acm.org/10.1145/1370788.1370793

[11] M. Auer, T. Tschurtschenthaler, and S. Biffl, "A flyweight uml modelling tool for software development in heterogeneous environments," in *Proceedings of the 29th Conference on EUROMICRO*, ser. EUROMICRO '03. Washington, DC, USA: IEEE Computer Society, 2003, pp. 267–. [Online]. Available: http://dl.acm.org/citation.cfm?id=942796.943259

[12] T. Gorschek, E. Tempero, and L. Angelis, "On the use of software design models in software development practice: An empirical investigation," *Journal of Systems and Software*, vol. 95, pp. 176 – 193, 2014. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0164121214001022

[13] M. Auer, L. Meyer, and S. Biffl, "Explorative uml modeling - comparing the usability of uml tools," in *Proceedings of the Ninth International Conference on Enterprise Information Systems*, 2007, pp. 466–473.

[14] G. Scanniello, C. Gravino, and G. Tortora, "An early investigation on the contribution of class and sequence diagrams in source code comprehension," in *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, March 2013, pp. 367–370.