A USER INTERFACE FOR THE INTERACTIVE REFINEMENT OF AN OBJECT GRAPH

by

WESLEY TRESCOTT

THESIS

Submitted to the Computer Science Department

of Wayne State University,

Detroit, Michigan

in partial fulfillment of the requirements of the

HONORS THESIS

2015

MAJOR: COMPUTER SCIENCE

Approved By:

Advisor	Date

DEDICATION

To my family.

ACKNOWLEDGEMENTS

I would first like to thank Professor Abi-Antoun for advising me during this research project and providing well defined, interesting, and doable tasks, and also for supporting me with assistance on challenging problems and feedback on my work.

I also would like to thank the other students in the SEVERE lab, Ebrahim Khalaj, Anamul Haque, and Sumukhi Chandrashekar, for supporting my project with help on technological and conceptual questions, and for livening up the lab!

TABLE OF CONTENTS

Dedication ii

Acknowledgements iii

List of Tables v

List of Figures v

Chapter 1: Introduction 1

Chapter 2: Background 3

2.1 Previous Work 3

2.2 ArchDoc 4

2.3 Thesis Statement 7

Chapter 3: Requirements and Implementation 8

3.1 Tree View Project 8

3.1.1 Tree View Requirements 8

3.1.2 Tree View Implementation 9

3.2 Subproject: Preliminary OOGRE Evaluation 11

3.2.1 Evaluation Method 11

3.2.2 Evaluation Outcomes 16

3.3 Graph View Project 20

3.3.1 Graph View Requirements 20

3.3.2 Graph View Implementation 22

Chapter 4: Discussion and Conclusion 26

4.1 Discussion 26

4.2 Conclusion 26

References 28

Abstract 29

Autobiographical Statement 30

LIST OF TABLES

Table. 3.1 Results of evaluation 16

LIST OF FIGURES

Figure. 2.1 Screenshot of the tool. The tree view is shown on the left and the graph view is shown on the right. 6

Figure. 3.1 OOGRE tree view with refinement table 10

Figure. 3.2 ScrabbleModel high-level system view. Dotted lines represent pointer relationships and solid lines represent composition, while the triangles indicate ownership. 13

Figure. 3.3 Evolution of intended architecture: on the top is the initial architecture and below is the final architecture. 17

Figure. 3.4 OOGRE graph view before and after 23

Figure. 3.5 Dropping the newTop:Link object into the XStack::PD domain . 24

Figure. 3.6 OOG after dropping newTop:Link into the XStack::PD domain . 25

Chapter 1: Introduction

As software developers, it is often very difficult or impossible to quickly gain an understanding of the workings of a software system, even when it is a system developed oneself. This is a major problem in the software engineering process, as it slows down development by adding enormously to time requirements for software maintenance tasks. This is demonstrated by a study that found that fifty percent of the time devoted to maintenance is spent on code comprehension rather than actual implementation [3], and further research shows that over sixty percent of all business expenditures on computing go to software maintenance [2]. As such, reducing time spent on software comprehension could have major cost saving potential, making it an intriguing research area. Various methods of increasing software comprehension have been proposed, and many of them have enjoyed widespread usage in industry and research, including UML class diagrams and system sequence diagrams. Another tool addressing this need is the research software OOGRE, the **O**wnership **O**bject **G**raph **R**efinement **E**ngine, which is the topic of this thesis. OOGRE assists in code comprehension by giving a visual depiction of what a software system will look like at runtime without actually running the program. This means that OOGRE is a static analysis tool presenting the user a sound abstraction of the system's runtime architecture, in which every possibly created runtime object is represented by an abstract object. I discuss this analysis technique, and contrast it with dynamic analysis, in greater depth shortly. OOGRE supports manipulation of the abstraction of the system's runtime structure, allowing users to modify the abstraction and change it interactively to fit their mental image of the architecture or to model their design intent.

Given the large amount of time spent in program comprehension during software maintenance, it is important to have a sound, reliable comprehension tool like

OOGRE to aid developers. This can be accomplished using OOGRE's interactive user interface to refine representations of the software system to aid in developers' understanding of it. This research contributes interactive graph and tree views to the tool in support of this user manipulation functionality. With these considerations in mind, I give a brief background to the development of OOGRE, discuss contributions to the tool's development and evaluation, and consider the outcomes of the project.

Chapter 2: Background

2.1 Previous Work

To begin, we briefly consider previous work that lead to the creation of OOGRE. In the first place, it is helpful to define a few of the concepts mentioned above; specifically, the two sort of software analysis, static and dynamic, were made reference to. Here, dynamic analysis refers to examining software by running it, while static software analyses examine code without running the program. This is useful, because dynamic analyses often run the risk of not accounting for important runtime objects that arise during other executions, presenting a gap in understanding of the system's functionality.

This work builds on a previous approach which, recognizing the advantages of static analysis noted above, established the use of ownership domain code annotations to drive the extraction of a hierarchical abstract object graph that supports the comprehension of software systems. The annotations group objects into ownership domains. Domains help to give the abstraction hierarchy so that abstract objects can be nested inside other abstract objects. This way, a complex system can be modeled more simply, without having countless objects and countless relations between them depicted illegibly on a single graph. When mentioning abstract objects, it is useful to explain that OOGRE does not represent each object possibly created in the code by an individual node. Rather, it abstracts one or more instances of an object of the same type into one abstract object, making the number of objects represented in OOGRE much more manageable. Additionally, each abstract object contains domains, as mentioned. There are two types of domains, public domains and private domains. Public domains are used to hold objects logically contained in the parent abstract object. Private domains imply that the object is strictly encapsulated inside the parent object and never accessed from outside the parent. A graph containing ab-

stract objects organized into a hierarchy with public and private domains, and edges representing pointer relationships between objects, is referred to as an Ownership Object Graph (OOG) [1].

An additional inspiration for this project is the Ownership Object Graph Interactive Editor (OOGIE), which first proposed an interactive interface for manipulating a default OOG model. OOGIE forms the basis for this work by providing a first prototype of an interactive graph interface and a set of requirements for their performance and functionality, which created a baseline for the further development of another tool called ArchDoc, which we discuss next.

2.2 ArchDoc

ArchDoc is meant to assist developers in comprehending object-oriented code; it therefore supports one of the most popular object-oriented languages, Java. Additionally, the tool is a plugin to the Eclipse integrated development environment (IDE), which is one of the most popular Java-supporting IDEs. An additional benefit to using this platform is that Eclipse is open source, allowing for easy plug-in development. Within Eclipse, different perspectives show users different windows, or, as they are called in Eclipse, views, dealing with different aspects of the program. For example, there are perspectives for general Java development, for debugging, for exploring source control repositories, and for many other purposes. The ArchDoc Eclipse plugin is another perspective, and has several views, two of which are of particular relevance to this evaluation and which are discussed momentarily.

ArchDoc harnesses the ownership domains discussed above by displaying an OOG and an abstract object tree view extracted from the code annotations. Neither the tree nor the graph in ArchDoc is interactive, as both are merely static renderings based on code annotations. To update the OOG system model, the user must change the code annotations and rerun a backend analysis to extract a new tree and graph. But

this is problematic: updating annotations is a time-consuming and tedious process involving slight code modifications followed by verification using an annotation type-checker. As a primer for my development work for this project, I used this process to annotate a simple software system and experienced firsthand the formidability of this process. Clearly, if ownership annotations were to be useful and time-effective in comprehension tasks, another method was necessary and this is where OOGRE came into play.

OOGRE works by adding the aforementioned annotations to Java source code automatically. A backend analysis then creates a default model of the architecture by traversing the system's abstract syntax tree using the Crystal Static Analysis Framework and grouping objects and domains based on the annotations. The model of the system architecture is then presented to the user in two formats: an abstract object tree and an OOG. These two views correspond to those in ArchDoc mentioned earlier. The OOG and the tree both represent the same hierarchical nested abstract objects mentioned above, but in different format. Each node represents either an abstract object in the system or a domain, with domains being nested inside objects. These views are displayed in Fig. 2.1.

As the tool is designed to assist developers in code comprehension and software analysis, the user interacts with the default OOG produced to increase understanding of the code and reflect his design intent. The user can expand and collapse nodes to see which objects are located inside which domains, and then manipulate the rendering of the OOG. This is supported through the drag and drop functionality of the abstract object tree. When the user wants to move an object into a different domain, he simply has to select and drag the node, dropping it on the target domain. This same functionality can also be done in the graph view, though all features of the graph have not yet been implemented. After the user makes a change to the tree or graph, called a refinement, he runs the OOGRE Crystal analysis, which validates

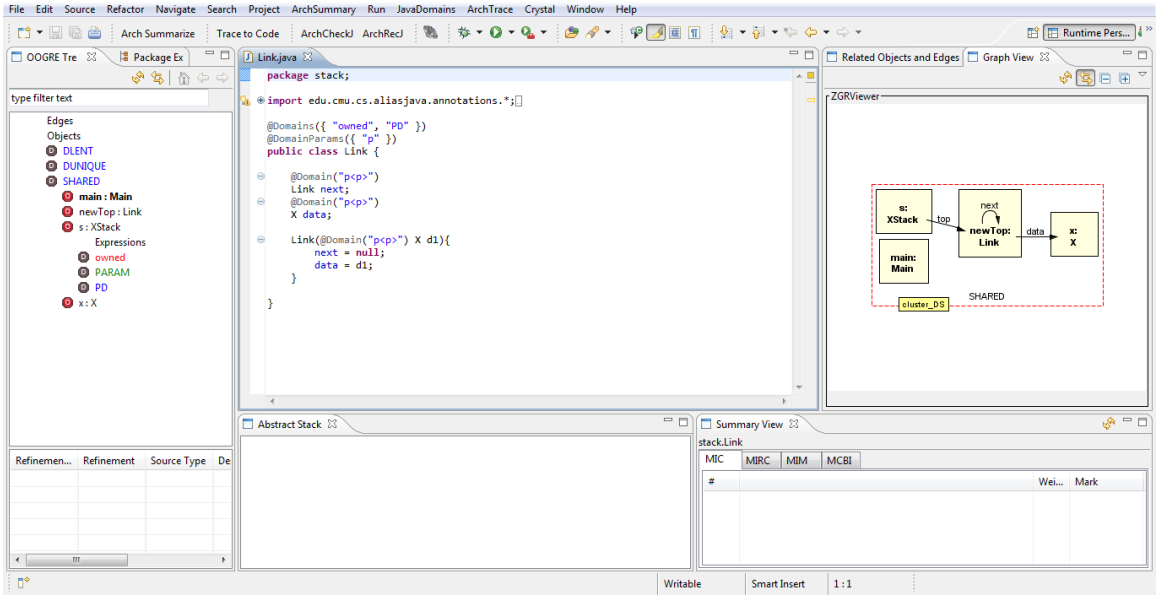

Figure 2.1: Screenshot of the tool. The tree view is shown on the left and the graph view is shown on the right.

the refinement, and, if supported, updates the code ownership domain annotations; another analysis then automatically re-extracts and displays the new OOG model. It is this functionality that gives OOGRE its name as a refinement engine for OOGs. OOGRE thus speeds up the analysis process considerably by removing the requirement of manually annotating code. While the OOGRE analysis and the ArchDoc views were completed as part of prior work, the primary contribution of this research is in replacing ArchDoc's static tree and graph views with the interactive views of OOGRE.

As discussed above, in adding support for drag and drop operations in the tree and graph views, we may increase usability of ownership domains by allowing the user to bypass the tedious and time-consuming process of manually changing the code annotations when doing refinements. After evaluating the tree view functionalities in OOGRE, I am able to posit the following thesis statement.

2.3 Thesis Statement

Developers interactively refine a hierarchical object graph by direct manipulation of the ownership tree or of the graph.

Chapter 3: Requirements and Implementation

To arrive at the conclusions summarized in the above thesis statement it was necessary to define the functional requirements of OOGRE's interactive tree and graph views and implement them, before we could verify their usability. I also conducted a preliminary evaluation of OOGRE to better understand the limitations of the previous user interface. These steps are discussed in the following subsections.

3.1 Tree View Project

In creating a usable interface for making refinements with OOGRE, I began by improving the existing static tree view. The tree view displays a hierarchical rendering of the OOG, with each node representing an abstract object and containing domains as children. Users can now select an object node in the tree, drag it to a target domain of another object and drop it there to create a refinement. Since earlier prototypes of an interactive editor already included a functional tree view, it was a good starting point to bring the static ArchDoc tree up to the standards of previous work and to gain a better idea of the type of interaction targeted for the graph view.

3.1.1 Tree View Requirements

For the purposes of the tree view implementation, it was necessary to adhere to a few functional requirements to prioritize work items in the creation of a tool that fulfills its goal of supporting interactive refinements to an OOG. These requirements are:

- The tree view must enable drag-and-drop refinement operations.
- The tree view must enable correct generation of refinement objects based on refinement source and target and pass them to the refinement engine.

- The tree view must invalidate incorrect manipulation of the OOG, so that users cannot create refinements that are unsupported or in violation of domain logic.
- The tree view must have an additional integrated refinement window to show refinement information and status.
- The refinement window must update the status of each refinement after the analysis is rerun.
- The tree view must allow users to create a system abstraction that makes sense to them by providing customization options in a context menu.

3.1.2 Tree View Implementation

I then created an implementation based on the tree view functional requirements. Since the basic drag and drop mechanism had already been added to the tree view, my first contribution was the implementation of a correct refinement model object based on the type of refinement that was carried out. The refinement object is used by the OOGRE backend engine to update the code annotations, enabling an OOG model to be re-extracted. This task involved detecting the source and target of the refinement, and creating the refinement object based on whether an object was being pushed into a public or private domain. Next, to support the collection of refinement objects, I added a small window to the bottom of the tree view, called the refinement view. The refinement view is essentially a table containing an entry that holds information about each refinement carried out on the tree view. In the table there are column headings for target and source information, as well as an entry displaying the status of the refinement. The status indicates that the refinement is pending until the user reruns the analysis, at which point the table will indicate whether the refinement was successful or unsupported. Each time the user executes a drag-and-drop refinement

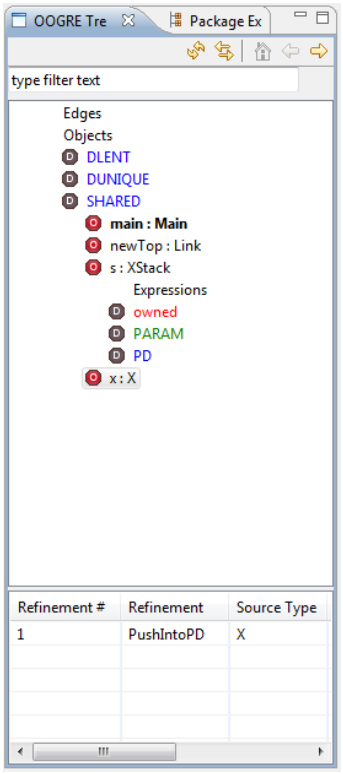

Figure 3.1: OOGRE tree view with refinement table

operation on the tree, another entry is added to the refinement view. Fig. 3.1 shows the refinement table at the bottom of the tree view.

At this point, the tree view allowed any node item to be dragged and dropped on any other node item. Thus it was pertinent to add logic that disallowed operations that were in violation of OOG conventions. This meant disabling the drag mechanism on domains, disallowing objects as drop targets, and requiring that the target domain belong to a different object than the one being dragged. In addition to this critical logic, I also added context menu support in an attempt to increase tool usability; this allows users the flexibility to generate a system abstraction that makes sense to them. Among the items in the context menu is an option allowing users to rename domains, and there are also entries to add public and private domains and to remove domains.

3.2 Subproject: Preliminary OOGRE Evaluation

After completing most of the major functionality for the OOGRE tree view, I conducted a preliminary evaluation to assess its usability. The outcomes of the evaluation also gave me better perspective for evaluating the results of my implementation tasks.

3.2.1 Evaluation Method

My evaluation of the OOGRE tree view used as its subject system ScrabbleModel, a 1327 LOC object-oriented Java program implementing a version of the well-known Scrabble board game, modified with the addition of special tiles that influence point scores and perform special effects. To support this functionality, the Scrabble implementation includes a user interface, which uses the large Swing library. However, due to the fact that OOGRE does not yet support annotations for library classes, this portion of the code was excluded from the evaluation, which only included the core game logic, referred to as ScrabbleModel to distinguish it from the game as a whole (including the user interface). I recently wrote this program for a software design class, so I was already familiar with the system. Prior to the evaluation, OOGRE had only been tested on small Java programs, and evaluating its performance on a larger system provided the opportunity to assess its functionality in a more likely use case and to take note of various aspects of the user interface that are not conducive to ease of use. In conducting this evaluation, the quantitative data I collected was whether the tool produced the correct annotations based on the refinements I made to the abstract object tree. This measured whether the tool can be used to achieve my design intent for the system. The qualitative data I collected were measures of the usability of the tool, such as ease of use of the drag and drop mechanism in the tree, or clarity of the tree context menu items.

Before beginning the evaluation, I had to make several changes to the Scrabble source code to enable compatibility with the then-existing constraints of the tool. Since OOGRE does not currently support adding annotations to library classes or generics, and the Scrabble code made extensive use of ArrayLists, Stacks, and HashSets, a workaround to this problem was to replicate these classes in my project with stubs, classes containing the same method signatures and having a member object but not supporting any data computations. This involved putting the stubs into a package inside the project named with the same name as the library package (java.util), so that the Eclipse compiler will find the local package on the classpath before the library one. Next, I traversed the code in Scrabble and replaced all references to library classes with the corresponding dummy class.

Another problem in the Scrabble subject system was its use of anonymous classes, objects created on the fly that are never given names. These anonymous classes were used mostly as arguments to a method and could be replaced by extracting a named local variable. The OOGRE tool currently does not handle the anonymous classes smoothly, as they appear in the abstract object tree with the generic name "NEWWWWWname." Beyond these code changes, there were no other customizations to the tool or subject system that were required to conduct the evaluation. So, after refactoring the Scrabble code to remove anonymous objects, I was ready to begin conducting refinements.

First it was necessary to consider what the target architecture of the system was. This would tell whether the refinement I was conducting made logical sense and should be supported by ArchDoc, or whether unsupported refinements I encountered were the result of incorrect operations. I began by envisioning the architecturally relevant objects in the system and their relationships. This lead to an understanding of the relationship between items from the traditional Scrabble game such as a Player, a Board, a LetterTile, a Dictionary, and a TileTray, and I viewed my target archi-

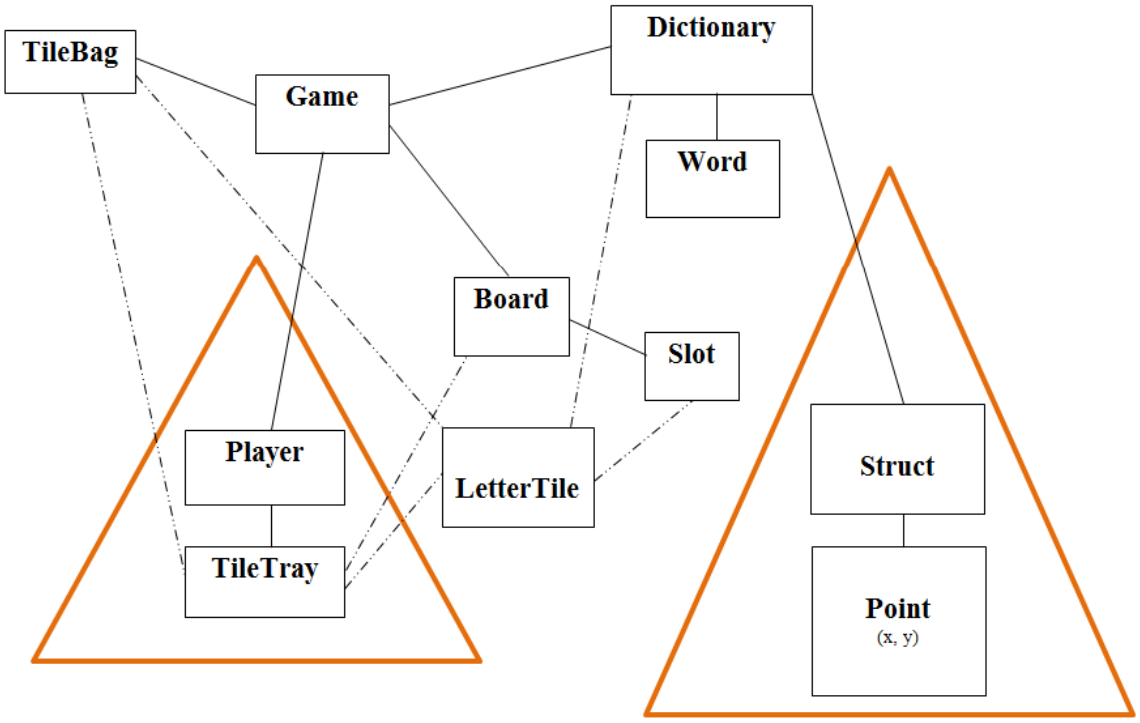

Figure 3.2: ScrabbleModel high-level system view. Dotted lines represent pointer relationships and solid lines represent composition, while the triangles indicate ownership.

tecture as corresponding as closely as possible to the relations between the different components of Scrabble in the physical game. After viewing the default OOG with its abstract objects, I came up with a rough diagrammatic high-level system view. This is shown in Fig. 3.2. I then accounted for all the abstract objects created in the default OOG and created a tree-like target architecture diagram for the Scrabble system, which enabled me to itemize the refinements I would conduct.

The bulk of my evaluation work was in performing drag-and-drop operations on the abstract object tree and rerunning the backend analysis to check whether my refinement was successful. I conducted all refinement operations in the tree view because at this point the graph view drag-and-drop functionality had not yet been integrated into the OOGRE. Each refinement was easily carried out by selecting an object in the tree and dragging it to a domain inside another object and dropping it there. After dropping the object, the refinement window updates with information

about the refinement performed, including the source object and target domain. Next, from the Crystal dropdown menu, I reran the OOGRE analysis and refresh the tree view. The refinement window contains a status field which notifies the user whether the refinement was successful, and, if the operation was successful, the source object is now displayed nested inside the target domain in the tree view. In summary, I followed these simple steps for each refinement on all three iterations of the evaluation:

1. Decide on target architecture
2. Create list of numbered refinements to implement target architecture
3. Carry out one refinement by dragging object into domain
4. Run backend OOGRE analysis and refresh tree and graph views
5. Verify status of refinement in refinement window of graph view
6. Repeat process for each refinement until target architecture is attained
7. Review results and restart process if necessary

In conducting the evaluation, I eventually completed three iterations of my refinements to the default OOG of Scrabble. Following the steps listed above, and starting with the high-level system view, I came up with a target architecture for the abstract objects of Scrabble. I then attempted to complete the refinements in the tree corresponding to this target architecture. I labeled each refinement with a number to keep track of which refinements were successful and which were not. The results of completing these refinements were that some succeeded as expected and others failed. I then discovered that some of the reason for the failure of a few of the refinements was due to a lack of understanding on my part of ownership domains, and also a lack of familiarity with some of the then-existing shortfalls of the tool. After the feedback to this set of refinements, I revised my target architecture and reattempted the

refinements. The results of this second iteration were that some of the refinements were still unsupported, and I discovered that this was also due to another gap in my understanding of ownership domains. Thus, I revised my target architecture a third time and reran the refinements once more. The results of each iteration are discussed in depth below.

An image of my first target architecture is given at the top of Fig. 3.3. I completed twenty-five refinements to the abstract object graph to implement this target architecture. After each refinement, I saved images of the graph at that point in the process, showing views with both all nodes closed and all nodes expanded. However, I had a conceptual misunderstanding of the meaning of the PARAM ownership domain, as I did not realize that it is used to bind contents to a different actual domain. Also, I was unaware that some of the objects I had originally believed belonged in the owned domain were not really strictly encapsulated. This resulted in attempted unsupported refinements. Also, I did not recognize other shortcomings of the early OOGRE prototype that still needed to be corrected, such as the requirements that all Push-Into-Owned refinements be completed before all Push-Into-PD refinements and that all refinements outside of the main system class be completed before refinements involving the main class[1]. I addressed these problems in my second iteration, and updated the target architecture accordingly. When completing the second iteration, I also discovered another conceptual feature of the tool that I had neglected in my first iteration, namely the SplitUp functionality. This is used when an abstract object merges several objects that belong in different domains, and it splits the abstract object into multiple abstract objects. The tool's trace to code functionality, which allows a user to trace from an abstract object to an object creation expression in the code, helped me to discover a few objects, such as objects of Point, that belonged in different domains. I tried to use SplitUp on them, some successfully and others

[1]Some of these limitations have been relaxed in the current version of the tool.

Table 3.1: Results of evaluation

Iteration #	Number of Refinements	Number Passing	# Failing	Pass Rate
1	25	2	23	8%
2	25	13	12	52%
3	25	16	9	64%

unsuccessfully. Finally, I discovered a final conceptual flaw in my design: data structures cannot encapsulate their data objects. Thus, I completed a final revision of the target architecture, and redid the refinements once more. My final architecture is shown at the bottom of Fig. 3.3. After this iteration, I could safely conclude that unsupported refinements were due to bugs in the tool rather than problems with my conceptual design.

3.2.2 Evaluation Outcomes

In discussing the results of the refinement operations, I only consider the results obtained on the last iteration of refinements, since the other iterations contained misunderstandings of conceptual topics and a lack of understanding of tool bugs that obscure the results of the evaluation. Nevertheless, it was still interesting to consider the statistics from all three iterations, presented in in Table 3.1, to make note of the significant improvement in the performance of the tool once it is used properly.

In the final iteration of the refinements on the Scrabble subject system, there were 16 true positives, or cases where OOGRE correctly executed a valid refinement to the OOG tree. Since the tool only produces results for refinements that were purposely conducted on the tree, there are no possible instances where OOGRE would report a true positive, an attempted legal refinement, that is not relevant to the system–all true positives in this case are legal refinements and relevant to the system. Likewise, in the final iteration, there were two false positives in which the tool reported that a refinement was unsupported when in fact it was a valid refinement. As discussed,

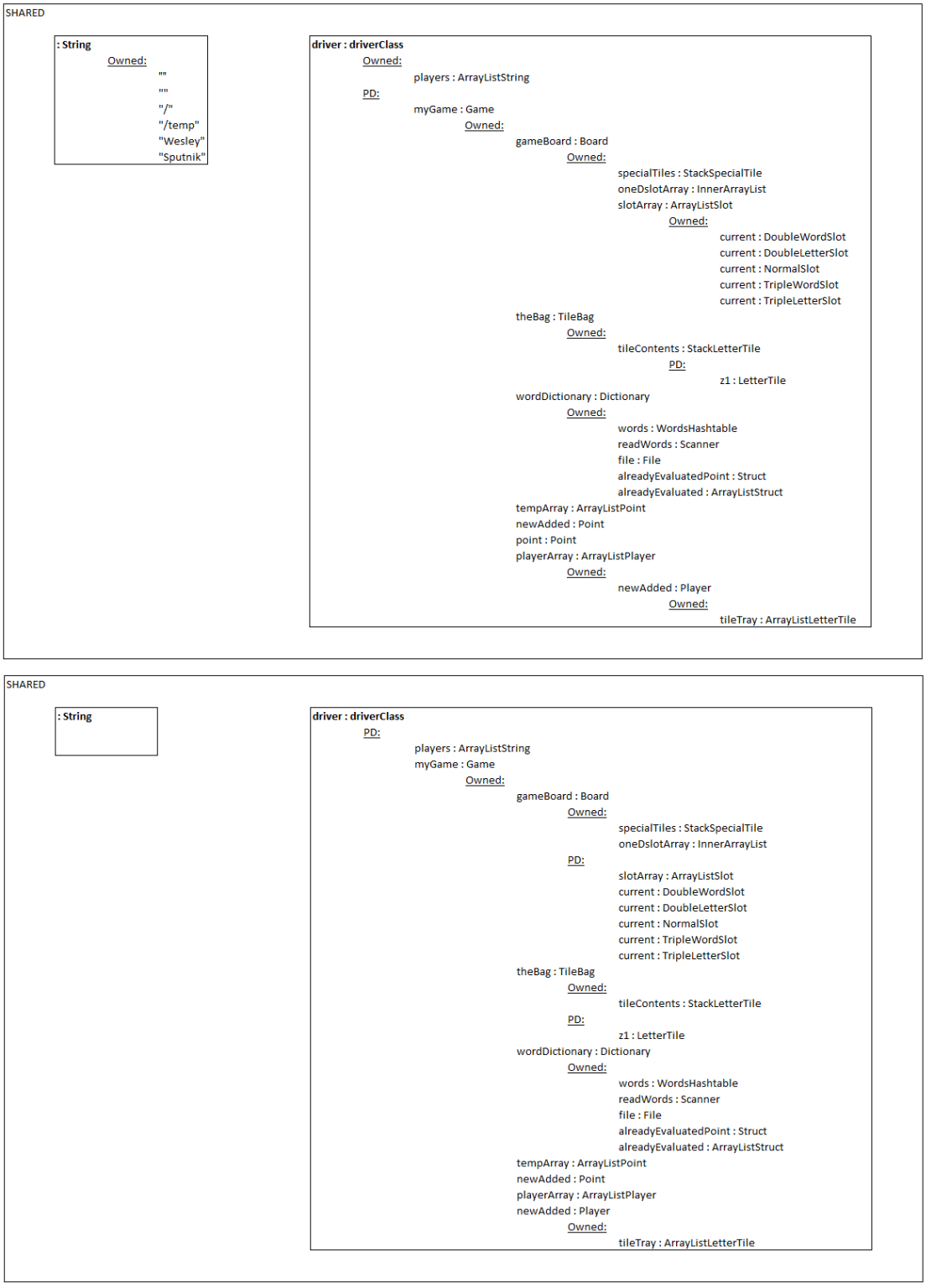

Figure 3.3: Evolution of intended architecture: on the top is the initial architecture and below is the final architecture.

when the tool user has a comprehensive understanding of ownership domains, it is much easier to use OOGRE to understand the software system. This implies that, for the tool to be more useful, there should be notifications, warnings, or some visual indication that a particular refinement is not allowed or unsupported. However, even with knowledge of ownership domains, only 64% of attempted refinements were successfully completed by the tool, indicating that there is significant progress left to be made, both to the backend engine and frontend user interface. However, in light of the fact that this evaluation was the first test of the tool on a realistic system, it is understandable that the pass rate is not satisfactorily high.

Nevertheless, the 64% pass rate was achieved only by following two tool limitations, that is, completing Push-Into-Owned refinements before Push-Into-PD refinements and completing all refinements outside of the main class before refinements inside the main class. These are major usability constraints, as they prevent the user from completing the refinements in the order that is most convenient or logical to him. Also, the drag and drop functionality of the tree was somewhat awkward due to the need to differentiate between similar looking node labels and the difficulty of scrolling down the tree to move an object near the top into a domain near the bottom. Furthermore, the amount of time it took to rerun the OOGRE analysis after every refinement could be problematic, as it caused the entire refinement process to take longer than a user would expect[2]. A final usability concern is the large context menu that appeared when a system contains many object creation expressions are abstracted into one abstract object. This was the case in the Scrabble game, in which there was one abstract object for each of the 100 object creation expressions in the game. The trace to code functionality in the context menu of the single abstract object displayed an enormous list of object creation expressions, which could be overwhelming to a user. However, such a result may be an indication of poor coding style

[2]The performance of OOGRE has been significantly improved since this early prototype.

rather than a tool issue, so this is not necessarily a problem that needs to be resolved in future tool fixes.

Usability issues aside, OOGRE also demonstrated several problems and erratic behavior issues that made it all but unusable in some situations. Perhaps the most noticeable problem was its treatment of different abstract objects with the same name but different type. For example, in Scrabble there were five abstract objects of different derived types of Slot, each named "current." When attempting to move one current object into the domain of another object, the tool treats all the current objects as one and moves all five together. This was definitely not intended behavior and is very problematic when attempting to implement a target architecture. Additionally, the tool had functionality where various objects were automatically pushed into a private domain of the main class without the user's knowledge or consent. Therefore, the automatic moving of objects without user input is a potential problem that may need to be resolved before the tool can be useful to developers trying to gain an understanding of a software system.

There were other, additional problems with the tool that make its use confusing or difficult, and these included display issues with the graph view, problems with SplitUp labeling, and problems with library types. In particular, objects with the same instance name in the same domain were incorrectly abstracted into one object in the graph[3]. Also, in the tree view, the different expressions under the SplitUp node did not correspond to the items in the trace to code context menu, making it difficult to know which item to perform a SplitUp operation on. Finally, there were problems with library types in the abstract object tree, since these did not contain any domain descendants to push other objects into, and there is also no SplitUp functionality available in cases of an abstract object of a library type. Each of these problems complicated or impeded the tool's use, and it was obvious that fixes to the tool would

[3]Some of these issues in the early prototype were eventually resolved.

be necessary before it could deliver on its intended usefulness.

Beyond the problems currently existing in the tool, OOGRE has great potential to improve the process of code comprehension for its users. In fact, one of the limitations of this evaluation was that I did not actively review the functionality of the graph view to verify for each object its correspondence to the abstract object tree. I also neglected to investigate the functionality of the other views, such as the abstract stack, in the aforementioned ArchDoc tool, and including these items in a future evaluation would give a more comprehensive analysis of the tool's usability. A final note in relation to the tool's usefulness would be to consider the learning curve involving ownership domains. As demonstrated in my evaluation, it took several iterations of refinements to fully understand the different requirements and standards for correct annotations. Thus, it might be helpful if future tool development included functionality to clarify correct and incorrect refinements to the user.

3.3 Graph View Project

After completing the OOGRE evaluation with the tree view, I embarked on my final and largest project, the new graphing interface that replaces ArchDoc's static OOG. As discussed, the purpose of the new graph is to allow users to interactively refine the rendering of the software system produced by the OOGRE engine, and this concept builds on the contributions of the Ownership Object Graph Interactive Editor (OOGIE).

3.3.1 Graph View Requirements

The following requirements describe the expected behavior capabilities of the graph tool and served as a guide for its implementation. Selitsky created the essence of the following requirements in her work on OOGIE [4].

Graph View Nonfunctional Requirements

- The graph view must be able to be ported into the ArchDoc tool.
- The graph view must have high ease of use so that users will not need to spend time learning to use the tool.
- The graph view must maintain diagram soundness, so that relationships and objects cannot be added or deleted directly.
- The graph view must visually distinguish between objects, public domains, and private domains in order to create distance between semantic ideas.

Graph View Functional Requirements

- The graph view must support iterative refinement through drag-and-drop operations on the graph. In particular, the tool must allow objects to be nested in domains and also removed from domains.
- The graph view must allow manipulation of the ownership hierarchy by allowing nodes at a higher hierarchical level to be moved down and nodes at a lower hierarchical level to be moved up.
- The graph view must allow users to collapse and expand the substructures of nodes in the nested hierarchy to show both more and less depth.
- The graph view must invalidate incorrect manipulations of the OOG so that the integrity of ownership domains is not violated.
- The graph view must enable correct generation of refinement objects based on refinement source and target and pass them to the refinement engine.

3.3.2 Graph View Implementation

The OOGRE graph implementation project began with an overhaul of an existing graphing tool[4], an Eclipse perspective built on a Java graphing framework called the Simple Hierarchical Multi-Perspective (SHriMP) view [5]. I chose this tool to develop because it has satisfactory documentation with readily available source code tightly integrated with Eclipse, meaning that though it did not support all of the functionalities we were interested in, it could readily be modified. These unsupported features had to do with the fact that the framework allowed nodes to be dragged, but there was no support for dropping one node into another, which is what would be required for the making refinements.

Since this new graph view contained an abundance of features unrelated to the purposes of OOGRE, the first task in developing the new graph view simply involved removing unnecessary visual clutter. Several features included in the tool either weren't fully functional or were deemed unnecessary for the purposes of the OOGRE graph, at least for the present. These included functionality to filter displayed graph nodes or edges by certain criteria, node searching and snapshot-taking functionality, as well as numerous view customization options, such as the ability to change node color and label position. Removing this functionality from the user's view involved hiding several toolbars, context menus, and view windows in the Eclipse perspective. In removing these extraneous features from view, I was able to focus on the essential drag-and-drop functionalities without encountering distractions, and this simplification may also be in line with the objectives of OOGRE, as additional complication in the graph window may interfere with the goal of clarifying the working of a software system. In any case, these additional nonessentials can easily be re-added in future work.

Next, I worked to make the look-and-feel of the OOGRE graph view resemble

[4]OOGUI is a prototype that Nariman Ammar started working on.

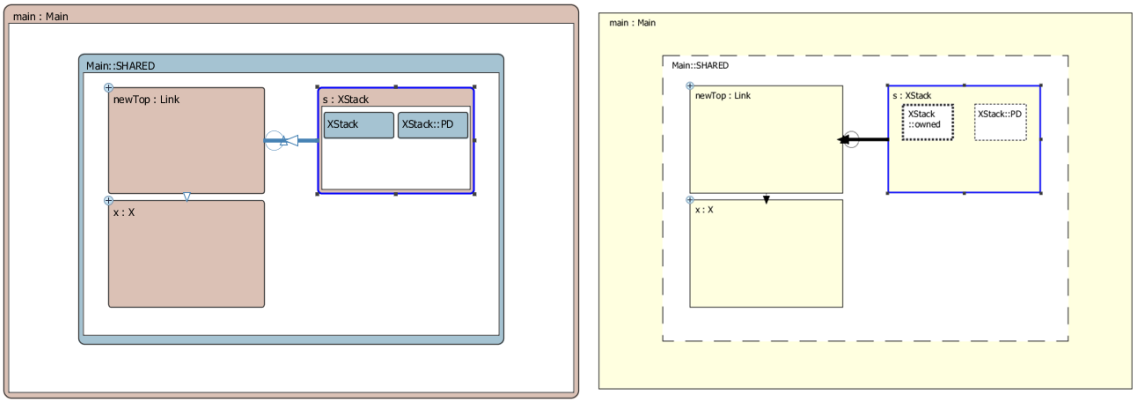

Figure 3.4: OOGRE graph view before and after

the static one currently used in ArchDoc. It was important to adhere to the current conventions being used to avoid confusion and comply with the existing OOG visualization conventions. This involved disallowing some user customization of node shape and color by assigning a default and hiding the menus allowing such customization. The original roundy graph nodes were changed to the rectangular ones used in ArchDoc. The baby blue color used in the graph for domains and light pink object color were changed to the white and pale yellow of ArchDoc. Additionally, the inner boundary of each node was filled in and its border was removed; its label was also fixed to the node rather than floating above it. Also, the edges of domains were changed made dashed and private domains were made thicker to differentiate them from public ones. Finally, the graph arcs were modified to be the same black color as ArchDoc and their arrowheads were moved from the center of the arc to the end. Fig. 3.4 below displays a view of the graph before and after these changes; on the left is the default and on the right is the graph matching the ArchDoc conventions.

After bringing the new OOGRE graph view into conformance with OOG conventions, the critical piece of the implementation project was to add drag-and-drop functionality to the graph. Since this was a feature not originally included in the framework, it took more time and effort to complete. The first step in solving this problem was implementing logic that caused a node to open up and reveal its children

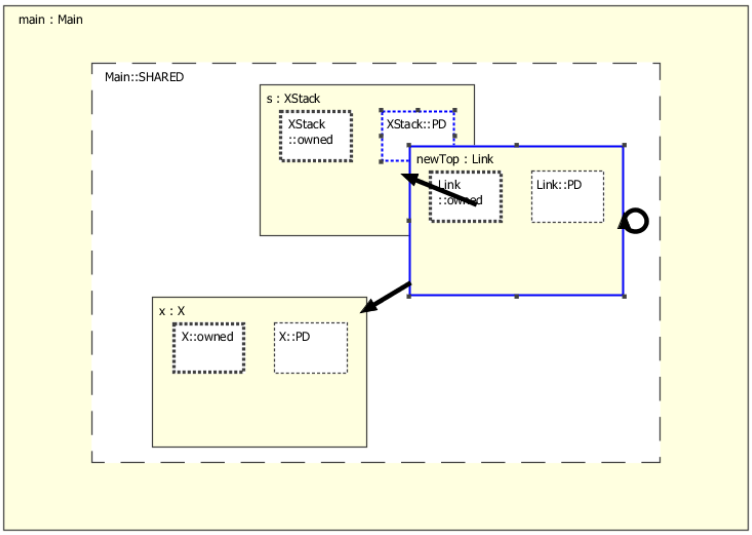

Figure 3.5: Dropping the newTop:Link object into the XStack::PD domain

whenever another node was dragged over it. This creates the notion of a dragged node and a target node, with the dragged node being the object that is to be dropped into a domain of the target object. Having accomplished this, I then created an algorithm to determine which node is the drop target, i.e. the node the user would like to drop the dragged object into. The difficulty lies in the fact that the dragged node could be touching several different nodes, so it was important to decide which of the nodes to open and select as the drop target. The generic solution is to open the leftmost node, or whichever node is entered first. The node currently targeted as the drop target is highlighted in blue. The user can continue to drag the node around the graph; when he releases the mouse, the dragged node will be nested as a child of the node highlighted as the drop target (if any). An example software system illustrates this functionality. Fig. 3.5 illustrates the public domain drop target of the XStack object with blue highlighting and Fig. 3.6 shows the dragged Link node object dropped and nested in the target node.

When implementing the drag and drop operations discussed above, drops into any domain or object were at first allowable in order to easily assess the functionality of the drag and drop mechanism. This was corrected by an algorithm that deter-

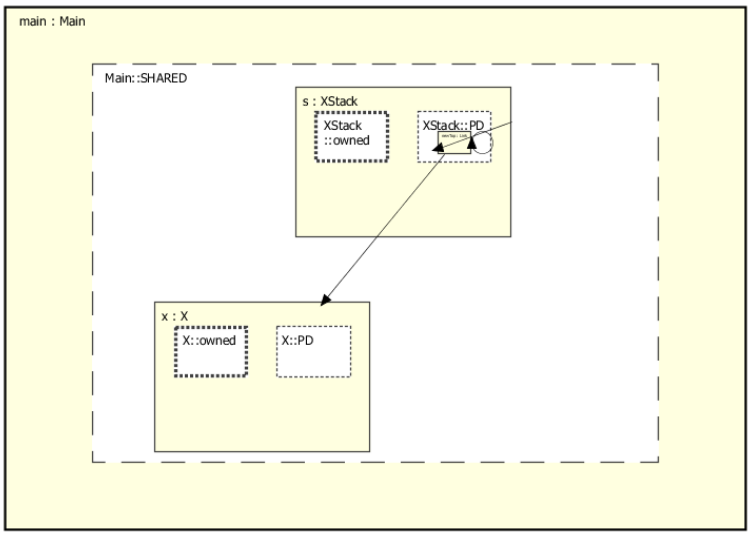

Figure 3.6: OOG after dropping newTop:Link into the XStack::PD domain

mined whether the object selected could be dragged (only objects may be regarded as dragged nodes) and whether the target being "hovered over" by a dragged node was a valid drop target (domains belonging to a valid node). After this logic was in place, it was necessary to implement additional functionality to move nested objects back out of domains, which required reapplying the framework's node layout algorithm. This completed the functional features of the graph view allowing for iterative refinements to the default OOG system abstraction.

The final step in the development of the project is the integration of the OOGRE graph view into ArchDoc to replace the former static graph view. Since ArchDoc produces a default OOG system abstraction in memory while the OOGRE graph uses a different method of creating its graph representation, slight reworking of code is required. In completing an integration with ArchDoc, the OOGRE graph will require a change to read the OOG from memory instead of the current method of reading from an existing XML file. This task concludes the merging of the OOGRE and ArchDoc tools.

Chapter 4: Discussion and Conclusion

4.1 Discussion

The outcomes discussed above were generally positive, with the most critical features, namely the drag-and-drop support for both the tree and the graph, being implemented and functional. This outcome allowed me to posit the thesis of this work, and its reliability can be verified by using the OOGRE tool. This is not to imply that the tool is completely mature; rather, there are several other fixes and improvements that could be pursued for future tool development. These include a fully implemented and complementary context menu for both the graph and the tree, since the graph view currently does not have such a menu. This feature would allow for greater user customization, increasing the potential gains to code comprehension. A similar context menu could also be restored to the graph edges to provide further details on source and destination nodes, and a few problematic features in the graph, such as node rendering while zooming could also be resolved. Further possibilities for implementation in the graph include built-in framework features like a node and edge searching tool and a filmstrip snapshot-taking tool. Lastly, a full evaluation on a large code system would be necessary to verify the usability of the tool as a whole. These considerations aside, the success of the primary research objective and the general favorable review from the preliminary tool evaluation give substantial confidence in the assertions of the thesis.

4.2 Conclusion

As a software analysis and code comprehension tool, OOGRE has many promising features that contribute significantly to the existing technologies supporting this field. Foremost of these features is its automatic addition of code annotations, which are

used to create OOGs. Previous tools relied on manual code annotations, and having experienced firsthand the difficulty of implementing these, it is obvious that OOGRE can simplify the task considerably, especially for novice programmers, such as undergraduates in a software design course. Also, the use of static analysis presents a reliable means of gaining understanding about system architecture, and the tool is compatible with popular and widely used technologies. A preliminary evaluation found that the largest usability concern of OOGRE is the learning curve involved in understanding ownership domains and object abstraction. Beyond this usability constraint, the drag and drop functionality in the tree and graph views of the tool allow users to interactively refine a default OOG to reflect their mental model of the system or gain greater understanding of the software. Thus, OOGRE appears well situated to deliver on its potential to improve the software engineering process by cutting down on time spent on code comprehension.

REFERENCES

[1] Abi-Antoun, M., and Aldrich, J. Static Extraction and Conformance Analysis of Hierarchical Runtime Architectural Structure using Annotations. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)* (2009), pp. 321–340.

[2] Banker, R. D., Datar, S. M., Kemerer, C. F., and Zweig, D. Software complexity and maintenance costs. *Commun. ACM 36*, 11 (November 1993), 81–94.

[3] K. H. Bennett, V. R., and Wilde, N. Software evolution and the staged model of the software lifecycle. *Advances in Computers 56* (2002), 3–55.

[4] Selitsky, T. A Front-End for an Ownership Object Graph Interactive Editor. Master's thesis, Wayne State University, 2010. Advisor: Marwan Abi-Antoun. Available: `www.cs.wayne.edu/~mabianto/`.

[5] Storey, M.-A. D., Müller, H. A., and Wong, K. Manipulating and Documenting Software Structures. In *Software Visualization* (1998), P. Eades and K. Zhang, Eds.

ABSTRACT

A USER INTERFACE FOR THE INTERACTIVE REFINEMENT OF AN OBJECT GRAPH

by

WESLEY TRESCOTT

August 2015

Advisor: Dr. Marwan Abi-Antoun

Major: Computer Science Honors

Degree: Bachelor of Science

A significant problem in software engineering is the time factor involved for developers to gain an understanding of the workings of a software system or to implement a target system architecture. The OOGRE tool has potential to mitigate this problem by its use of Ownership Objet Graphs (OOGs), which group objects into ownership domains to reduce system complexity, provide an ownership hierarchy, and abstract related objects into a consolidated object. OOGs are created by adding annotations to Java source code and using a behind-the-scenes analysis to create graph and tree views of the model based on the annotations. Users can gain an understanding of the system by refining the OOG to reflect their design intent. But this process requires modification of code annotations, a tedious and time-consuming process. OOGRE mitigates this problem by automatically applying annotations to the code. The work in this thesis makes the OOGRE tool even more user friendly by implementing node drag-and-drop functionality in the tree and graph so that users can make refinements to the default OOG. Rerunning the OOGRE analysis validates the changes made by the user. In interacting with this interface, the user can investigate system architecture and improve his understanding of the software, reducing time factors of code comprehension.

AUTOBIOGRAPHICAL STATEMENT

WESLEY TRESCOTT

EDUCATION

- Bachelor of Science (Computer Science), December 2015
  Wayne State University, Detroit, MI, USA