VISUALIZING THE IMPACT OF CHANGES IN SOFTWARE CODE

MATTHEW FOLLETT

Visualizing the Impact of Changes in Software Code

by

A thesis submitted to the
School of Graduate Studies
in partial fulfilment of the
requirements for the degree of
Master of Science

Department of Computer Science
Memorial University of Newfoundland

August 2011

St. John's Newfoundland

Abstract

Although software projects continue to grow larger in size and complexity, the typical methods for debugging these projects have not changed much over the past decade. Software is more modular, with code reuse becoming very common. This can lead to bugs manifesting themselves in one or more sections of code, but originating in a completely different area. This thesis focuses on the development and study of ImpactViz, a novel debugging tool that considers the object oriented nature of modern software languages such as Java, and uses visualization techniques to aid in identifying the potential origins of software bugs. Results from a laboratory evaluation help show that participants find the new program ImpactViz to be both useful and easy to use. The field trials performed have also helped define the benefits and limitations of using ImpactViz in certain situations.

Acknowledgements

I would like to thank my supervisor, Dr. Orland Hoeber, who has been a source of support and inspiration over the past two years. Without his aid and suggestions, I would never have been able to finish this thesis. His suggestions and comments have helped shape this thesis into what it is today.

I also wish to acknowledge the members of the UXLab, for their comments and suggestions while demonstrating and discussing this thesis, as well as the developers for the tools used for this thesis, including Eclipse, LaTeX, and Prefuse.

Finally, I wish to thank my parents, who have always pushed me to do my best reach my goals. Without their support and encouragement, I would not be the person that I am today.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Modern software development often consists of multiple developers simultaneously working on a single project [26]. Software code is commonly shared through a version control system such as Concurrent Versioning System (CVS) [34] or Subversion (SVN) [40]. Such software repositories allow the changes from multiple developers who are working on the same files to be easily managed. However, within a team environment, as people add new code and change existing code, the chances of new bugs and unexpected behavior appearing increases [26, 36].

In object-oriented programming [38], software code is divided into classes which represent conceptual objects. Each class may contain a set of methods which implement the functionality of the class. Classes are able to inherit or encapsulate other classes, and their associated methods are able to call one another. As a result, the interaction between method calls within even a small project can become rather complex [24]. For example, a single method may call numerous methods from its encapsulated classes, each of which may make method calls to their own encapsulated

classes. As a result, the method call stack may become very deep.

When a bug is accidentally introduced into a class, it not only affects the method in which it originates, but also manifests itself as a problem in other classes that make use of the bugged method. Since the manifestation of the bug may appear many levels deep in the method call stack, tracing the bug back to its origin can be a difficult and time consuming task.

Traditional debugging practices require the software developer to analyze the project code line-by-line, trying to detect if a bug exists in a given class. When code within other classes is executed via method calls, the number of classes that need to be examined as potential places where bugs could be introduced increases. As the depth of the method call stack increases, it can become very difficult for the software developer to track down the source of a bug. Further, as projects grow in size this debugging process becomes even more complex, resulting in a very time-consuming and tedious task.

1.1 Motivation

While developing software projects, it can be easy to not realize the number of classes that depend on a single class, or the total number of classes a single class depends upon. When bugs are introduced into the source code of one method, all the methods that depend on that one method, either directly or indirectly, are also influenced by this single bug. In projects where the method stack can be quite large, it becomes hard to track down the original source of the bug.

The goal of this research is to design, develop, and test a software visualization

system that allows software developers to quickly and effectively see the full method call stack and the recently changed methods. Visualization methods allow the user to easily ignore the classes that have not recently been changed, focusing on those that have changed and which provide services to the class in which the bug is manifesting itself. Using this information, software developers can quickly track down the original source of the bug and fix it, instead of the manual process of identifying each method in the stack, analyzing its source code and testing it for validity.

1.2 Research Questions

The key research question that this research addresses is: **What is the value of using a visual approach to representing the method call stack and changes to software code for the purposes of supporting debugging processes?** Here, the value can be measured in terms of being able to find bugs quicker, more accurately, and with higher satisfaction. Within this thesis, a prototype visualization tool called ImpactViz is presented and studied to address this research question.

1.3 Assumptions and Constraints

For the purposes of this thesis, the main type of bug we will be focusing on are "logical" bugs - bugs that are correct in the use of syntax but are not caught at compile time. These logical bugs also only affect the methods in the method stack and not class members, where the change in the value is seen in an unrelated method stack. This thesis is mostly concerned with the concept of how a minor flaw in the

software code can lead to larger problems in other areas of the software system. This will be performed by scanning the software code as plain text and modeling the class relationships.

This thesis will not be examining the problem of bugs being introduced in a multi-threaded application. Bugs of this nature cannot be examined via the source code alone, nor can we accurately predict the total number of threads that may be a problem. Also, while we are entering the domain of software engineering, this software would not be able to pick up cases of incorrect polymorphism, as this is more related to the design of a software project, instead of the implementation of it. Lastly, this thesis will not keep track of changes made to class or static variables. While bugs might be introduced by incorrectly changing the value of these variables, there are too many of these types of changes that can occur, but could be addressed in future work. Visualizing this type of impact, along with the method impact, would be too much information for a user to keep the visualization clear and easy to understand.

1.4 Organization of Thesis

The remaining thesis is organized into chapters separated into common themes. Chapter 2 discusses the background information required to understand this thesis, including topics such as information visualization and various works performed in the field of software visualization. Chapter 3 will discuss the design of the thesis project, ImpactViz, and provides a sample scenario of how ImapctViz would be used to assist in the debugging of a software project. Chapter 4 outlines the methods for evaluating ImpactViz using laboratory studies and field trials. Chapter 5 reports the findings of

these studies and the statistical analysis of the data. Chapter 6 provides a discussion on the outcomes of the evaluations and what new knowledge has been generated as a result of this research. Chapter 7 outlines the future work to be done on ImpactViz. Chapter 8 concludes the thesis, summarizing the findings and primary contributions of this research.

Chapter 2

Related Work

This chapter presents background research related to this thesis. The purpose is to help provide context to the decisions that were made during the thesis work, and a commentary on the current state of software visualization. Background information will be provided pertaining to the areas of information visualization, including how people see and interpret colours and the cognitive activities that an individual undertakes when evaluating a visualization. A brief outline of software visualization research is also included, along with a taxonomy that provides an organizing structure for the work that others have done in this domain.

2.1 Information Visualization

Information visualization is the field of research that deals with the design, creation, and study of visual representations of abstract information [43, 39]. The goal is to enhance the cognitive abilities of the user, allowing them to understand, explore, and interact with the data to gain a greater understanding [43]. Abstract data has no

spatial properties, which gives the designer the challenge of finding ways to represent the data in a useful and meaningful way.

Information visualization excels at showing large and/or complex data sets. Data, when shown as plain text or as a table of numbers, can lack context in relation to the other data also being presented. It can be difficult to see subtle features in the data, such as outliers, patterns, or relationships that could help a person understand the underlying features or meaning in the data. By visualizing this information, certain aspects of the data can be easier to recognize and data anomalies can become more apparent [5].

When designing an information visualization system, there are a large number of ways to represent data in a visual manner (e.g., the use of colours, positioning, size, shapes, etc.) [39, 25, 6]. It can be a challenge to finding the right combination of representation and data to create an intuitive visualization that a user will find helpful within a given context or task.

Another concern in information visualization systems is the idea of scalability. While a system may work perfectly with a small amount of data, when more data points are added, the system can become cluttered and confusing. It is up to the designer to find methods to reduce this visual clutter (e.g., supporting filtering operations, developing alternate methods for representing the data) or find some comprise in the representation that may not be optimal, but may scale well.

The nature of the data that the designer wants to represent influences how they should represent it visually. There are three fundamental types of data: numeric, ordinal, and nominal [39, 25]. Numeric data is data with a quantitative value, like the price of an item, and there exists the possibility of data existing between two

points (e.g., the middle of $7 and $8 is $7.50). Ordinal data is data that is not numeric but there is an implied order, like days of the week. Unlike numeric data, data cannot exist between two points (e.g., there is no day between Monday and Tuesday). The last type of data is nominal, where there is no ordering or numeric representation, such as named entities. As an example, we can separate pets into five categories: reptiles, dogs, fish, cats, and birds. Looking at the list, there is no inherent way to organize it, which makes categories for pet types nominal data.

In addition to the above fundamental data types, data can also contain relationships between entities. A common way to represent such relationships is via a node-link diagram [20]. Node-link diagrams are composed of two elements: nodes and edges. Nodes are normally represented as glyphs that represent a particular entity, like a person or class object. They can contain visual encodings to represent various attributes associated with the entity. Edges are used to connect nodes to show a relation exists between the two objects, like a family relation between two people or a method call between software classes. Edges can be directed (one-way relations, like "mother of" or "calls method of") or can be undirected (the relation goes both ways, like a relation between siblings). Both these elements can include additional data that can be visualized (e.g., we can use assign the value of age to a node, and using the connecting edges to calculate the similarity or closest common ancestor between the two data points) [39, 20].

One challenge with node-link diagrams is in organizing them effectively and efficiently. Not only do we have to display the node-link diagram, we should also strive to make the diagram clear and easy to understand [20]. Ideally, the graph should be laid out such that nodes that are related are directly linked and should be close to one

another, while keeping the links as short and clear as possible. One popular method to organize these diagrams is to use a force-directed graph layout [20, 14]. The force-directed layout applies a calculated force on each node to determine whether it should move or not. The movement of a node is based on the other nodes in the graph; nodes that have a direct link between them will pull themselves together, while nodes that have no direct link will move apart. This approach helps create node clusters, by pulling together nodes that have relations together, while pushing clusters of nodes with no relation to the other cluster away. While a force-directed algorithm does not always create the most clear diagram, it does bring related nodes into close proximity to each other and does allow for user interaction to move nodes, and have the entire diagram update dynamically to compensate for the change in position.

User interaction is a tool that helps make information visualization systems more useful [39, 43], giving users the option to interact with and gain a better understanding of the data. Through interaction, the user can request additional information on a particular subset of data (inspection), or adjust the focus of the screen to show more or less information (zoom and pan). Through the use of filtering, a user can specify a subset of information they are interested in based on some criteria (the method of selecting these criteria is dictated by the design of the system). The system can then filter out and hide any data that does not meet the criteria. Another tool used to aid users in finding data is known as focusing. Unlike filtering, uninterested data remains visible in the visualization, but the focused items are given a stronger visual presence. This can be achieved by adjusting transparencies or alpha channels, changing sizes of various objects, or adjusting the use of colours. The advantage of focusing over filtering is that it helps reduce the amount of noise, but keeps all the data in context

with the overall dataset [39].

Creating an effective information visualization system takes more than just visually encoded data and supporting interactive components: the system needs to be designed with the user's end goals in mind [39, 15, 9]. Understanding who will be using the system and what goals they want to achieve will help in deciding how to encode the information visually (e.g., understanding the user's needs allows us to decide the best method to put visual prominence to attract the user's attention) and what interactive tools will help achieve these goals. Understanding the experience that the user already has before using this system can aid in the training of the system by using vocabulary and symbols with which the user would already be familiar [15, 9]. The systems that take into account user expectations and previous knowledge are more likely to be perceived as useful and intuitive [43].

Since information visualization systems are composed of visual elements and human interactions, it is important to understand the core theories that guide visualization research. These include theories that guide colour usage (opponent process theory of colour), explain how visual stimuli are interpreted (Gestalt principles), and describe the cognitive gaps between evaluating a visual display and acting upon it (Norman's stages of actions). Each of these are described in detail in the sections that follow.

2.1.1 Opponent Process Theory of Colour

The opponent process theory of colour describes the method by which colour is interpreted by the human brain from stimuli received within the human eye [43, 19]. The

theory states that colours are interpreted along two chromatic channels (red-green; and yellow-blue) and one luminance channel (black-white). How the lightwaves translate onto these three channels is illustrated in Figure 2.1. Our brains are hard-wired to be able to detect differences among colours along these channels. Understanding this visual sensitivity helps provide guidance for using colour to represent data. However, due to differences in how numerical, ordinal, and nominal data are to be interpreted, we must take care that the colour encoding can properly be decoded.

When using colour to represent a numerical value, we may choose a colour scale that varies monotonically on one or more the channels, producing a relatively straight line through the colour space that is perceptually ordered. In this colour gradient, the maximum value of the data is represented by the colour on one extreme, while the minimum value is represented by the colour at the other extreme. Values in between may be represented by a proportional colour between the maximum and minimum colours.

If the data can contain both positive and negative values, special care must be taken to ensure accuracy in decoding. A light hue of a neutral colour (somewhere in the middle of the colour space) should be used to represent zero. The colour scales for positive and negative values should use colours in opposite directions that are progressively darker. For example, a zero value could be encoded as light grey, positive values in darker and darker shades of yellow, and negative values in darker and darker shades of blue.

Much like numerical data, when using colour to represent ordinal data, we may choose a colour scale that varies monotonically within the colour space. Specific perceptually ordered colours within this scale can be selected to represent the unique

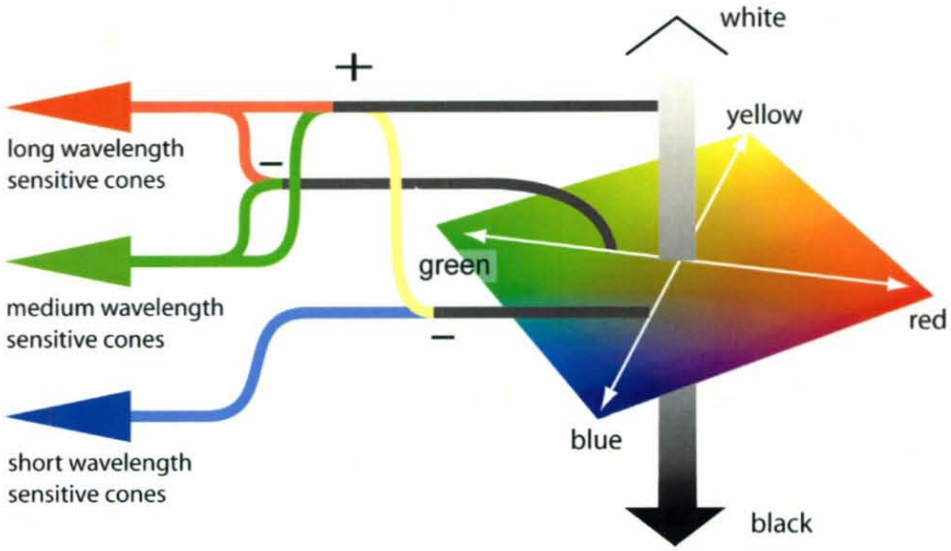

Figure 2.1: A diagram showing how long, medium, and short wavelength light is interpreted onto the three channels of the opponent process theory of colour; red-green, blue-yellow, and white-black.

elements within the ordinal data. While the ability to decode the colours into their ordinal values depends on the number of steps within the ordinal scale, the ordering of the data will be decodable. For example, if there are a large number of potential values for the data, viewers may have a hard time decoding the data to a unique value; however, the approximate location of the data element within the ordinal scale will be decodable.

For representing nominal data for labeling with colour we want colours that are both distinct and have unique hues. According to Ware [43], there are 12 colours that can be reliably used for labelling purposes: red, green, yellow, blue, black, white,

pink, cyan, grey, orange, brown, and purple. The first six of these colours are the extreme ends of the colour channels, while the remaining six colours are combinations of colours from these channels.The average person has been found to be able to easily distinguish between these colours and not interpret them as the other selected colours [17]. If we placed these 12 colours onto a colour cube created by the opponent process theory's three colour channels, these colours are placed far away from each other. This means that there are as perceptually different from one another as possible. Adding more colours for representing nominal data is possible (e.g., choosing additional "in-between" colours). As more colours are added to this colour space for nominal data labelling, the ability for the user to reliably decode the data diminishes. A practical limit here is carefully chosen to allow for each colour to be sufficiently distinct from one another.

2.1.2 Gestalt Principles

The Gestalt principles are theories that relate to visual perception and how people group items together and perceive foreground from background [27, 43]. Although there are a large number of such principles, the ones that are relevant to this research are proximity, similarity, and connectedness.

The principle of proximity states that items that are closer to one another are often perceived as being grouped together and thought to be related. By placing items that are related to one another, like when we have a cluster of dots on a graph, the observer will quickly interpret that the items are related in some fashion.

The principle of similarity states that items that look similar are also thought to be

related. With this principle, we can use similar shapes and colours to represent certain data and users will instinctively think they are related. We often see this principle used with symbols on a map, to indicate common points of interests (restrooms, tourist spots, information booths, etc.).

The principle of connectedness states that items with a visual connection (like two nodes connected by an edge) are perceived as related to one another. We can see this in a family tree, where family members have a connection going from one individual to the other. This is one of the strongest grouping principles and requires no additional information to help users form an opinion on the objects [43]. Further, because the principle of connectedness is so strong, it can be used to represent a different type of relationship than what is encoded with proximity and similarity, without confusing the user. People will readily interpret such a display as different classes or relationships.

Understanding these principles allows us to design visualizations that can be intuitive and interpreted in a manner we expect. With an understanding of how the user will interpret certain visual representations, we can design our software to take advantage of these principles to deliver a clear and easy to understand the data being shown.

2.1.3 Norman's Stages of Action

Norman's stages of action [30] describe the cognitive steps that an individual goes through when performing an action, starting with the individual evaluating the situation and planning the action. The events are broken into two separate stages; the gulf

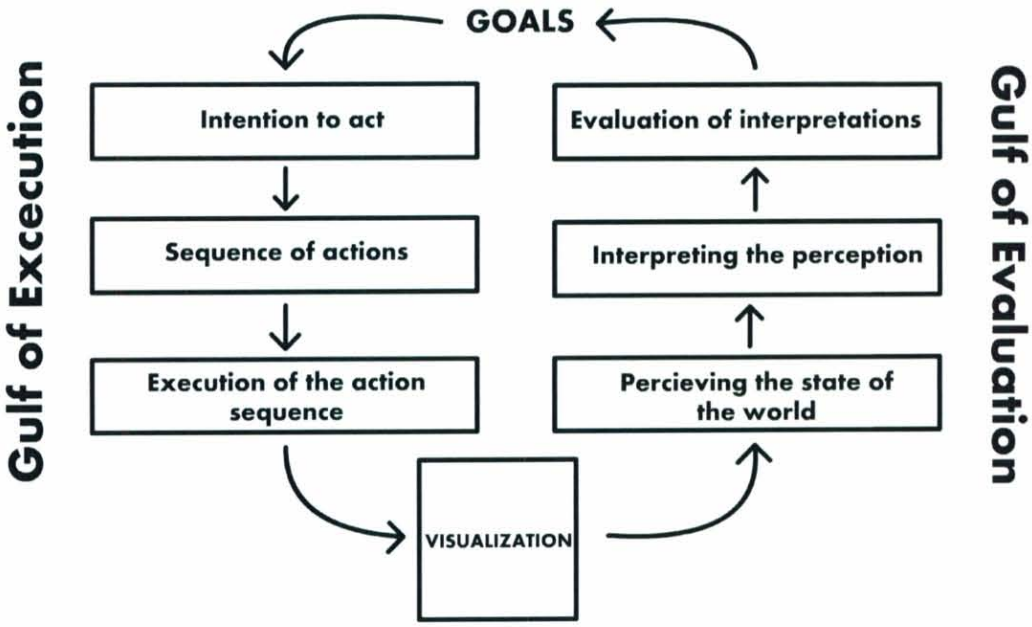

Figure 2.2: A visual representation of Norman's stages of actions that an individual goes through when performing an action, according to Norman's stages of actions, including both the gulf of evaluation (right) and gulf of execution (left).

of evaluation and the gulf of execution. These stages are part of a cycle, where the user observes the visualization and travels the gulf of evaluation to understand what they are seeing with respect to the goals they wish to achieve. Then, the user travels through the gulf of execution as they interact with the system to achieve their goals. When the execution is completed, it impacts the visualization, and the user starts to go through the gulf of evaluation to interpret those changes. The cycle continues until the user finally has achieved the goals and no longer needs to interact with the visualization system. The cognitive cycle can be seen in Figure 2.2.

Within the gulf of evaluation, there are specific cognitive steps a user takes to un-

derstand the visualization being presented to them. The user's first step is perceiving the state of the world, to see the current state of the visualization. Then, the user starts interpreting what they have perceived, and begins to understand what is being presented and how it has changed. The user then starts evaluating the outcome, to understand if the goal has been accomplished. After evaluating, the user may decide to continue to use the system or decide they no longer need the visualization system and leave. If the user decides to continue to use the system, they will start forming a goal. This goal may be a new goal or, if the evaluation earlier concluded that their previous goal was not reached, they may decide to continue or modify the previous goal. In either case, the user then moves to the gulf of execution.

The gulf of execution explains the cognitive steps a user takes to achieve their goal with the system. A user forms the intention to act based on their goal, in order to bring themselves closer to accomplishing the goal. With an intention formed, the user then starts to plan what actions they believe will help in achieving the goal. With the action sequence planned, the user then performs the planned action. Once these actions are finished, the person moves back into the gulf of evaluation to evaluate the impact their actions have had on the visualization and if their goals have been met, and if they should continue their previous goal or form a new one.

When the user has finished crossing the gulf of execution, the user's environment has now changed as a result of the interaction. The user enters the gulf of evaluation to examine what has changed in the visualization. Upon finishing the gulf of evaluation, the user will then enter the gulf of execution, to make changes, to formulate a goal, and to make changes to the environment. The user will move back forth between the gulf of evaluation and the gulf of execution, until the user can no longer formulate

any more goals or if the user no longer wishes to use the software to achieve the goals. At that point, the user will then exit the software.

By applying Norman's stages of actions to designing a information visualization system, we can understand the cognitive steps an individual performs while using the system. With this understanding, we can use representation techniques and interactive designs that will help them achieve their goals faster and much more easily. From the gulf of execution, we can understand the steps that a user will undertake to achieve their goals and help provide the tools and actions to perform these actions as efficient and easily as possible. By understanding the gulf of evaluation, we can understand the steps that a user will take to perceive, interpret, and understand the visualization, and provide representations that help make this clear and easy to understand.

2.2 Software Visualization

Information visualization systems that focus on software code and related statistics is referred to as software visualization. Large software projects can become very complex and difficult to analyze and support without the proper tools. Software visualization systems can be used to help users analyze the software architecture, learn developer information, the frequency that changes occur, execution tracking, and how the software evolves. Through these tools, software developers can learn interesting patterns and anomalies in software systems, and investigate them further.

It has been suggested that the methods for visualizing software code can be divided into three categories: those that visually represent lines of code, files, and

Table 2.1: Common software activities and software visualization systems

	Software architecture	Developer information	Frequency of changes	Execution tracking	Software evolution
Line Based Visualization					
SeeSoft [10]	No	Yes	Yes	No	No
Augur [13]	No	Yes	Yes	No	No
CVSScan [42]	No	No	Yes	No	Yes
File Based Visualization					
Release History [16]	No	No	Yes	No	No
Voinea et al.'s Work [41]	No	Yes	Yes	No	No
Folder Based Visualization					
StarGate [31]	No	Yes	No	No	No
Voronoi Treemap [2]	No	No	No	Yes	No
Class Based Visualization					
UML [12]	Yes	No	No	No	No
Byleas and Telea's Work [3]	Yes	No	No	Yes	No
Pich's Work [32]	Yes	No	No	No	No
Hierarchical Edge Bundles [22]	Yes	No	No	No	No
ImpactViz [11]	Yes	No	No	Yes	No

folders [1]. Given the prevalence of object-oriented software development practices, a fourth representation is also appropriate: those that visually represent the software project's classes and class interaction. Important and interesting work in each of these categories are summarized in Table 2.1, and described in more detail in the sections that follow.

2.2.1 Line Based Visualization

Line based visualization systems are software visualizations that represent attributes related to the individual lines that make up software files. These attributes could be the author who last changed a particular line, when the same line was last changed, the syntax of the line, etc. The tools typically help users understand how a particular section of code was written or changed, with the goal of helping them to understand the purpose of each line and how it relates to the file overall.

The seminal work in visually representing lines of software code is SeeSoft [10]. By scanning a repository history, the system uses colour to visually represent how long ago each line of code was changed. Through the use of a colour range, a user can see what areas have undergone changes recently and what lines of code have remained stable for a long period of time. Lines of code that have recently underwent a lot of changes might contain new bugs. Users of SeeSoft can also modify the colour encoding to represent other information such as the author who last modified the line as well as the syntax of the line. A number of other researchers have expanded upon the fundamental methods in SeeSoft, including Augur [13] and CVSscan [42], which have added the ability to see when new lines have been added to the class and how a file can grow over time.

The primary difficulty with visually encoding changes on a line-by-line basis is that the length of the line of code has an impact on the visual weight it carries in the interface. Similarly, multiple consecutive lines of changed code will appear as more important that the change of a single line of code. While identifying which lines of code have changed may assist the user in determining where a bug may have been

introduced, it is still up to the user to inspect each line of code manually for the source of the bug.

2.2.2 File Based Visualization

While the previous visualization method focuses on individual lines, file based visualization puts emphasis on the files that the lines are contained in. Such visual tools might be used to show properties related to files, including when the file was last modified, the date the file was created, the file type, the file size, etc. File based representation can help a debugger or a software developer to identify the files that are being changed considerably or frequently, and may need to be inspected for the purposes of quality control.

An example of a file based visualization is Release History [16]. Release History shows how much a file has changed between software releases. During testing phases, this system can be very useful to identify which portions of the source code have changed the most. If a new bug is identified between releases, Release History can be used to identify files that are candidates for the source of the bug (i.e., those that have changed considerably since the previous release). However, this approach may not be particularly effective if there is no stable baseline against which to compare the system or if a large amount of these changes were in the documentation and comments portions of the file.

Another example of a file-based representation is Voinea et al.'s work [41]. In the project-based view of this software, each individual file is visually represented, as well as a representation of each file's history in the project. This software visualization

tool shows when a file was first included in the project and, as changes occur to that file, who was the last to make any changes. From this, users can see when a large change in architecture occurs (when a large number of new files are introduced) and when new software developers start making changes to a particular file changes (based on when the colour of a file changes in the timeline).

2.2.3 Folder Based Visualization

When representing a group of files contained in a common folder, we refer to this as a folder based visualization. These types of visualizations allow software developers to see changes and attributes that are common to files that we know are related somehow. Some attributes that a user may want to visualize include folder size (either total number of files or amount of memory used), dates of changes, and frequency of changes. Since the folders in our software projects commonly have sub-folders of a finite depth, these software visualizations need to show both the folder file size and the depth of a particular folder. We can also see if other folders have common attributes (e.g., changes made on the same date) in order to identify potential relationships to other groups.

As an example of folder based visualization, StarGate [31] uses an abstract visual representations to show the relationship between developers and the folders they access and change. A ring is used to represent all the folders of the project with the lowest level of the ring representing the root. The portion of of the ring that each folder is assigned to is based on the total number of files in that folder and its sub-folders. The deeper that the folder tree goes, the more layers the ring will use to

represent the depth of the tree. Inside the ring, dots are placed to represent individual developers. As a developer makes changes to the files in the folders, the developer dots are placed closer to the folders they have changed. Software developers can use this system to identify who has been modifying different areas of a project, and where specific developers have been focusing their attention.

With StarGate, a user can quickly track down a developer in a large software team who would be the most familiar with a particular aspect of the software code. In the case of tracking a new bug, we can approach the developer to help provide context to the code. However, tracking down a particular bug is not aided by this software visualization system, but instead highlights an individual who can help. There are some difficulties with the approach, however. In cases where a single developer works exclusively in two folders that are opposite each other on the ring, the developer will appear closer to the center of the ring. This developer will appear equally involved in all areas of the project, instead of being seen as an expert in two areas.

A popular way to visualize software projects based on folders is by using treemaps [23]. Treemaps are visualization methods often used to visualize hierarchal data, in which we preserve the hierarchal structure of the data. A space is defined (based on some attribute, like total hard drive size or total number of lines of code) in which a root folder is represented. This area is then split into proportional sub-spaces based on the individual folder and files that belong to that folder. The area of the sub-folders are then further divided based on their contents, recursively, until there are no more folders. An example of this is an implementation of Voronoi treemap [2], in which the authors use complex polygons to represent the folders representation, rather then the traditional rectangle representation. The data is based on package information,

which is often present in Java projects. The area given to each package or class is based on various software metrics used to identify which classes and methods are frequently used, so that users of the system can see which portions of the software is frequently depended on.

2.2.4 Class Based Visualization

Although class based visualization is not outlined in Ball and Eick's taxonomy of software visualization [1], it does represent an important group of software visualization methods. Software classes exhibit many common properties that are also present in files (e.g., date of creation, last modified, size, etc.). However, classes also exhibit a property that is fundamentally important to object oriented programming: relations to other classes. In large software projects, it is uncommon to have classes work in complete isolation: classes typically make method calls to other classes, and may inherit properties or methods from classes (parent-child relations, interfaces, templates, etc.) [26, 38]. It is these relations that make classes very distinct entities from the files in which they are stored. These unique properties give an advantage to help accomplish tasks that a software developer would be interested in performing, including debugging.

By taking advantage of the class information, software visualization tools can be used to represent the interaction within a software project. For example, the class diagram within the Unified Modelling Language (UML) [12] produces a visual representation of class dependencies, inheritance, and encapsulation. While UML notation was not meant to show how changes propagate through a software system, it can show

which classes are related to a class in which a bug is manifesting itself, and provide a starting point for debugging. As well, Pich et al.'s work [32] evaluates classes based on the number of direct dependencies that they are responsible for. Taking this number, it places classes that are more dependent on others higher on the graph. This placement shows the user what classes are highly depended upon. If one of these classes becomes infected with a bug, the manifestation of the bug would quickly propagate through the system. As such, this tool allows the software developer to quickly identify these classes and ensure that they are well debugged at all times. Holten's work with Hierarchical Edge Bundles [22] uses a radial tree to show how classes are related across the entire software project. With Hierarchical Edge Bundles, we can see both how entire subsystems work with one another at a glance and gain the ability to see which classes are interacting with other classes. Unfortunately, the usefulness of this software visualization system is limited, offering no other information than to allow us see the connections between software subsystems and their individual classes.

Others have extend UML to provide more information, such as the work done by Byelas and Telea [3]. However, given that UML is already a very graphically-rich representation, doing so runs the risk of producing a visualization that is cluttered with coloured edges, edges with symbolic meanings, and coloured regions that stretch across the screen. The system tries to detect what software classes share common properties that should be of interest to the user and refer to these items as an area of interest (AOI). In this system, the software diagram is represented by using a UML diagram to show how classes are related to one another and use a coloured region to represent an AOI and the software components related to it. The end result is to help show how changing one component might have consequences in other areas of

the software project.

As will be discussed in more detail in Chapter 3, the software developed in this thesis, ImpactViz, can be classified as a class based software visualization technique. The program visually represents the connections between classes through method calls and the method call stack. ImpactViz shows the relations between class in a project and helps users to see the ways in which the classes in the project interact with one another, both directly and indirectly.

Chapter 3

ImpactViz

ImpactViz is a software debugging tool that visually represents the method dependencies between classes in a software project. It allows users to examine how changes in the software code can propagate throughout the entire system. Using the source code of the project and the related revision history data, ImpactViz generates a graph layout that illustrates the method call stack dependencies of each class and visualizes where the changes occur. A change is recorded when ImpactViz observes a change in a method's code from one revision to the next, marking the revision and method where the change was observed. The users can filter the revision history data, showing only changes within a specific time period. They may interact with the system to select a class and discover the other classes on which it depends, visually identifying whether any recent changes have occurred that could have an impact on the selected class. How many classes that are affected by a change in a single method can be observed in large coloured regions, referred to as change impact regions. Figure 3.1 shows how the entire information visualization tool is laid out.

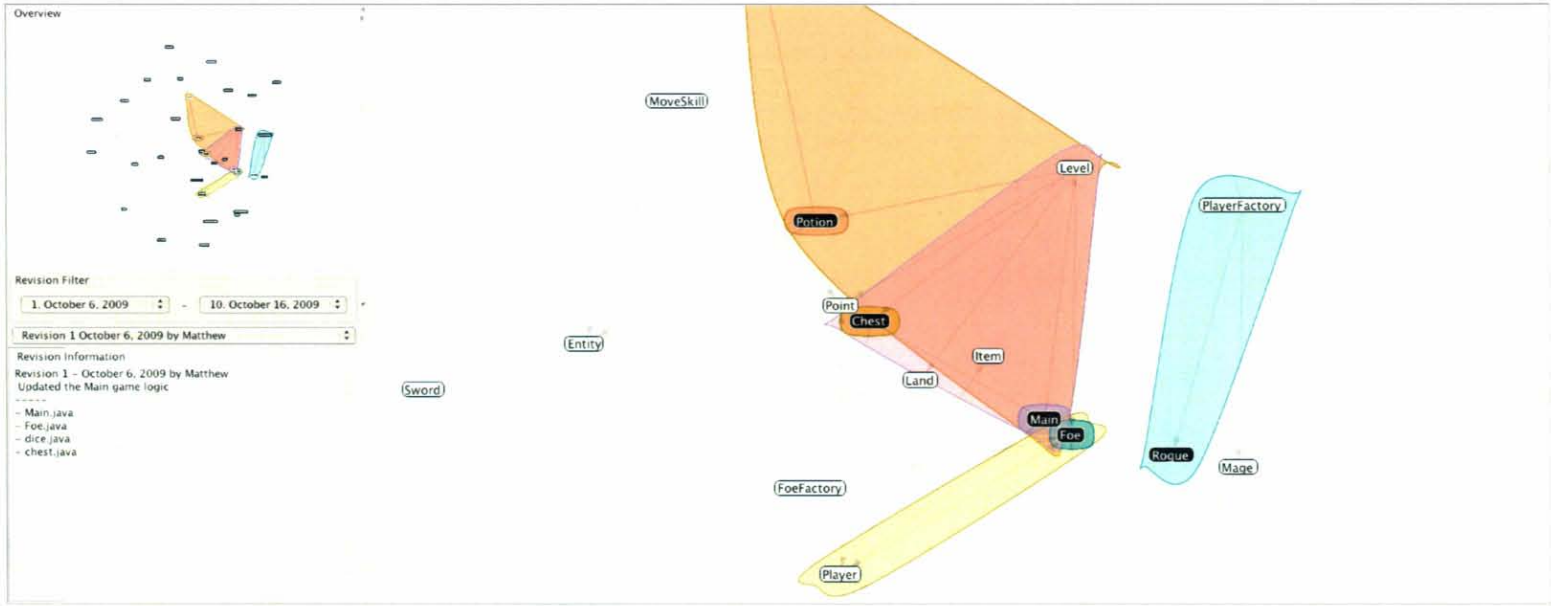

Figure 3.1: A screenshot of ImpactViz in action. ImpactViz is divided into three sections; the overview, the revision filter panel, and the graph representation of the changes and their impacts in the software system.

The goal of ImpactViz is to allow users to visually identify recent changes in the software code that may be the source of a bug in a particular method. By allowing the user to select the classes that they know are being influenced by a change, they can quickly trace the bug from where it has manifested itself to classes that might be responsible for the bug. Users can examine these classes that are highlighted more closely for programming errors.

The fundamental assumption with using ImpactViz is that the software developer is aware of some prior revision within the repository in which the bug was not present. This knowledge allows the user to set the revision history filter to only show changes that have occurred since this known bug-free revision. The user may also be aware of a tighter window in which the bug was introduced, allowing them to also filter out recent revisions. As such, any changes before or after this known window in which the bug was introduced are ignored, allowing the user to focus on those classes that have changed and may therefore be the source of the bug. While previous work with delta debugging [44] has been performed using automatic testing, not all software systems can easily use unit testing and some cases require human judgment. ImpactViz allows a knowledgeable user to debug in an effective manner, rather than applying a brute force algorithm to produce a list of alternatives for when and where the bug was introduced.

The current implementation of ImpactViz supports the construction of the method call stack dependencies for a Java project and using an SVN repository to manage the changes in the software project. However, the techniques are general enough to support any object oriented programming language and any source code repository that maintains a history of changes.ImpactViz uses Java framework called "Prefuse"

[18]. Prefuse is information visualization rendering engine, which helps provide aid in rendering graphs and the baseline interactions with those visualizations, while providing the ability to add additional interactions and visualizations. Prefuse helps provide a starting point when developing new information visualization projects, while flexible enough to allow developers pursue their own concepts and ideas for a new visualization tool.

3.1 Representation of Classes and Changes

ImpactViz uses a graph based representation to show the relationship between classes, using a force-directed algorithm [14] to organize the graph. ImpactViz also uses colour-encoded change impact regions to help show the overall impact that a single change can have on their entire software system. The nodes of the graph are also visually encoded, denoting the difference between classes that have seen a change over a desired time period.

Change impact regions which surround the nodes are used to represent the overall impact that a single change has on the system; each class that is encapsulated in the region has been influenced by the same method change. If a user knows that a method contains a bug, there exists a possibility that the other classes in the region have been affected by the same bug and may be the root cause of some other bugs observed. Change impact regions can overlap, showing that some classes can be influenced by more than one change.

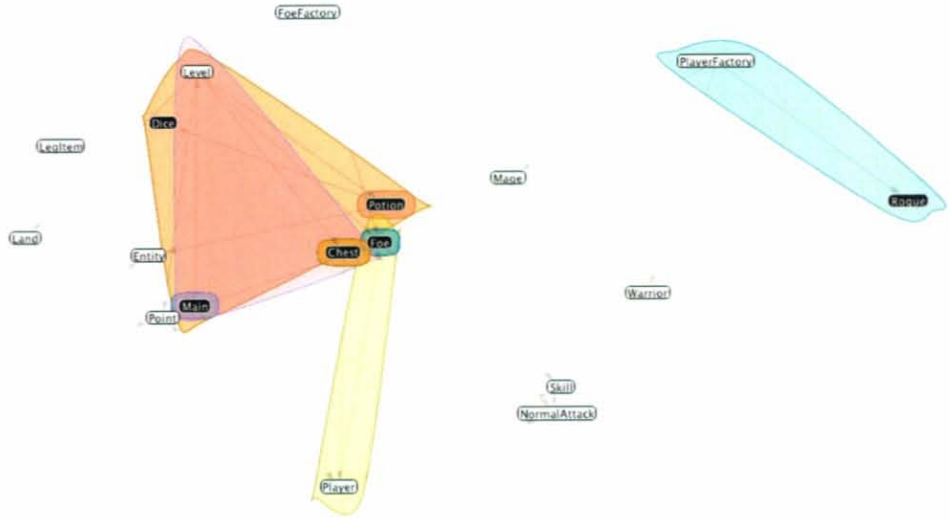

Figure 3.2: A screenshot of the primary visualization window in ImpactViz. Classes that make method calls to one another are represented in a force-directed graph. Nodes that have changes within the selected revision range have a black background. The impacts that these changes have on other classes are encoded using colour-encoded change impact regions. The graph presented here is not the complete software graph but a zoomed in portion of the graph.

3.2 Visualization Techniques

Since ImpactViz is a software visualization project, several visual techniques are being used to help ensure that the user of the program will be able to have a maximum understanding of the visualization, including the opponent process theory of colour [19, 43] and the Gestalt principles [27, 43]. Each visual item was carefully designed using different visual principles. The main visual items in ImpactViz are the nodes, their colouring, the change impact regions, and the graph layout.

3.2.1 Graph Layout

ImpactViz uses a graph representation of the method call stack. Each node represents a single software class; edges represent method calls between the connected classes. The direction of an edge shows the method call direction: an edge pointing from class A to class B illustrates that class A calls a method from class B (i.e., that a class of type class B is encapsulated in class A, and that a method in class A is calling a method from class B). This edge representation places the focus on the direction in which methods are being called, similar to how UML [12] class diagrams use the direction of an association or ownership. The graph representation can be seen in Figure 3.2.

ImpactViz uses a force-directed graph layout to organize the class nodes within the display. By using a force-direct algorithm [14] provided by the Prefuse API [18], nodes that are directly related to one another pull towards each other and repel nodes that have no direct relation. Through the usage of this graph layout, classes organize themselves into clusters based on their relationships, allowing classes that are related be close to one another. In addition, if there are two subsets in which there are no common connections, these two sets will push themselves away and help show the user small pockets of code with no relationship to other areas in the software. This can help separate different modules or show classes that are no longer in use in the software system and should be considered for removal (classes that are not part of the main system and are pushed to the extremes of the graph).

Another advantage of using the force directed algorithm is that it allows the user to drag nodes into open areas and have the entire graph structure dynamically

re-organize itself to take into account the new positions and re-create the clusters discussed earlier. This helps the user remove a class from a dense cluster of classes to examine it more closely for any relationships that they may find interesting.

In order to help control the graph, a graph control panel has been included on the left side of the screen. This gives the user access to several actions, including the ability to move to other areas of the graph quickly via the overview of the entire visualization on the top, the ability to filter changes between a time period that the user is interested in examining the changes, and the ability to examine notes left by the people who made the changes on the bottom.

With the force directed algorithm pulling directly linked nodes together, it allows for the Gestalt principle of proximity to come into effect. Nodes that are placed near one another supports the interpretation that they are related. The edges connecting the nodes are unweighted and each edge is treated the same, despite the total number of method calls between the two classes. This was done on purpose to help avoid situations where two classes are very strongly linked together, such that they are too close to be readable. By being able to ensure a relatively healthy distance between the nodes, we can help increase readability and reduce the total number of false-positives of classes being contained in a the change impact regions. As well, since we push away unrelated nodes, the opposite is also being implied; the nodes that are not close to each other are not closely related. As well, since we are using a line to connect linked nodes (classes) together, we are also using the Gestalt principle of connectedness to indicate a strong relationship between the nodes.

Another alternative to the force-directed algorithm is an implementation of an Euler diagram algorithm [35]. Euler diagrams are excellent for organizing sets of

data clearly, but make interacting and moving individual nodes difficult. For the purposes of representing class relationships, a traditional graph layout makes it easier for users to see the relationships between classes and to interact with a single class that is of interest.

3.2.2 Node Colouring

ImpactViz takes advantage of the principles derived from the opponent process theory of colour [19, 43] to help create several different visual distinctions among the class nodes. Class nodes are represented using two different colours: white to represent a class that hasn't changed within the specified revision date range, and black to represent a class that has been changed. This choice of colours on the extreme edges of the luminance channel ensure that the user can readily perceive the two different classes of nodes.

The visual encoding of nodes allows the software developer to visually trace the important classes; the classes that have recently seen a change made to them. During this process, the developer can ignore white nodes, as these are nodes that have been unchanged and spend more time analyzing the classes represented by the black nodes instead.

Through the usage of Gestalt principle of similarity, users of the system will associate the similarly coloured nodes as having some related attribute to make them the same colour, which will be true since the colour of the nodes dictate whether a change has been observed in the class that the node represents.

3.2.3 Change Impact Regions

The impact that a particular change in a class can have on the project is determined at the method level. Each change in a particular method for each revision in the software repository generates a separate change impact region. These change impact regions are subsets of the method call stack graph, including the class in which the method is defined along with all other classes that make use of the changed method.

The change impact regions are visually encoded as coloured regions layered on the background of the method call stack graph, enclosing all the class nodes that can make use of the changed method. Since the human eye can only reliably differentiate between 12 unique colours for encoding nominal data [43], the change impact regions are visually encoded using a set of ten distinct colour hues, with the remaining two colours used for node colouring, as described above. Obviously, there may be more than ten change impact regions that need to be represented. To address this issue, colour hues are reused for multiple change impact regions; wherever possible, this reuse occurs for disjoint regions. ImpactViz uses the Prefuse API to render the change impact regions, using a convex-hull algorithm. Using this algorithm can produce false positves, when classes not related to the change drift into the change impact region due to the force-directed layout used to help organize the nodes of the graph. While there are other algorithms that can avoid this issue, they required too much computational resources to make ImpactViz work as a real-time system. Users can identify the actual members of the change impact region by clicking the region. If Bubble Sets [7] can render in real time a large data set, it would be worth while to switch over, due to the reduction of false positives of nodes that look like they belong

to a change impact region.

Two scenarios are possible for the change impact regions. A change impact region may encapsulate only a single class (i.e., the class in which a change has been made). In this case, the change in the class is local and is not impacting any other classes within the project. Therefore, if a bug is manifesting itself elsewhere in the project, its source cannot be in this class. In other circumstances, a change impact region may encapsulate multiple classes (i.e., the class in which the change has been made, and other classes that make calls to the changed method). In this case, if the bug is manifesting itself in one of the classes within this change region, the changed class is a candidate for the source of the bug.

As seen in Figure 3.2, change impact regions can overlap one another. This can occur when a particular class is making use of methods from multiple other classes that have been changed. Or it may occur when multiple methods have been changed within a particular class, and these methods are being used by different sets of other classes. The colours used to encode the change impact regions are rendered using partial transparency, allowing the developer to see the overlapping regions and interpret the extent of the change impact regions.

The coloured change impact regions take advantage of two Gestalt principles, closure and proximity, to assist the user in perceiving that classes in the coloured regions are related. Through the use of the force-directed layout, classes that are related are pulled closer together, keeping the coloured area as small as possible. In cases where a class may be impacted by several different changes, the coloured regions are overlapped. Due to partial transparency in the coloured regions, the user can see the overlapping regions and interpret that a class is being influenced by more then

one change.

This visual encoding can allow the software developer to visually trace the manifestation of a bug to its source class by following the directed edges that connect a class node to a changed class node within the same change impact region. Paths that exit the change impact region indicate method calls to classes that have not changed.

3.2.4 Crossing the Gulf of Evaluation

The representation of ImpactViz was designed with the gulf of evaluation from Norman's stages of action [30] in mind. When the user first loads the program with the software system data, the user is greeted by a large cluster of classes. The user first perceives this visualization, examining what is being presented. At this stage, the user may notice large clusters of classes, colour-encoded change impact regions, as well as classes that are no longer part of the software system that are being pushed to extreme edges of the graph. The user then starts to interpret what is being presented. This includes decoding the change impact regions, examining the graph for classes of particular interest, and examining the node relationship with other classes. The user then attempts to make sense of what they are seeing with respect to their current task goal for using the system (i.e., discovering the source of a particular bug). They then determine whether their goal requires any interaction with the visualization in order to be achieved. Such interaction with the system (as described below) will result in changes to the visual display, which require the user to again cross the gulf of evaluation. This loop continues until the user no longer needs to interact with the system in order to resolve the goal.

3.3 Interaction Techniques

ImpactViz uses a variety of interaction techniques to give the users a great deal of control and to help them find the information they are interested in. Using the revision filters, revision information, node selection, node dragging, and panning and zooming, the users are able to manipulate the visual representation to help track down the origins of bugs and help see the overall software structure and relationships between classes.

3.3.1 Revision Filter

The revision filter is responsible for adjusting the colours of nodes depending on if there was a change in the revision dates and filtering what change impact regions are visible. The user selects the time period in which they are interested. From this time period, the user can see the changes that were made and how these changes influence the overall project. If the user's goal with this visualization is to track down a bug, they can use information from when the system was last working as intended to see the changed classes and the impact these classes have had on the system. This allows them to limit what is being shown to only the changes that could be the cause of the bug, since the impact change regions only show the changes since the bug's introduction. This allows ImpactViz to remove all the clutter that can come from having a very large revision history to a narrow time period that the user is actually interested in investigating.

The change in colour that occurs when adjusting the revision filter, even though the change impact region has not changed in size, was designed on purpose. Without

this feature, there are cases in which a change in the revision timeframe would yield no change in the visualization, as each change impact region could receive more than one changes over the lifespan of the project, which only alerts the user to the idea that there exists one or more changes to that change impact region user can take as the program not working. The change in colour now alerts the user to the updated change impact regions and the overlap in the regions can help provide a more accurate number of the number of changes in the change impact region.

3.3.2 Revision Information

At any point, the user can examine information about a particular revision. There is a drop down menu that displays the revision number, the author of that revision and the date on which the revision was made. The user can select a revision and see the comments that the author made about the revision as well as all the files that underwent a changed during that revision.

3.3.3 Node Selection

When selecting a class node, the method call stack that the selected class relies on is highlighted. This selection of a class node operates as a focusing technique. The method call stack of classes is brought to focus by adjusting all the other classes that are not in this selected class's method call stack to make them partially transparent and smaller. The edges that are also not part of this class stack are also made partially transparent, giving the connecting edges that are related to the selected class more visual emphasis.

Using the selection method, the user can analyze the chain of classes that have focus and examine these to see what classes have undergone some sort of change during the selected revision range. This gives the user some insight into which classes should be analyzed closely if there is a bug manifesting itself in the selected node. If the user was more interested in seeing how the classes interact with each other, the user can see the classes that the selected class makes a direct call to and how close these class calls are to the original design of the software system, as specified by the UML or other software artifacts developed for the project.

While we could also filter out all the change impact regions that are not related to the selected node (much like how we put less focus on the nodes and edges), we feel this might be confusing. We already allow for filtering based on revision intervals, adding an additional filter from selection might get confusing. Secondly, we do not want our users to neglect the other impact regions. We wish to ensure that, once they start investigating one region, they might come across other interesting impact regions (potentially, even larger ones), and we wish not to hide this information from the user. Since the change impact regions already are transparent, there is little else we can do to them to put less of a focus on them. Based on the two previous reasons, we have decided not to include change impact regions as being influenced by node selection.

3.3.4 Change Impact Region Selection

Due to the usage of a force-directed layout, nodes can be pushed into change impact regions while not beloning to that particular change, creating the illusion that a class

belongs in the change impact region, while containing no references to those classes in any fashion. We can overcome this by allowing the users to select a change impact regions by clicking on it. By clicking on the change impact region, all nodes and related edges become focused, much like clicking on the source node of the change impact region, while the false positives are left unfocused and easily distinguishable from the node and edges that do belong in that change impact region.

3.3.5 Node Dragging

When a user has found a class they are interested in, they can drag the class node to another area of the screen. This allows the user to take a class and the classes it is directly related to and drag them to an empty area. When a single class is dragged, the other classes are pulled along and re-organized into new positions based on the force directed layout algorithm [14]. In this way, these directly related class nodes remain together, and unrelated classes are pushed away. Prefuse has this functionality built into it.

3.3.6 Panning and Zooming

ImpactViz uses panning and zooming operations to deal with the space constraints of representing large software projects. Using the zooming technique, users can zoom into a particular area of the visualization that contains a particular subset of classes they are mostly interested in. This helps remove classes of lesser interest from their sight and allows the user to focus on the class clusters they are more interested in. With panning, the user can adjust where the centre of the screen is located, spatially

filtering classes that are not of interest and keeping the relevant class nodes near the center of the display. ImapctViz takes advantage of the built in capabilities of the Prefuse API to handle panning and zooming.

Combining panning and zooming allows a user to zoom out to get an overall view of the visualization and then zoom into an interesting portion of the visualization. The panning feature helps aid in adjusting the focus of the screen to the section that the user is currently interested in. These two features allow the user to take an exceptionally large visualization, and allow them adjust their focus on a particular subset of nodes.

On the graph control panel, we have also included an overview visualization. The user can feel free to pan and zoom throughout the graph and the overview visualization will help the user keep everything in context to the overall visualization. The user can click on the overview to move their focus from one area of the graph to another instantly. We also include the ability to let the user right click on the graph to have the camera zoom out just far enough to show the entire visualization in the main visualization panel.

3.3.7 Crossing the Gulf of Execution

Considering the gulf of execution from Norman's stages of action [30], we can aid the user by providing tools to help them quickly traverse this gulf. Doing so allows the user to easily make the software perform the operations they need to resolve their task goal, so that they can continue examining the visualization presented, crossing the gulf of evaluation to learn and understand more about their software system.

Once a user has formed a goal, whether it be to see the classes that a class depends on, the classes affected by a single change, move to another section of the system, or to bring to focus a set of changes that the user has an interest in, the user has created an intention (i.e., a short-term goal for what they want the system to do). From this intention, the user then forms a series of actions to bring the system closer to achieving the goal. All of the interaction in the system follows the guidelines for normal computer interaction with a mouse and keyboard. As such, the steps of planning the action sequence and executing the action sequence occur with little cognitive effort as they become more experienced with ImpactViz.

3.4 Debugging Scenario

Tracking down a software bug with ImpactViz is quick and easy, and can be broken into four fundamental steps. Suppose that in a video game project, whenever a *Level* class is created an error is appearing in the form of how items within the level are being displayed and represented on the map. Upon examining the *Level* class itself, all the source code appears to be working as intended. Therefore, the bug must be in a method from another class that is used by *Level*. That is, while the bug has manifested itself in *Level*, its source is another class upon which *Level* is dependent.

After loading the source code of the project and the SVN information into ImpactViz, a visual representation of the entire software project along with all of the revision changes is shown. From here, the software developer can see how the classes in the project are connected to one another, and if any particular changes have had a large impact on the overall system (i.e., changes to methods that are used by many

Figure 3.3: The initial view of ImpactViz after loading in the software project and repository information. The user can see how classes are connected, and which classes may be impacted by a change in another class as illustrated by the black nodes (classes that have changed) and their associated change impact regions.

different classes).

In the example shown in Figure 3.3, we can see that there were changes made to the following classes: *Main*, *Foe*, *Chest*, *Rogue*, *Dice*, and *Potion* (as illustrated by the black class nodes and their associated change impact regions). The other classes that may be affected by the changes can readily be identified within the change impact regions (e.g., the change to the *Chest* class may have an impact on the operations of *Level* and *Main*, as illustrated by the purple impact region that encapsulates these classes and the edge paths that all terminate at *Chest*).

During the debugging process, software developers often have some knowledge regarding when a bug was introduced. They may have test documentation that specifically states that everything related to the class in which the bug has manifested itself was working as intended under a specific scenario at some point in time. If the

Figure 3.4: By filtering the revision history data, the software developer is able to focus on the classes that have been changed and their impacts on other classes in the project.

software class is no longer working as intended, this can give the debugger a time-frame in which to search for the bug. Suppose that in this example the software developer had performed some testing on October 11, 2009 and found everything was working as intended. The revision filter can be set to start on *Revision 5 (October 11, 2009)* and end at *Revision 10 (October 16, 2009)*, the date at which the bug was first identified.

As shown in Figure 3.4, the outcome is an update to the visual representation of the project code which shows only the classes that were changed within the selected revisions, and their corresponding change impact regions. This revision history filtering automatically simplifies the debugging process by eliminating from consideration changes and the impacts of those changes that cannot be the source of the bug.

In the third step, the software developer can zoom into a set of change impact regions and select the class in which the bug has manifested itself to identify the

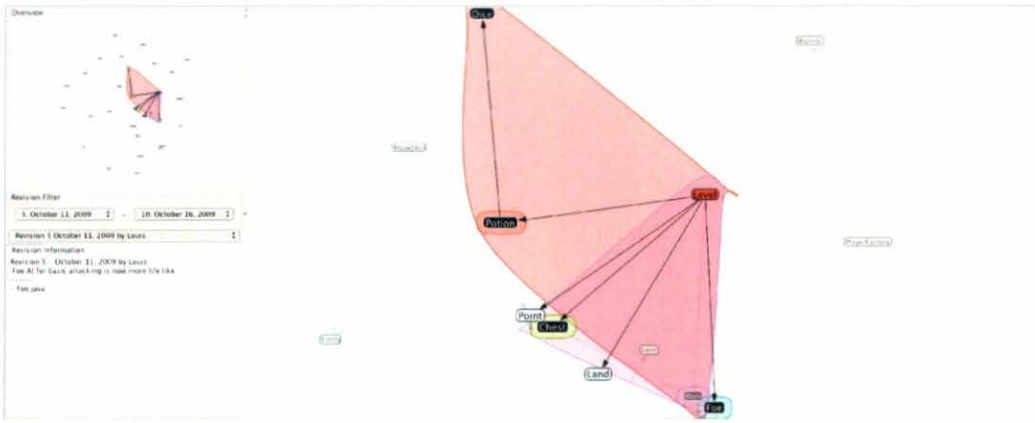

Figure 3.5: Zooming in and selecting the class in which the bug has manifested itself (*Level*) allows the software developer to quickly identify which classes have been changed that might contain the source of the bug (*Chest* and *Dice*). Note that *Foe* and *Potion* can be excluded since their change impact regions do not cover the *Level* class.

classes that may contain the source of the bug. In this example, as illustrated in Figure 3.5, the software developer has zoomed in and selected the *Level* class. This selection highlights the classes that *Level* relies upon at some point in the method call stack; other classes are shrunk and faded into the background. The classes that are most interesting from a debugging perspective are those that have been changed within the specified revision range and have some impact on *Level* (i.e., those that are shown in black nodes and have change impact regions that contain *Level*).

From the visual representation, the software developer can see that the classes that *Level* makes use of, and which of those have been modified within the revision history range. While *Chest*, *Foe*, *Potion*, and *Dice* have all been modified, the change impact regions of *Foe* and *Potion* do not cover *Level* indicating that those changes

were not made to methods which *Level* uses. As such, the remaining candidates for the source of the bug are those classes which were changed, and which also have change impact regions that cover the class in which the bug manifested itself: *Chest* and *Dice*.

In these three steps (load the source code and revision history, adjust the revision history range, and select the class in which the bug has manifested itself), software developers are able to visually identify the classes they should analyze in order to track down the source of the bug. The final step is to examine those classes in detail (using their regular software development toolkit) to locate and correct the bug.

3.5 Software Prototype Design

ImpactViz was designed to be used with software projects using SVN as a repository base and Java as the programming language, as these are the common tools used by Computer Science undergraduate students at Memorial University (who were used as the participants in evaluating ImpactViz, as discussed in Chapter 4). However, the prototype system was designed such that swapping between programming languages and different repository systems is relatively easy.

The Prefuse [18] framework was intended to be used from the start. Prefuse helps ImpactViz by providing the graph rendering and animations, while also providing the interactive tools of clicking, panning and zooming. The reason for using Prefuse framework was to help provide a starting point for the prototype. It allowed for generating the sophisticated graphs that ImpactViz requires, while allowing the developer the time to focus on the data collection required to run this project and the specific

tools that this project requires for the interface, which includes the ability to select a chain of nodes from an aggregate collection and from a node itself and the revision filter. Prefuse's role was to provide solutions to problems already solved in information visualization (graph generation and interactive techniques), which allowed the developer to focus on trying to find the answer to the research question - if a visual debugging tool would be an asset in the debugging process.

ImpactViz data collection is broken into two pieces: the pseudo compiler, and the repository handler. The repository handler is given the location of the software project and is responsible for reverting the code to earlier versions. The pseudo compiler is responsible for converting the source code into an object representation. The psuedo compiler achieves this by finding the method definitions and detecting the method calls that each method makes. The pseudo-compiler is also capable of taking two versions of the same method with different timestamps and identifying and recording if any changes have been made. These two modules combine to generate the data that is used to create the visual items in ImpactViz's interface.

3.5.1 Pseudo-Compiler

The pseudo-compiler is responsible for taking the source code and interpret the information in a way that is usable for ImpactViz's graph layout interface and change impact regions. ImpactViz requires four pieces of information for any given project: whether a class is a sub-class, all the method definitions, all compilable code, and all other methods to which the class methods makes calls. The pseudo-compiler is responsible for extracting this information for all class files. While the discussion in

this section is focused on Java-based projects, the general concepts also apply to a majority of other object-oriented programming languages; a similar pseudo-compiler could be written for other such languages.

Detecting the sub-class and method definitions is a simple matter, as all sub-classes are defined in the class definition, using the keyword *extends*, while method definitions are found in a block nested inside the class definition. Finding all compilable text is also a simple text processing activity, since all that is needed is to match the opening and closing blocks of the methods. In this process, comments are excluded from consideration, since they have no impact on the execution of the code and the generation of method call stacks.

Finding the methods each method depends on can be somewhat complicated. Once scans of all classes have been performed, with a record of every method in the project, we can then start to break down each method's source code to find other embedded method calls. Each line is examined for a variable definition. If the line does contain a variable definition, we record the variable and the type into a list, to look it up if the same variable is used at a later point in the code. Next, we examine the remaining line to see if a method definition is being called. Method calls are denoted by an opening and closing of the round brackets after alphanumeric text, without any symbols between the text and brackets. A method call made to an exterior class outside the current class, also contains a '.' to separate the method call and the variable name or class name (for static methods). Since some methods return classes or some classes have public variables, we can have multiple '.' (e.g., variable.method1().method2() and variable1.variable2.method()) to denote different layers of variables and method stacks. Thus, the method and variable detection

must be recursive. Once a method call has been identified, we must also look at the method call arguments. The Java language allows arguments in methods to also contain method calls, requiring that the text in each arguments be examined for method calls.

When the pseudo-compiler has finished this text processing, the end result generated is a complete listing for all the software classes in the project, with full details about their method definitions and the method calls made by each of these methods and the related classes. This information is used to see the full method stack that each class's methods can be a part of, which is used in the visualization portion of ImpactViz. It should be noted that this pseudo-compiler does not act as a proper compiler and cannot detect if certain blocks of codes are unreachable.

3.5.2 Repository Handler

The repository handler is responsible for being given the repository location (and any related usernames and passwords required to access it) and download a history of the project for detecting when each method in the project has been modified or changed.

The repository handler finds the latest version the project and downloads the information. Once the full code base has been downloaded, the repository handler passes the information to the pseudo-compiler to perform the method detection on it (we will denote this version as version r). When the full collection is finished processing, the repository then starts to roll back each class file to an earlier version (class version r-1) and has the pseudo-compiler break the older version into methods and related code and compare the two versions of the methods (class version r and

r-1), to see if a compilable change can be noticed. Note that in this step, only the software code is examined, not the method calls. If a change has been detected, the repository handler marks the change in its repository log of the class, method, and revision number. It then rolls the code back to an earlier version (class version r-2) and compares the two earlier versions (r-1 and r-2), continuing in this fashion until all the revision changes for the one class have been processed completely. This process is performed for all classes in the project.

Once the repository handler has processed all of the revisions of the project, it produces a complete listing of the method stack (provided by the pseudo-compiler) for all classes, and notations for which revisions each method was changed in the compilable code. This information is used in the ImpactViz visualization in the revision filter (to examine only changes that occur at a specified time period), which reflects both the colouring of the nodes (to denote a change to a class during the revision range) and the change impact regions.

Chapter 4

Evaluation Methodology

Since ImpactViz is a novel visualization tool for debugging, we cannot state that it is an actual improvement over other methods for debugging software projects unless it is properly evaluated. While the goal in the design of ImpactViz was to create a system that would be superior to traditional techniques of analyzing the source code of a project and inspecting repository logs (hence forth referred to as traditional debugging techniques), there is a need to confirm to what degree this goal has been achieved through user evaluations. For the purposes of this study, we define the traditional debugging method to be that a user can only inspect the original software code and can review the the repository logs for the software project. Users will not be running the software or making their own changes through out the study.

Evaluating information visualization systems is a tricky problem. Since such evaluations include human elements, care must be taken to ensure that correct question are asked and that the focus of the evaluation is placed on realistic tasks in order to avoid skewing the results [4]. Other challenges include introducing personal bias

with the participants and inconsistencies in the evaluation method [4]. These concern can be addressed by careful preparation of the experiments and creating a set of guidelines to follow during the experiment. There's also the challenge of selecting the right evaluation methods [33]. A short study will quickly give results, while a study performed over a longer period of time could show patterns that emerge once a user becomes an expert user of the system. A longitudinal study is much harder to arrange and finding participants who would be willing to participate may not be feasible in some situations.

The evaluation of ImpactViz follows the stepped model of evaluation and refinement, as proposed by Hoeber [21]. Inspections of the prototype system were conducted first, following Nielsen's heuristic evaluation [28], a cognitive walkthrough [29], and a visualization-specific inspection [45]. Based on these inspections, the prototype system was refined and improved. Then, two forms of user evaluations were performed. The first was a laboratory study, followed by field trials. Results from both these studies are presented and discussed in Chapter 5.

4.1 Inspection Methods

Using Nielson's heuristic evaluation [29], members of the User Experience Lab at Memorial University were asked to comment on ImpactViz. Issues regarding error prevention and recovery, aesthetics, and several others areas were discussed and problems were identified. After this activity, several changes were recommended, including adjusting the interface so that the controls were on the left side of the screen, adjusting the revision drop down to only allow users to select information within a selected

time period, and allowing users to hover over a class node and see the changes made to it over the selected time period.

A cognitive walkthrough was then conducted, placing the researchers in the position of typical users of the software system. The evaluators judge whether the actions required to accomplish the goal are intuitive and if the user would be provided with enough information to accomplish their task [29]. Based on the results of the cognitive walkthrough performed on ImpactViz, changes were made including modifying some of the terminology in the interface and removing the ability to change the effect of clicking on a class node in the interface.

As well to the above inspections, we also analyzed the design and visual representations of ImpactViz following visualization-specific guidelines [45]. In particular, we analyzed colour usage to ensure that the colours were decodable and were used consistently throughout the interface. In ImpactViz, black and white are only used to represent whether a class has undergone a recent change or remained unchanged, while other colours are used to show the change impact regions. Although there is the possibility for the repetition of colour when there are many change impact regions, the system ensures that such regions are displayed in disjoint space whenever possible.

4.2 Laboratory Study

The laboratory study followed a 2 x 2 (debugging technique x software projects) mixed design. The debugging technique (traditional vs. ImpactViz) was assigned to participants as a within-subjects variable; the two separate software projects were

assigned between-subjects. The participants were divided into two groups, which allowed the order of exposure to the debugging technique to be varied. Both groups used traditional debugging techniques to debug their respective first software project, and then ImpactViz to debug the second software project. Since the two projects are completely unrelated, and the participants were selected such that they were already familiar with traditional debugging techniques (i.e., senior undergraduate computer science students), the learning effects were minimized. The participants used the traditional debugging technique first since this is a method that they were all already familiar with.

With the laboratory study, the goal is to determine if the different debugging interfaces have an influence on the participants' ability to perform the debugging task. With this in mind, the dependant variables are the time to find the bug, whether the answer given is correct, the ease of use and usefulness rating the participant, and whether the user prefers using ImpactViz for debugging in comparison to the traditional method.

4.2.1 Hypotheses

There are five main hypothesis that were tested during this laboratory study. Since the goal is to compare ImpactViz to the traditional debugging method that they are already familiar with, we consider this traditional method the baseline in this comparison (herein referred to as the Baseline system).

The first hypothesis is that participants who use ImpactViz to debug will be faster in completing their task than those using the Baseline system. We expect this result

since ImpactViz was designed to help users in traversing the method call stack in a software system in a visual manner, which should reduce the number of classes that need to be inspected for the source of the bug.

H1: Participants will be able to identify the source of bugs faster with ImpactViz than the Baseline.

The second hypothesis is that more participants using ImpactViz will find the source of a bug than the participants using the Baseline system. ImpactViz helps users to narrow down a particular subset of classes to inspect. With a smaller set of classes to examine, this should lead to less confusion and information overload, allowing the participants to inspect the correct class.

H2: Participants will be more accurate when finding the source of bugs with ImpactViz than with the Baseline.

The third hypotheses is that participants will find ImpactViz more useful and easy to use that the Baseline. ImpactViz is a visual debugging tool, which should make it easy to see how classes interact with one another, and help narrow down classes to inspect in three short steps. One of the goals in the design of the system was to make it both useful and easy to use.

H3: Participants will respond positively to questions regarding the usefulness and ease of use of ImpactVis.

The fourth hypothesis is that participants will prefer to use ImpactViz for most of their debugging tasks rather than the Baseline technique. Although the efficiency

and effectiveness of a system are not always tied directly to user preference, this hypothesis predicts that the participants' subjective opinions will be in favour of using ImpactViz.

H4: For the task of supporting debugging activities, participants will rank ImpactViz as a preferable to the Baseline.

The fifth hypothesis is that users will prefer the Baseline system for understanding the source code of the project over ImpactViz. While ImpactViz provides information on how the classes in the project interact with one another, the current implementation is not linked directly with a software development environment and does not show the source code of the classes. Further, it offers no aid in understanding the lines of code that make up the class methods.

H5: For the task of understanding software code in general, participants will rank the Baseline as preferable to ImpactViz.

4.2.2 Measurements

In order to evaluate our hypotheses, we need to measure the participants performance under the various experimental conditions. The first two hypotheses are simple to measure, as they are time and accuracy. The last three hypotheses, however, are trickier, as they are subjective questions and require more care to receive an accurate measurement on them.

For the first empirical measurement, time, we measure the amount of time it takes a user to give us their final answer and will be used for the validation of H1.

The second empirical measurement, accuracy, is measured by grading the answer the participant gives. In debugging you either find the bug or you do not find the bug. With this in mind, we grade the participants on a binary scale.

The last three hypotheses are subjective. In order to receive an accurate measurement, we must carefully ask the participant questions about what they think of ImpactViz. Using the Technology Acceptance Model (TAM) [8], we carefully word our questions specifically to see how useful and easy to use participants think ImpactViz is for H3. For H4 and H5, we ask the participants to rank ImpactViz and the Baseline on their ability to perform standard software tasks and which of the two debugging methods they preferred.

4.2.3 Statistics Test

In order to validate that there exists a significant difference between the two debugging systems, we need to use statistical tests to compare the two sets of data. H1, H2, H3 and H4 all require statistical tests, detailed in the following paragraphs. H3 does not require any statical analysis, since we are not comparing the results gathered from the TAM questionnaire to the Baseline method.

For H1, we will be using the analysis of variance (ANOVA) [37]. ANOVA is used to determine the probability that the mean value of groups of data are equal. We use the ANOVA test to see if the samples from both test are equal. If we can successfully find that the probability of the two sets being the same is insignificant (p ¡ 0.05), we can say with confidence that the two sets are not equal. From this information, we can then compare the different mean values of the sets and see whether the the test

involving the time spent using ImpactViz has a lower value, proving that ImpactViz does indeed speed up the debugging process.

The data collected for H2 requires us to determine the frequencies in which bugs are found between the two debugging methods, the chi-squared test [37] will be used. Since each test only has one independent variable (ImapctViz or Baseline method for debugging), with only two possible results (bug found or not found), the chi-square contingency table will be used to determine if the sets of data are significantly different. frequencies, the Chi-square test would be a good fit. By creating a 2 x 2 box (debugging methods x number of possible results), we can place all of the possible results from the accuracy test (bug found or not found). We can now apply the chi-square test to see if there's a significant difference between the two frequencies.

H4 and H5 require to ask participants to choose preference between ImpactViz and the Baseline method. For this comparison, we will be performing the Wilcoxon-signed rank test [37]. The Wilcoxon signed rank test is used to compare paired data that does not rely on assumptions that the data has distribution model (non-parametric). The Wilcoxon signed test can be used since we cannot confirm if the data is parametric (following a distribution model like normal distribution). Wilcoxon signed rank test is used to pair data from two separate groups and examine the magnitude of the differences, to detect if the difference between the two sets is significant. By finding if the difference between the two sets is significant (p ¿ .95), we can then find which debugging method provides the more successful rate of finding bugs, to help provide validate our hypothesis that ImpactViz helps users be more accurate in their bug tracking.

4.2.4 Software Projects

Two software projects will be used in the laboratory study for the participants to debug. The first project, titled "Game" was developed by a single developer over the course of several months. The second project, "Catering" is a modified group project developed by four individuals over the course of four months. The second project was trimmed down in size to match the same size and complexity as the first project. Both projects have around 35 software classes each, with no more then six method calls to different software classes.

4.2.5 Participant Recruitment

The participants in this study were third and fourth year undergraduate and first year Master's students from the Department of Computer Science at Memorial University. These participants were targeted because of their experience in working on group projects; they would have all used a repository system at this point in their academic and professional careers for various group projects. With these experiences, the participants would understand the situation of debugging other people's code and likely would have encountered debugging situations similar to the ones outlined in the tasks. They would understand the frustrations and difficulty that occur when a bug is accidentally introduced into the repository that is difficult to track down.

4.2.6 Study Procedures

When participants arrived for the evaluation, they were thanked for volunteering their time to this study. They were given a short introduction about the laboratory study

and what problems this research is trying to address. Then they were given a brief outline of how the study will progress. They were given two consent forms to sign; one copy was kept by the researchers for proof of consent and the other copy was given to the participant, in case they had any further questions or concerns about the study.

After the consent form had been collected and secured, the participants were given a pre-task questionnaire, to gauge their prior experiences and background. Questions on this form include the number of software projects they had worked on using a repository system, how well they gauged themselves in knowing repository systems, self-evaluated debugging skills, and preferences for programming language and operating systems (see Appendix A). The last two questions are important, as the study was performed on an Apple iMac running OS X and the software code they examined was written in Java. If a user was very unfamiliar with the OS X operating system or didn't understand Java very well, it could lead to a negative bias in the measurements that is unrelated to the debugging tasks.

After the questionnaire had been completed, the researcher briefly went over the traditional debugging technique and how to follow the method call stack in the Eclipse Java development environment (which is a popular Jave IDE used by the undergraduate students). The researcher also provided an explanation on how a single bug in any part of the method call stack can affect the result given to the initial call and how quickly the method call stack can grow.

When the participant acknowledged a sufficient understanding, they were given their first task information sheet (see Appendix A), and informed the researcher when they would like to start the debugging task. This first task was performed using the

traditional debugging techniques; half the participants were assigned Project 1, The Game, first; the other half used Project 2, Catering, first. more information about the tasks themselves can be found in section that follows. When they started the debugging work, the researcher started the timer and video recording devices. When the participant completed the task, the timer was stopped and their final answer and time to task completion was recorded on a task evaluation sheet.

As the user finished the first task, the researcher readied the system for a second demonstration, this time involving ImpactViz. A tutorial was provided explaining what the visualizations represent, the interaction techniques, and how the system reacts to certain inputs. This tutorial was performed on the same sample code as used in the demonstration for the traditional debugging technique, to help show how ImpactViz is different from the traditional debugging technique.

When the tutorial was finished, the participants were given time to read the second task information sheet. After the participant acknowledged their understanding of the task, they were permitted to begin the activity. At this time, the timer was started and the video recording device was turned on. When the user finished the task, the timer and video recording devices were stopped, and their final answer and time was recorded. The participants used Ecliplse to explore code sections in the same fashion as they did in debugging the traditional task.

Once both tasks were been completed, the participant was given a post-task questionnaire (see Appendix A). This questionnaire was provided to evaluate how useful and easy to use the participant found ImpactViz in comparison traditional debugging techniques (details related to how the questions were formed for this questionnaire can be found below).

With the consent forms signed, pre-task questionnaire filled in, both tasks completed, and the post-task questionnaire answered, the study was completed. The participant was thanked for their time, and given compensation in the form of $10.

4.2.7 Study Tasks

Each task had the participant debug a software project. Participants were allowed to examine the source code and use repository information about when changes were made, who made them, and what files were changed. Participants were not allowed to modify or run the code to identify the bug, but were provided with sufficient information about the bug to be able to track it down. Participants were given a sheet of paper outlining the details of the bug, including the class that the bug was manifesting itself in, and the date when the class was last tested and identified as working as intended (also known as the bug free state).

With the information provided above, the participants were expected to traverse the method call stack starting with the class in which the bug was manifesting itself. In each method, the participant was to analyze the software code for any erroneous code that could cause the bug described. Since participants were not familiar with the software project, they were allowed to ask the researcher questions about what a particular section of code did and its influence on the method. The bugs were designed to be noticeable to someone who had no prior experience with the specific requirements of the evaluation projects. When the participant came to an answer they were confident in, they could inform the researcher and conclude the task. In order to avoid unnecessary participant frustration, after 20 minutes had passed partic-

ipants were given the opportunity to abandon the particular task. If the participant abandoned the task, the time would be recorded as undetermined and the grade for their final answer as 0, as they were unable to find the source bug method and class.

4.2.8 Subjective Questions

The post-study questionnaire (see Appendix A) included questions from the Technology Acceptance Model (TAM) [8] in order to gauge perceived usefulness and ease of use. In addition, participants were asked to indicate their impressions of specific features of ImpactViz. For both sets of questions, a five-point Likert scale was used.

Participants were asked which debugging method they would prefer to use in future debugging tasks. These tasks included understanding software code, the method call stack, revision history, the overall effect of some software changes, and their preferred system in future debugging tasks.

4.3 Field Trials

While the laboratory study was conducted under a controlled environment, the field trials attempt to study how the software would be used in a real world environment. In the laboratory study, we provided artificial tasks and an environment for the participants to work in. During the field trial, participants provided their own software project and questions that they had about their software. While performing this study, there were only a few participants, but their actions were closely monitored and examined. The documents relating to the the field trial can be seen in Appendix B.

Benefits of using a field trial as an evaluation method includes allowing the observers see how users of the software will actually use the tool. While we may think that we have a good understanding of how our software is being used, field trials allow us to observe and make notes about how the software is actually being used [33].

4.3.1 Study Questions and Measurements

For this study, our main research question was to determine whether participants would find ImpactViz beneficial to solving their own debugging problems and understanding how the pieces of their own projects are related to one another. There are no quantitative measurements for the field trial evaluations, as it is not reasonable to make time and accuracy comparisons between different projects. Of more value are the qualitative impressions of the system and its value.

From the field trials, we wish to measure how well accepted our software will be in a real-world environment. We use the same TAM questions as in the laboratory study to measure usefulness and ease of use. As well, we record comments and actions performed by our participants and study them afterwards to see how successful ImpactViz was in aiding the participants with their own software systems. Post-study analysis can also answer the question of when ImpactViz will be useful and under what circumstance would a software developer benefit from using ImpactViz to help them debug code.

4.3.2 Participant Groups

Two of participants within the field trials were selected from among students at Memorial University who were enrolled in COMP 4770 (Team Project) in the Winter semester of 2010. These students were finishing their Computer Science degrees, with COMP 4770 being the last mandatory course required for the B.Sc. degree. During the previous years, the participants would have used various team management skills and code management software to help them complete their group projects. These are individuals who have a lot of experience in software repositories and have spent much of their time crafting their projects to match associated UML diagrams.

A second group of participants for the field trails were two very experienced software developers. They were both post-doctoral researchers working within the Department of Computer Science. They both had significant experience in using software repositories, and were working together on a large software project during the course of the study. Each field trial participant was compensated for their time in the form of $20.

4.3.3 Subjective Questions

Similar to the subjective questions asked at the end of the laboratory study in the post-task questionnaire, the field trial members were asked questions related to ImpactViz, its debugging capabilities, its use as a software exploration tool, and questions from TAM model. The participants were also asked to grade different features of ImpactViz, so that the researchers could observe what parts of ImpactViz the participants found to be the most useful.

Unlike the questionnaire presented at the end of the laboratory study, the participants were not asked to perform any comparison between ImpactViz and another system. Since the participants are not using another system under the same constraints, there is no information to be gained in asking this question to such a small sample of participants.

Chapter 5

Results

In this Chapter, the results for the evaluations described in the previous Chapter will be be presented and discussed, with statistical analysis to determine how successful the prototype was in satisfying the hypotheses, where appropriate. The results of the laboratory study will be presented first, followed by the field trials.

5.1 Laboratory Study

For the field laboratory study, a total of 16 students were recruited. 12 of the participants were undergraduate students, while the remaining four students were first year Master's students. The 16 participants were broken into two groups consisting of six undergraduate and two Master's students in each group. The raw quantitative results can be seen in the four tables below in their respective sections (Table 5.1, Table 5.2, Table 5.4, and Table 5.5).

The hypotheses referenced here can be found in Section 4.2.1. As a reminder, users were given the opportunity to abandon the task after 20 minutes (1200 seconds), after

showing visible signs of frustration. Participants who did choose to abandon their task have been noted in the tables.

This laboratory study was performed as a comparative study. We asked participants to use two systems, the Baseline system, which is the traditional debugging system that is in current use by most programmers at the senior undergraduate student level, and ImpactViz. The results are compared on this basis for empirical metrics measured in both projects and the participants' opinions after using both systems for the debugging tasks.

While analyzing the results from the laboratory study, we will be analyzing each project separately. Since both project are unrelated to one another, a direct comparison between the two in terms of time and accuracy would not be a fair comparison. However, comparing between interfaces allows us to see how the participants' performances changes when performing the same task, but using different interfaces.

5.1.1 Project 1 - The Game

The results presented here are related to the programming project titled "The Game". "The Game" is an computer game, with randomly generated maps and multiple level dungeons. The player can choose a character and move the character through the dungeons, defeating monsters and collecting treasure. While testing the game, a bug was discovered in which health potions, an item that help recover the player's health, were healing for a static amount instead of the intended random amount.

Table 5.1: The laboratory study results of participants debugging Project 1, The Game, using the Baseline.

Participant ID	Time (seconds)	Accuracy	Comment
1	421	0%	
2	491	100%	
3	1644	100%	
4	397	100%	
5	323	100%	
6	1245	0%	Abandoned
7	230	100%	Master's Student
8	490	100%	Master's Student

Table 5.2: The laboratory study results of participants debugging Project 1, The Game, using ImpactViz.

Participant ID	Time (seconds)	Accuracy	Comment
9	520	100%	
10	179	100%	
11	199	100%	
12	154	100%	
13	132	100%	
14	177	100%	
15	133	100%	Master's Student
16	357	100%	Master's Student

5.1.1.1 Time to Task Completion

In H1, we predicted that participants would be able to identify the source of the bug faster using ImpactViz than the Baseline. In order to verify this, we measured the time to task completion using both debugging methods. Table 5.1 shows the time to task completion data collected for participants debugging the project using the Baseline methods, while Table 5.2 shows the data for using ImpactViz. We can see that one participant abandoned the task using the Baseline. Meanwhile, under ImpactViz, no participants abandoned the tasks.

Two outliers were removed from the participants who used the baseline method. Participant 6 abandoned the task, meaning a final task completion time could not be found, while participant 3 got lost while navigating the software code resulting in a time to task completion that was nearly 2.5 times higher than average time achieved by the other participants to complete the same task with the same interface. The standard deviation of this time in relation to the other times was approximately 2, placing this participant's time as an outlier. No outliers were identified from among the participants that used ImpactViz.

An ANOVA test was performed on the time to task completion measurements, resulting in a validation of the statistical significance of the differences ($F(1, 14) = 5.811, p < 0.05$). Since $p < 0.05$, the difference between both sets of data is statistically significant. The average time taken to find the bug using the Baseline was 392 seconds, while the average time using ImpactViz was 231 seconds. Using ImpactViz, participants were able to identify the source of the bug 60% quicker than the baseline. As a result, we conclude that the data validates H1.

5.1.1.2 Accuracy

H2 predicted that participants would be more accurate in finding the source of the bug using ImpactViz. Accuracy was measured based on whether participants correctly identified the class and method in which the bug existed. 100% of the participants correctly identified the bug using ImpactViz, while 75% of the participants found the bug using the Baseline. A chi-squared test shows that these results are not statistically significant ($\chi^2 = 0.46$). Therefore, we can conclude that H2 is not validated by the data, although there is evidence implying that there is a positive co-relation in the usage of ImpactViz and an increase in debugging accuracy.

5.1.1.3 Usefulness and Ease of Use

As stated in H3, we expected that participants would respond positively to statements regarding the usefulness and ease of use of ImpactViz. The TAM guided the development of the questions for measuring the participants subjective reactions to the system. Six statements were prepared that delved into the issues of usefulness, and another six addressed issues related to the ease of use. Measurements were made on a on five-point Likert scale.

The TAM statements related the participants' use of the system under investigation (i.e., ImpactViz) to existing practice. As such, the comparison to the Baseline is inherent within the responses.

The frequency of responses were aggregated for each set of six questions. As such, there are 48 measurements for each of usefulness and ease of use. These results are presented in Figure 5.1.1.3. Clearly, there are consistently positive responses for both

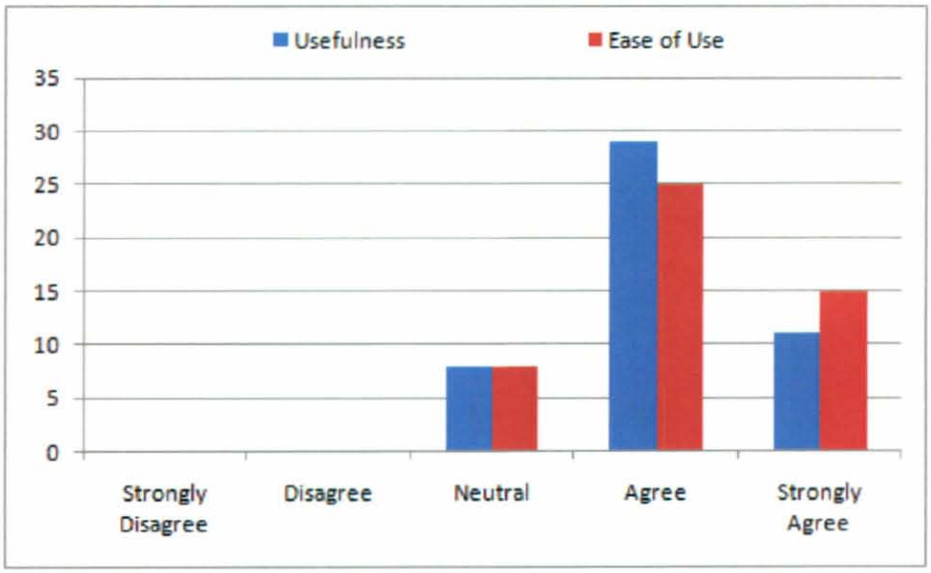

Figure 5.1: Aggregate responses for the TAM statements related to the usefulness and ease of use of ImpactViz by participants who used ImpactViz on Project 1, The Game.

measurements, which supports H3.

5.1.1.4 Supporting Debugging Activities

H4 predicted that participants would rank ImpactViz as preferable to the Baseline for supporting debugging activities. Four questions were asked in the post-study questionnaire that addressed specific and general debugging tasks. These are outlined in Table 5.3, along with the raw results for all participants in the study, and the outcome of Wilcoxon signed rank tests.

For three of the four questions users greatly preferred ImpactViz over the Baseline method, with the results being statistically significant. However, for the task of

Table 5.3: Frequency of rank preference for debugging activities for ImpactViz for Project 1, The Game.

Debugging task	Baseline	ImpactViz	Wilcoxon signed rank test
Understand the method call stack	2	6	$Z = -1.00, p > 0.05$
Understand the revision history	1	7	$Z = -3.00, p < 0.01$
Understand the effect of changes	1	7	$Z = -3.00, p < 0.01$
Preferred system for debugging	1	7	$Z = -3.00, p < 0.01$

understanding the method call stack, the opinions of the participants did not show a statistically significant difference between ImpactViz and the Baseline. This could be explained by how quickly participants found the answer (in under 2.5 minutes), that they never really had the chance to learn the method call stack. Since three of the four questions that are related to supporting debugging activities showed a statistically significant preference for using ImpactViz, we conclude that H4 is strongly supported, but not uniformly validated.

5.1.1.5 Understanding Software Code

In H5, we predicted that participants would rank the Baseline as preferable to ImpactViz for the task of understanding the software code. Our expectation was that participants would find the ability to directly access and browse the software code a valuable tool for understanding. By contrast, since ImpactViz provides only an overview of the method call stack, the resulting relationships between classes, and a visual representation of the impact of changes, understanding the software code would be more difficult.

In the post-study questionnaire, participants were asked to rank their preference of methods for understanding the software code. The participants were split in their preference, with four participants preferring ImpactViz and four preferring Baseline. A Wilcoxon signed rank test showed that this difference is not statistically significant ($Z = -0.00, p = 0.500$). As such, we conclude that H5 is not supported by the data.

5.1.1.6 Video Analysis

The usage of the video recorded during the laboratory tests were used to confirm that there was no reason to exclude any participant from the accuracy statistics for improper debugging usage for either interface.

5.1.2 Project 2 - Catering

The results presented here are related to the programming project titled "Catering". "Catering" is an online web service which allowed customers to access a caterer and make requests for catering services. Customers were able to browse menus, make orders, as well review order histories. A bug was detected when users were reviewing old orders; the items in the menu lead to either the wrong item or to a page, informing them the item does not exist.

5.1.2.1 Time to Task Completion

In H1, we predicted that participants would be able to identify the source of the bug faster using ImpactViz than the Baseline. In order to verify this, we measured the time to task completion using both debugging methods. Table 5.4 shows the data collected for participants who used the Baseline for debugging, while Table 5.5 shows

Table 5.4: The laboratory study results of participants debugging Project 2, Catering, using the Baseline.

Participant ID	Time (seconds)	Accuracy	Comment
9	1390	0%	Abandoned
10	548	0%	
11	1174	100%	
12	771	0%	
13	1464	0%	Abandoned
14	1184	0%	
15	306	0%	Master's Student
16	1025	0%	Master's Student

Table 5.5: The laboratory study results of participants debugging Project 2, Catering, using ImpactViz.

Participant ID	Time (seconds)	Accuracy	Comment
1	999	0%	
2	1200	0%	
3	768	100%	
4	912	100%	
5	345	100%	
6	666	0%	
7	826	0%	Master's Student
8	696	0%	Master's Student

the data for participants who used ImpactViz on the same project. Similar to the first project, two participants abandoned the task while using the Baseline, while zero participants abandoned the task while using ImpactViz.

One outlier was identified during this project. Participant 7 using ImpactViz refused to use ImpactViz for the study, instead quickly bypassing the visualization to analyze the source code of the project, performing the task in a style very similar to the baseline debugging technique. Participants 1 and 5 from the Baseline group were removed, due to their abandonment of the task after 20 minutes of debugging.

The ANOVA test was performed on the time to task completion measurements. The result shows no significance between the values ($F(1, 13) = 0.044, p > 0.05$) for this sample size. The average time taken to find the bug using the Baseline method was 834 seconds, while the average time using ImpactViz was 798 seconds. Participants using ImpactViz found the bug in 95.7% of the time then the participants using Baseline. While overall we can see a minor trend in favour of ImpactViz, the results are not significant, and do not verify H1.

5.1.2.2 Accuracy

H2 predicted that participants would be more accurate in finding the source of the bug using ImpactViz. Accuracy was measured based on whether participants correctly identified the class and method in which the bug existed. Participant 7 of the ImpactViz group was removed from this analysis, due to refusing to use the ImpactViz tool during the evaluation. 37.5% of the participants correctly identified the bug using ImpactViz, while no participant identified the proper bug using the Baseline method. A chi-squared test shows that these results are not statistically significant

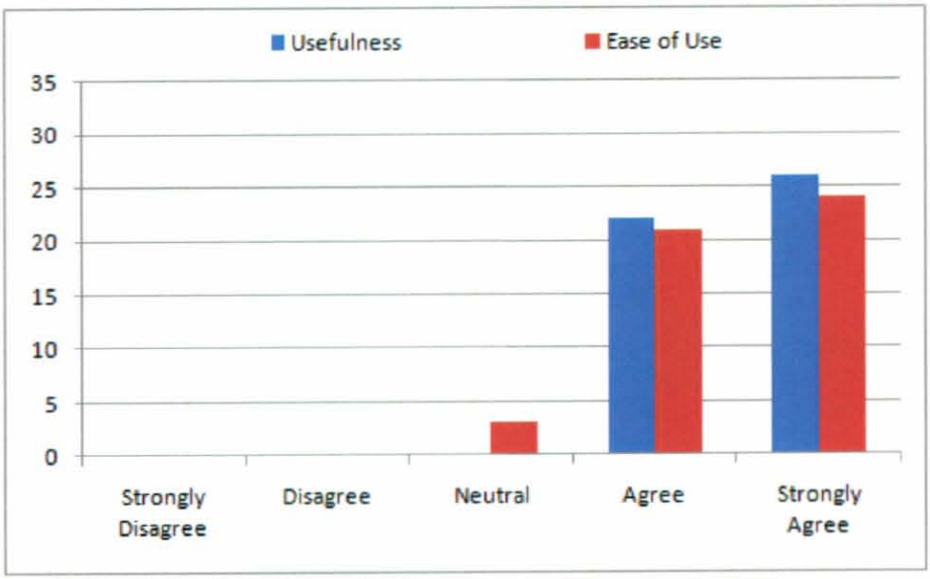

Figure 5.2: Aggregate responses of the TAM statements related to the usefulness and ease of use of ImpactViz by participants who used ImpactViz on Project 2, Catering.

($\chi^2 = 0.0769$). As such, we conclude that H2 is not verified by the data, but there is a positive trend being shown in favour of ImpactViz.

5.1.2.3 Usefulness and Ease of Use

As stated in H3, we expected that participants would respond positively to statements regarding the usefulness and ease of use of ImpactViz compared to the Baseline method. Results from the TAM questions were aggregated, resulting in a total of 48 measures for each of usefulness and ease of use. These results are presented in Figure 5.2. Clearly, there are consistent positive responses for both measurements, which supports H3.

Table 5.6: Frequency of rank preference for debugging activities for ImpactViz for Project 2, Catering.

Debugging task	Baseline	ImpactViz	Wilcoxon signed rank test
Understand the method call stack	0	8	$Z = -3.00, p < 0.01$
Understand the revision history	0	8	$Z = -3.00, p < 0.01$
Understand the effect of changes	0	8	$Z = -3.00, p < 0.01$
Preferred system for debugging	0	8	$Z = -3.00, p < 0.01$

5.1.2.4 Supporting Debugging Activities

H4 predicted that participants would rank ImpactViz as preferable to the Baseline for supporting debugging activities. Four questions were asked in the post-study questionnaire that addressed specific and general debugging tasks. These questions are outlined in Table 5.6, along with the results given from the individuals who used ImpactViz for debugging on the second project.

Every participant who used ImpactViz on the second project preferred to continue using ImpactViz for future software debugging tasks. The results are statically significant, as presented in Table 5.6, and support H4.

5.1.2.5 Understanding Software Code

In regards to H5, we hypothesized there would be little preference for participants to use ImpactViz for understanding the software code of a project. Of the participants who debugged Project 2 using ImpactViz, five preferred ImpactViz for understanding the software code versus three who preferred Baseline. A Wilcoxon signed rank test showed that this difference is not statistically significant ($Z = -1.00, p = 0.159$). As

such, we conclude that H5 is not supported.

5.1.2.6 Video Analysis

Through studying the video recordings for the participants during the debugging of the second project, participant 7 was removed from the analysis in both the time to task completion and accuracy evaluations. Participant was removed due to improper use of the ImpactViz debugging task, by refusing to user the graphical tool, and, instead, prefer to complete the full task by only inspecting the software's source code.

5.1.3 Summary of Laboratory Study Results

Comparing the data between the two projects, it quickly became apparent that participants had a much harder time finding the bug in Project 2, Catering, compared to Project 1, The Game. There were more abandoned tasks, less correct answers for the bugs location, and higher overall times. Although both test projects were designed to be of similar difficulty (i.e., similar number of classes, similar level of class inter-dependency, similar distance between the manifestation of the bug and its source), participants had a much more difficult time conceptualizing the design and class interaction within Project 2. Further, the source of the bug in this project was in a class constructor; although most participants correctly identified the class as a potential source of the bug, few inspected the class constructor method. Based on this information and the fact that users had a much harder time in understanding the code, even with assistance, we feel this project does not properly represent a normal debugging situation.

From the five hypothesis that guided the design of the laboratory study, two

hypothesis were confirmed (H3 and H4). One hypothesis achieved mixed results (H1) depending on if the participant used ImpactViz under a project of normal task difficulty (Project 1) or an abnormal task difficulty (Project 2). Two hypothesis was not confirmed (H2 and H5), although we saw a large change in probability results in H2 from Project 1 to Project 2. From these results, we can see that participants find ImpactViz as much more preferred method for debugging, and can help decrease debugging speed and increase debugging accuracy under normal situations.

5.2 Field Trials

The field trial participants were divided into two groups. The first group consisted of two students from the COMP 4770 (Team Project) undergraduate course who worked on a Java server application, while the second group contained two post-doctoral researchers working on a geo-visualization project.

5.2.1 Group 1 - Undergraduate Students

The first group's project had a total of 13 students developing the software over four months. The entirety of the project was developed from scratch. In the end, 37 class files were created and modified, with over three hundred revisions made. The two participants in this group used ImpactViz after their project was finished. As such, these two individuals used the prototype software as a retrospective look at their team project.

Both participants came into the study with opposite opinions on the project. The first member felt very pleased with the overall project. Meanwhile, the second

participant was far more dissatisfied with the project, even as far as saying in his recruitment e-mail that "the course material is complete (sort of)...", showing that, while the course was finished, he still felt there was a lot of work left unfinished.

Both individuals, while in the development phase of the project, focused their efforts in separate areas of the project. Both participants reported that they felt they were better informed via ImpactViz of how other classes outside their expertise areas worked and was designed. The first participant was happy with what he saw, stating that it matched their initial UML designs for the project, developed early in the course. While the second participant was less pleased about the relations, stating that "it doesn't make much sense." He observed inconsistent names for classes that had common functions, and that these common classes were in completely separate areas in the visualization with no connections. The second participant then started to discuss how he would re-work the design of the system, to help make the system make more sense.

Both participants said they found the the visual representation of the method call stack in ImpactViz very helpful in understanding the flow of method calls. Both also said they could use the visualization to illustrate the quality (or lack of quality) in how the system was designed. The first participant felt very strongly that the project was well designed and their execution of the design was very accurate. The second participant was able to use the visualization to help show flaws in the design and use the visual tools to help illustrate where new classes should be and where class relations don't quite make sense. Both individuals also found older classes that still remained in their repository that no longer had any place in the system, as these classes had no relations to any of the other classes. Both had thought this group of

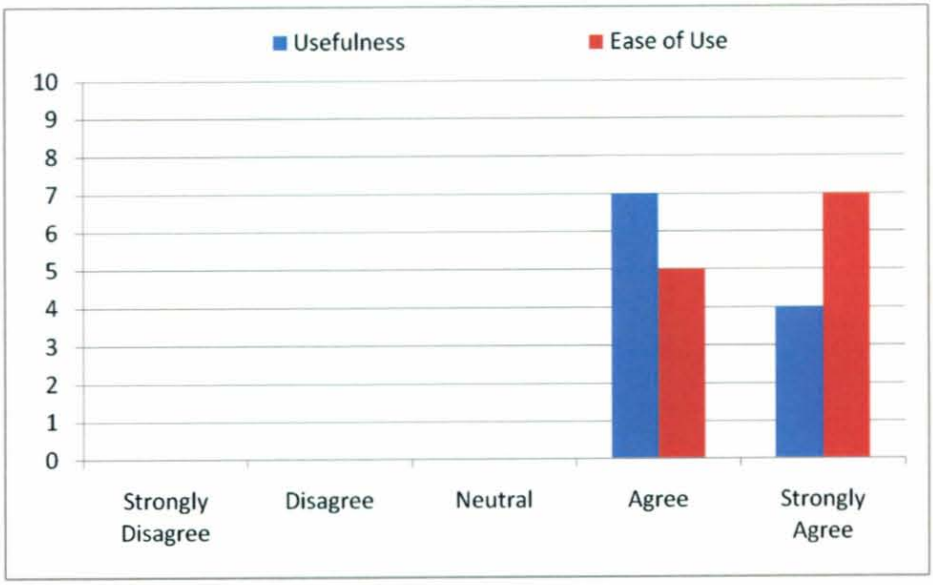

Figure 5.3: Responses to the TAM statements related to the usefulness and ease of use of ImpactViz by participants in the first group of the field trial.

class had already been removed, but they were clearly visible in the visualization.

The participants from this group responded very positively when they answered the TAM questions. As seen in Figure 5.3, all the answers to both sets of questions regarding ease of use and usefulness as either "Agree" or "Strongly Agree." The results of the TAM questions for this group shows that this group found ImpactViz to be both useful and easy to use for their software development needs.

This group found ImpactViz to be very helpful in code exploration. It helped both individuals see the overall project, outside of the small scope they had been focusing on. While the participants attempted to debug their project using ImpactViz by trying to track down older bugs, they had difficulties in evaluating whether their efforts were worthwhile due to the fact that the project was complete and there

were no significant bugs to be found. After the field trial was complete, one of the participants asked when a Python version would be released, which he could use for his job. This suggests that developers may be interested in using ImpactViz to help them debug software projects in a real world setting.

5.2.2 Group 2 - Post-Doctoral Researchers

The second group of participants were working jointly on a large geo-visualization project. The total size of the team was two members, with both members consenting to participate in the field trial. The project contained 37 class files, with 70+ revisions made over the course of the project. At the time of the study, the project development was still ongoing and had been in development for four months. The developing environment was an office in which both team members worked side-by-side and were in constant communication.

The first team member in this group was much more versed in the usage of repositories, often only changing one or two files before committing his changes. The second member was less familiar with using software repositories system and made less frequent commitments that affected many more files.

While the two-person team found ImapctViz innovative, they found little use for the prototype software. Both members were very aware of the software system and kept in constant contact during the development of the software. As such, they gained very little new insight about their software system. When inquired about any debugging tasks they wish to perform, neither member had any specific debugging tasks for which they could evaluate the system.

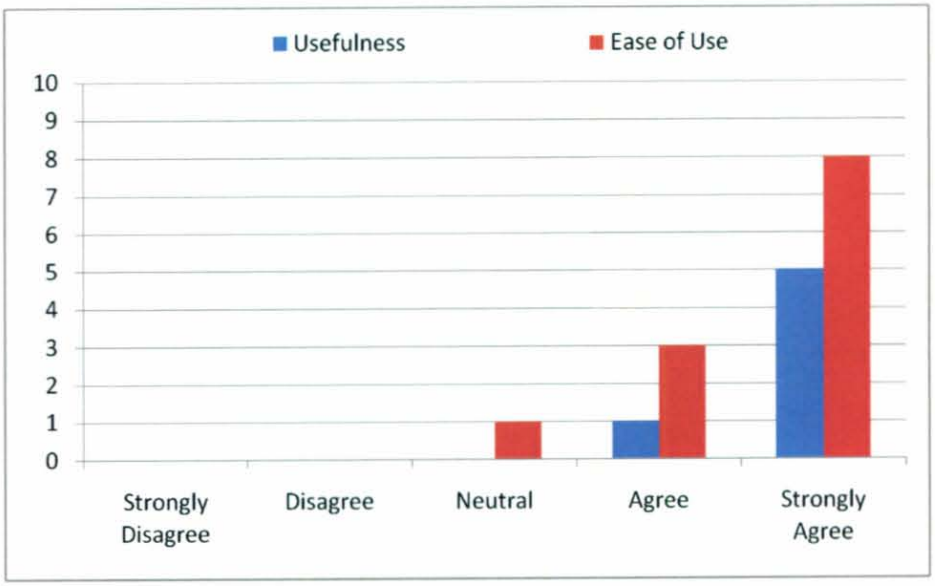

Figure 5.4: Responses to the TAM statements related to the usefulness and ease of use of ImpactViz by participants in the second group of the field trial

Participants in this group were less accepting of the prototype, as can be seen in the TAM results in Figure 5.4. While both members of this team found the technology easy to use, one member had applied "n/a" to all 6 of the "Usefulness" questions, hence the low number of total usefulness results in the figure. This shows that in a small team environment, ImpactViz may not be as useful as in a larger team setting. This may be due to the fact that small teams are able to communicate with one another while developing and debugging software more effectively than large teams.

Chapter 6

Discussion

After performing evaluations on the prototype system and presenting the results in the previous Chapter, it is now important to interpret the results from the laboratory studies and field trials. In this Chapter, we discuss how the results relate to real world debugging activities.

6.1 Laboratory Study Results

While designing the laboratory study in Chapter 4, there were five hypotheses that we wanted to test in relation to the traditional debugging techniques. The first two hypothesis required quantitative measurements to gauge the participants' debugging speed and accuracy in tracking down the source of the bug. The next three hypotheses were qualitative and subjective to each participant's experiences. The following two subsections will discuss these hypothesis.

6.1.1 Quantitative Hypotheses

From the two projects that were used in the laboratory study, we have contrasting qualities. In Project 1, we have a scenario where the participants were able to quickly understand the source code and start debugging, while in Project 2, we had a scenario where the source code was more complicated and unfamiliar to the participants.

Project 1 represents an ideal scenario, where participants understand the project's structure and can quickly start interpreting the ImpactViz visualization and associated source code to track down the bug. Under these circumstances, we can see that difference in both time and accuracy is significant, with participants using ImpactViz to complete the task 60% faster than the participants who used the baseline method, and with 100% of the participants finding the bug versus 80% of the participants finding the bug using the baseline method.

Project 2 was more complex and harder for the participants to understand. This scenario represents a case where participants are new to the project and are trying to figure out how the system was designed. As well, many users failed to properly inspect the method in which the bug was contained, as it was a constructor method. This scenario is an uncommon scenario, as people are only unfamiliar with an entire project only at the start of their involvement and become more knowledgeable overtime. Nevertheless, participants showed a positive trend using ImpactViz over the baseline debugging techniques, performing 20% faster in identifying where they thought the bug was, and three times as many users found the correct bug (30% of ImpactViz participants found the bug, versus 10% of the participants using Baseline method). While not significant results, this suggests that ImpactViz allows users to be faster

and more accurate in less than ideal debugging conditions.

While the difference between both interfaces were significant for Project 1, this wasn't the case for Project 2, although there was a positive trend in favour of ImpactViz. Looking at the raw data, we can say that participants had an easier time with debugging Project 1 over Project 2, based on the much lower average time to track down the bug and the much high success rate with both interfaces. We cannot conclusively say that H1 was validated by the evaluation, as only half the tests yielded statistically significant results.

From the two projects, we can see that ImpactViz does help users in finding the right bug faster, when provided with enough information. In a real world situation, the information provided in these tests is not uncommon to most developers. We can also see that even if the developer is not familiar with the project, as shown by the second project, participants were still shown to debug faster and more accurately over the traditional methods, although this difference was not statistically significant.

6.1.2 Qualitative Hypotheses

The last three hypotheses predicted that participants would respond positively to questions about the usefulness and ease of use, that participants would rank ImpactViz higher than the traditional debugging method for supporting debugging tasks, and that when it comes to understanding the software code, the traditional method would be preferred to ImpactViz.

For H3, both groups of participants found ImpactViz to be both useful and easy to use. What is most interesting, however, is that the group who used ImpactViz on

Project 2 (which was considered the more difficult project) found it to be much more useful and easy to use (Group 2 had 15 more "Strongly Agree" responses for usefulness and 9 more "Strongly Agree" responses for ease of use). While ImpactViz did not allow the participants to be much faster or accurate (from a statistical perspective), they did prefer to use ImpactViz to help explore and understand the software project. The results, however, could of been from the novelty of a new system; a longitudinal study should be performed to see how users respond to ImpactViz after repetitive use.

It is very interesting to see that despite being given a "harder" task, the participants who debugged using ImpactViz for Project 2 greatly preferred to use ImpactViz in their future debugging tasks (H4) over the other participant group, despite using the Baseline method on the "easy" project. Since the average completion time for participants using ImpactViz on Project 1, The Game, was significantly less than Project 2, Catering. With this extra time during the catering project, the participants were able to spend more time learning and understanding the tools of ImpactViz, and the many ways the tools can be used. The participants using ImpactViz on the catering project received additional time with ImpactViz, which might be enough extra time to receive a better impression of the capabilities of ImpactViz in a real world situation.

For H5, there was no significant difference between both groups' opinions of using ImpactViz to help understand the source code for the projects. While ImpactViz was not specifically designed to help understand the software code in general, it is interesting to see that participants found the visual representation of the method call stack assisted them with this task. While not an expected result, it is good to see

that some participants could use ImapactViz to understand the actual software code with ImpactViz.

The results given in Sections 5.3 through 5.5 show that participants found ImpactViz useful and easy to use when it comes to debugging tasks. This provides evidence that users are ready and willing to use a visual tool for future debugging tasks.

6.2 Field Trial Results

In the field trials, the participants were divided into two groups. The first group consisted of two students from a large team-based project, developed over four months. While this project was finished, it was still interesting to see the benefits of ImpactViz as a debugging tool. The second group consisted of two post-doctoral researchers working on a project in which they were the only developers. The project had been underway for two months, and was on-going.

The first group of field trial participants found ImpactViz to be very helpful for code exploration. While developing the software, each participant focused their energies on a particular area of the project, and had lost a sense of the overall project. Using ImpactViz they started to see patterns they hadn't noticed before and errors and problems in the overall design of the system. One participant started to make suggestions on how to improve the implementation. Since this was a large group project that had 13 developers, this field trial reinforces that participants of large team-based projects have much to gain in using ImpactViz, helping to improve their overall understanding of how the project has been implemented. In this project,

neither participant was able to think of a specific debugging scenario, as the project had been completed for about six weeks by the time the field trial was conducted.

For the secod group, the team size was much smaller. The two developers worked together in the same office, with their workstations located near enough to one another to allow nearly constant communication when necessary. In this scenario, the two developers found ImpactViz less useful for code exploration, as both individuals were already very familiar with how the system was designed and implemented. However, both participants agreed that ImpactViz would be very beneficial to the team if another developer joined them and to help bring this individual up to speed. We were unable to evaluate a debugging scenario, since the participants were unable to think of a bug to track down in their own system.

From the field trials, we can see that ImpactViz is a much more useful tool in a large team environment than for a small team. When people are given a specific area to work in, they can quickly lose touch with other areas of the project and forget how other sections of code interact with one another. ImpactViz helps visualize the dependency and method calls and helps the developer to quickly recall how the system is designed and further understand how the code in other areas of the project are put together. In a small team environment, where all developers are already familiar with the project and classes, there is less information to gain from using ImpactViz. But the tool can still act as an orientation aid for new developers, helping them gain familiarity with the system.

6.3 Benefits

The main research question that this thesis was focused on was whether or not a visual debugging tool was necessary and whether users would find it beneficial. From the results of both studies, we can conclude that software developers can benefit from a visual debugging tool to aid them in debugging tasks. We have seen that under normal circumstances, that the usage of ImpactViz helps increase the speed and accuracy of finding a bug. In more difficult debugging cases, there is also evidence to suggest it can help improve their speed and accuracy. The majority of the participants reported that they found ImpactViz both useful and easy to use in tackling their debugging tasks. Lastly, users reported a preference to use ImpactViz in future debugging tasks over traditional methods.

All the evidence collected leans very heavily in favour of users being ready and willing to use visual techniques to help aid them in their debugging tasks, and that ImpactViz is a useful step in the this direction.

6.4 Drawbacks

While performing the evaluations, we noticed areas where the use of ImpactViz had little to no benefit to a software developer. For example, as we saw in the Project 2 of the laboratory studies, if users have a hard time understanding the code or the context of the code, the benefit in using visual debugging tools to help them find the bug is limited. This may be due to the fact that in such cases, it is difficult for the user to generate an accurate mental model of the software, even when shown the class

dependencies through the method call stack.

Another area in which ImapctViz offers little value is for developers who are working in small teams and are very familiar with the code, as seen from our second field trial. Since both members of the team worked in a variety of areas in the project, both members were very familiar with the code they each had written. ImpactViz offered little in the way of allowing the developers to further learn about their software. As such, these participants graded it as not being very useful.

Lastly, using ImpactViz in a project with a very shallow method call stack would not be useful. A software project where there is little in the way of code reuse, produces a shallow method call stack. A method call stack that is four or five levels deep may be quickly transversed manually through most software tools. In this case, the overhead of running and using ImpactViz may be more of a burden than a benefit.

Chapter 7

Future Work

Through out this thesis, a software visualization system known as ImpactViz has been explained, evaluated, and the results discussed. In this chapter, we will outline further plans involving ImpactViz. There are four main areas in which we want to focus our future work in: integrating ImpactViz within an integrated development environment (IDE), conducing further evaluations, enhancing the information that is visually represented in ImpactViz, and adding further interaction tools that can help address potential information overload issues with exceptionally large software projects.

7.1 Plugin With an Integrated Development Environment

Ideally, the ultimate goal for ImpactViz would be to develop a plugin to work with an IDE, such as Eclipse or Netbeans. By bundling ImpactViz into an IDE, users

can use the software project within the IDE to aid in the debugging project, without resorting to switching between programs to review the code and to analyze the visual tool. Users could debug their source code and switch between the development view and ImpactViz, with instant interaction between the visualization and the text based view of the latest version of their software code.

7.2 Further Evaluations

While the results of the previous evaluations were in favour of ImpactViz, there is still a need for further evaluation. While the current evaluations in the laboratory study reveals that ImpactViz is very useful and aids in debugging under the right circumstances, further study will allow us to see how well ImpactViz can perform in abnormal scenarios. These scenarios may include missing information, like an unknown starting point or missing temporal data, and could compare the results to traditional debugging methods.

Since the evaluation results were not significant for the one abnormal case we did test (where users had trouble understanding the source code), recruiting more participants to perform that abnormal circumstance is required to prove if ImpactViz can be beneficial to complex software projects that participants are unfamiliar with. Evaluating this scenario further can help show whether ImpactViz can be an aid to new developers. A longitudinal study would be well worth performing to see if the qualitative responses from the evaluation was a result from the novelty of ImpactViz skewing the results.

Further studies could also be used to find the boundaries under which ImpactViz

can be useful. On how small of a software project will the overhead of using ImpactViz result in more of a burden in comparison to traditional debugging techniques? While designed to be scalable, are there extremely large projects for which the visualization method employed by ImpactViz is more difficult to use than traditional debugging methods? If such large projects do exist, what other methods and tools can we design and implement to help make ImpactViz scale better? Designing tests around the size of a project should help uncover when the overhead of using ImpactViz is too high to be of use, as well as how large a project has to be before the current representation becomes too difficult to understand.

There is also the question of what types of software projects for which ImpactViz is most useful. Would users find more benefitial using ImpactViz in a software project where software code is strongly connected, or in software where there is very little code reuse? Would ImpactViz perform better in a project that is broken into independent modules or linked together via a framework or database system? Could users perform debugging tasks successfully on concurrent or grid computing software? Future evaluations will expand on the types software projects to outline where ImpactViz's strengths and weaknesses lie.

7.3 Representation

In the current version of ImpactViz, we are visualizing everything at the class level, despite the fact that the method relation information is used to show how classes interact with one another. In future versions of ImpactViz, the actual method belonging to the classes will also be represented. Clicking on a class will result in all the

methods related to that class appearing, ballooned around the selected classes, with edges pointing to the methods of the classes upon which they depend. Clicking one of the method nodes could then select only the method call stack that the selected method depends upon. This visualization would require further user evaluations to ensure that participants will be able to properly decode the difference between class nodes and method nodes, and trace the method call stack amid the added visual complexity.

This new visualization would also require further thought to how the edges in the graph layout are represented. Since methods can override or make abstract methods concrete, showing these relations can be helpful for providing a deeper understanding of the structure of the project. The visual representation could also be extend into class relations, showing parent-child relations in classes using ImpactViz, as well as template usage.

The last item left to discuss on the future of ImpactViz's representation is the use of variables. Bugs can be introduced that cause class members to provide what should be an invalid value, through the use of public level access to the members or by local method calls. As an example, a sum-square method, a method that squares the value in a list and adds the squares together, could set the class member sum to -1, when the array is empty. When this value is stored in a variable and another method uses it as the input to a square root function, an exception will occur. The current version of ImpactViz can not help to trace bugs of this nature, as the summing method is not part of the method call stack used with the square root function (the value is passed through an intermediate variable). An alternate visualization, showing how the values calculated by one set of class members are used by another set of methods

might be useful for showing this inter-method reliance that is linked through value calculations. Evaluations will be required on this type of visualization, to see if the information presented to the developer can be understood and used effectively. This type of visualization could also extend into scenarios where multiple threads are modifying the same class members, helping in the debugging of multi-threaded projects.

7.4 User Interaction

While the original ImpactViz was designed to be scalable with larger projects, observations from participants showed some difficulty in locating a particular class. While this may not be a problem for software projects with a relatively small number of classes that can be visually searched and inspected, for much larger projects finding a specific class could be very difficult. In order to aid in users in finding classes, there are two tools we are considering to help aid in this process, a search function and a tree navigator.

The search function will allow users to enter in a class name in a textbox, resulting in the focusing of the matched class in a manner similar to if the user had clicked on the node. A dynamic drop-down menu may help speed up the process, reduce errors, and show the classes that partially match the text entered beneath the textbox. Another idea is to allow for all partial matches to be shown as focused nodes and have all their method stacks highlighted. Any further text entry will narrow the matches, reducing the number of focused classes.

The second idea is to use a tree navigation tool. Since organized projects have

classes separated into packages (or similar structures in other languages, like folders), using a tree navigator could allow users to select the classes they want to be visible in the visualization. This would help reduce the visual clutter of very large projects, and could allow users to tell the visualization system what area of the project they are interested in. The complicating factor here is how to represent method calls that go from visible classes, to one or more that are not being show, and back to visible classes. One possibility is to use a special glyph placed between the two visible classes to illustrate that there are intermediate classes within the subset of the method call stack. Clicking on this glyph could then show these intermediate classes.

Chapter 8

Conclusion

ImpactViz is a software visualization tool that allows developers to analyze the method call stack and associated class dependencies. ImpactViz also allows users to visualize the impact that changes in the software code may have on the operation of other classes within the project. The goal is to enable the user to easily trace a bug from the class in which it has manifested itself to its source location. By considering which classes have been changed between a previous known bug-free state and when the bug was first identified, the software developer can readily identify whether these changed classes have an impact on the class in which the bug has manifested itself. The system was designed to take into account modern modular programming practices and to use visual representations to indicate classes in which the bug may have been introduced.

The novel contribution that ImpactViz makes to the literature is the way in which it supports users in identifying the impact of classes that have been changed have on other classes in the project. This information is automatically extracted from the

source code of the project and an analysis of the revision history within a software repository. The interdependency of the classes are visually encoded as a graph; the change impact regions are visually encoded using colour. Together, these allow the user to readily perceive, interpret, and evaluate the potential impact that a change in one class might have on another class. Interactive features further support the debugging process, allowing the user to filter the revision history information, zoom into an area within the graph layout that is of interest, and select classes to focus on their specific method call stack dependencies.

The user evaluations provide empirical evidence of the value of the visual and interactive approaches used in ImpactViz over traditional debugging methods. Using ImpactViz, participants were able to see which classes may contain the source of the bug, limiting their evaluation to only these classes. As a result, participants were able to find the bugs faster and with more accuracy in one of the two cases. They greatly preferred using ImpactViz over more traditional debugging techniques, even in the situations where it didn't allow them to perform better from a quantitative perspective. These results illustrate the value of using visualization to represent the complex information that is present during debugging activities.

During the field trials, we examined how two different groups could use ImpactViz for their own software development needs. The main difference between the groups was that the first one consisted of novice developers who worked in a large team setting; whereas the second team consisted of experienced developers who worked in a small team. Each group had a different view on the usefulness of ImpactViz. The members of the first group found ImpactViz to be very useful and helped them identify patterns and inconsistencies in the design of the project, as well as how

classes interacted with one another. Meanwhile, the second group found limited value in using ImpactViz, as both members of the team were involved in developing each portion of the system. This has lead to the conclusion that ImpactViz is most useful to larger teams, where individuals are less likely to have a strong understanding of the entire software project. From both sets of evaluations, the participants all agree that ImpactViz was very easy to use.

From the two sets of evaluations, we have begun to see the areas in which ImpactViz is useful and users have responded very positively to it. Further evaluations will be required to help find the settings in which ImpactViz is most useful. However, the results from the evaluations suggests that ImpactViz is a useful tool for developers to use when tracking down bugs in a large method call stack, as well as identifying how a system's architecture has been implemented.

Bibliography

[1] T. Ball and S. G. Eick. Software visualization in the large. *IEEE Computer*, 29(4):33–43, 1996.

[2] M. Balzer, O. Deussen, and C. Lewerentz. Voronoi treemaps for the visualization of software metrics. In *Proceedings of the ACM Symposium on Software Visualization*, pages 165–172, 2005.

[3] H. Byelas and A. Telea. Visualization of areas of interest in software architecture diagrams. In *Proceedings of the ACM Symposium on Software Visualization*, pages 105 – 114, 2006.

[4] S. Carpendale. Evaluating information visualizations. In A. Kerren, J. T. Stasko, J.-D. Fekete, and C. North, editors, *Information Visualization: Human-Centered Issues and Perspectives*, LNCS 4950, pages 19–45. Springer, 2008.

[5] W. S. Cleveland. *Visualizing Data*. Hobart Press, 1993.

[6] W. S. Cleveland and R. McGill. Graphical perception: Theory, experimentation, and application to the development of graphical methods. *Journal of the American Statistical Association*, 79(387):531–554, 1984.

[7] C. Collins, G. Penn, and S. Carpendale. Bubble sets: Revealing set relations with isocontours over existing visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 15(6):1009 –1016, 2009.

[8] F. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *Management Information Systems Quaterly*, 13(3):319–340, 1989.

[9] A. Dix, J. Finlay, G. Abowd, and R. Beale. *Human-Computer Interaction.* Prentice-Hall, Inc., 1997.

[10] S. G. Eick, J. Steffen, and E. Sumner, Jr. SeeSoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, 1992.

[11] M. Follett and O. Hoeber. ImpactViz: Visualizing class dependencies and the impact of changes in software revisions. In *Proceedings of the ACM Symposium on Software Visualization*, pages 209–210, 2010.

[12] M. Fowler and K. Scott. *UML Distilled.* Addison-Wesley, 2000.

[13] J. Froehlich and P. Dourish. Unifying artifacts and activities in a visual tool for distributed software development teams. In *Proceedings of the International Conference on Software Engineering*, pages 387–396, 2004.

[14] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Software: Practice and Experience*, 21(11):1129–1164, 1991.

[15] W. O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques.* John Wiley & Sons, Inc., New York, NY, USA, 2007.

[16] H. Gall, M. Jazayeri, and C. Riva. Visualizing software release histories: The use of color and third dimension. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 99–108, 1999.

[17] C. Healey. Choosing effective colours for data visualization. In *Proceedings of Visualization*, pages 263–270, 271996-nov.1 1996.

[18] J. Heer, S. K. Card, and J. A. Landay. Prefuse: A toolkit for interactive information visualization. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 421–430, 2005.

[19] E. Hering. *Outlines of a Theory of the Light Sense.* Haravrd University Press, 1964.

[20] I. Herman, G. Melancon, and M. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, 2000.

[21] O. Hoeber. User evaluation methods for visual web search interfaces. In *Proceedings of the International Conference Information Visualisation*, pages 139–145, 2009.

[22] D. Holten. Hierarchical edge bundles: Visualization of adjacency relations in hierarchical data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):741–748, 2006.

[23] B. Johnson and B. Shneiderman. Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the Conference on Visualization*, pages 284–291, 1991.

[24] J. Li, Z. Guo, Y. Zhao, Z. Zhang, and R. Pang. Towards quantitative evaluation of UML based software architecture. In *Proceedings of the ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing*, pages 663–669, 2007.

[25] J. D. Mackinlay. Automating the design of graphical presentations of relational information. *ACM Transactions on Graphics*, 5(2):110–141, 1986.

[26] S. McConnell. *Code Complete.* Microsoft Press, 2nd edition, 2004.

[27] W. Metzger. *Laws of Seeing.* MIT Press, 2006.

[28] J. Nielsen. *Usability Engineering.* Academic Press, 1993.

[29] J. Nielsen and R. L. Mack. *Usability Inspection Methods.* John Wiley & Sons, 1994.

[30] D. A. Norman. *The Design of Everyday Things.* Basic Books, 2002.

[31] M. Ogawa and K. Ma. StarGate: A unified, interactive visualization of software projects. In *Proceedings of the IEEE Pacific Visualization Symposium*, pages 191–198, 2008.

[32] C. Pich, L. Nachmanson, and G. G. Robertson. Visual analysis of importance and grouping in software dependency graphs. In *Proceedings of the ACM Symposium on Software Visualization*, pages 29–32, 2008.

[33] C. Plaisant. The challenge of information visualization evaluation. In *Proceedings of the Working Conference on Advanced Visual Interfaces*, pages 109–116, 2004.

[34] D. Price. CVS - open source control. http://www.nongnu.org/cvs/, December 2009.

[35] N. Riche and T. Dwyer. Untangling Euler diagrams. *IEEE Transactions on Visualization and Computer Graphics*, 16(6):1090–1099, 2010.

[36] J. Rossenberg. Some misconceptions about lines of code. In *Proceedings of Software Metrics Symposium*, pages 137–142, 1997.

[37] M. Samuels and J. Witmer. *Statisticss for the Life Sciences.* Pearson Education, 2008.

[38] S. R. Schach. *Object-Oriented and Classical Software Engineering.* McGraw-Hill Publishing Co., 2001.

[39] R. Spence. *Information Visualization: Design for Interaction.* Prentice Hall, 2nd edition, 2007.

[40] Tigris. Open source software engineering tools. http://subversion.tigris.org/, December 2009.

[41] L. Voinea, J. Lukkien, and A. Telea. Visual assessment of software evolution. *Science of Computer Programming*, 65(3):222–248, 2007.

[42] L. Voinea, A. Telea, and J. J. van Wijk. CVSscan: Visualization of code evolution. In *Proceedings of the ACM Symposium on Software Visualization*, pages 47–56, 2005.

[43] C. Ware. *Information Visualization: Perception for Design.* Morgan Kaufmann, 2nd edition, 2004.

[44] A. Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the Software Engineering Conference*, pages 253–267, 1999.

[45] T. Zuk, L. Schlesier, P. Neumann, M. S. Hancock, and S. Carpendale. Heuristics for information visualization evaluation. In *Proceedings of the AVI Workshop on Beyond Time and Errors: Novel Evaluation Methods for Information Visualization*, pages 1–6, 2006.

Appendix A

User Evaluation Documents

On the proceeding pages are the documents given to the participants of the user evaluation (laboratory studies). The first document is the consent form which every participant signed to confirmed they understood what the study entailed and where they can register complaints about this study if they feel it to be necessary. The second document was a post-task questionnaire asking information on the participants programming history. The third document was the post-task questionnaire, which was given after both tasks were completed to evaluate how the participant felt about ImpactViz as a debugging tool.

Informed Consent by Subjects to Participate in

Visualizing the Impact of Changes in Software Code
(User Evaluation)

I understand that this form and the information it contains are given to me for my protection and full understanding of the procedures of this research. My signature on this form signifies that I have received a copy of this consent form, that I understand the procedures to be used in this study, and the personal risks and benefits to me in taking part. I voluntarily agree to participate in this project. I understand that I may withdraw my participation in this study at any time, and that my decision to participate in this study, and my subsequent involvement. In it, will have absolutely no bearing on any other dealings I have with Mr. Follett or Dr. Hoeber.

I will not be required to write my name or any identifying information on the research questionnaires. My use of the system will be recorded (video and audio), and comments I make relevant to the assigned tasks or to the use of the software will be transcribed. I consent that the video recording of my use of the system will only be used for analysis purposes and that it will not be distributed or shared in its raw format. All research materials will be held confidential by Dr. Hoeber and kept in a secure location for five (5) years upon completion. I consent that the data gathered from this study will be used in research papers and thesis work and that no information that identifies me as a member of this study will be published.

I agree to participate by completing a debugging task using two different interfaces; and a post-task questionnaire. I understand that these activities will require approximately half an hour, and will be conducted in the User Experience Lab in the Department of Computer Science at Memorial University. I understand that there is compensation for my participation and time in the form of $10.

The proposal for this research has been reviewed by the Interdisciplinary Committee on Ethics in Human Research and found compliance with Memorial University's ethics policy. If you have ethical concerns about the research (such as the way you have been treated or your rights as a participant), you may contact the Chairperson of the ICEHR at icehr@mun.ca or by telephone at 737-8368.

I may obtain copies of the results in this study, upon completion, by contacting Dr. Hoeber, in care of the Computer Science Department, Memorial University.

NAME (please print legibly): _____

SIGNATURE: _____

INVESTIGTOR: _____

DATE: _____

Investigators:

Mr. Matthew Follett (Primary Investigator)
Student of Masters of Science
Department of Computer Science
Memorial University

Dr. Orland Hoeber (Master's Supervisor)
Department of Computer Science
Memorial University
hoeber@cs.mun.ca
709-737-32222

Pre-Task Questionnaire

Please answer the following questions in regards to your background. Circle the answer the best describes you.

1. What year of Computer Science are you in?

   a. First b. Second c. Third d. Fourth (or higher)

2. Have you ever worked on a group project that required the use of a software repository (CVS, SVN, Git, etc.)?

   Yes No

3. If yes, how many projects have you worked on that have used a software repository?

   1 2 3 4 5+ N/A

4. What is your level of understanding of software repository and the value they provide to software development team?

   Low Medium High

5. How would you rate your debugging skills in locating bugs in software?

   Terrible Below Average Average Above Average Exceptional

6. What Programming Language are you most comfortable with?

   Java C/C++ Visual Basic Other (Please list below)

   _____

7. What OS are you most comfortable with?

   Windows Platform Apple Platform Linux Other (Please list below)

   _____

Post-Task Questionnaire

The following questions relate to your experience using ImpactViz for finding bugs in software code. Your answers to the following questions allow for a more accurate analysis of the data collected for this study.

INSTRUCTIONS: Please rate how strongly you agree or disagree with the following statements by circling the appropriate number.

n/a	strongly disagree	disagree	neutral	agree	strongly agree	
0	1	2	3	4	5	Using **ImpactViz** for debugging enabled me to accomplish tasks more quickly.
0	1	2	3	4	5	Using **ImpactViz** improved my debugging performance.
0	1	2	3	4	5	Using **ImpactViz** for debugging increased my productivity.
0	1	2	3	4	5	Using **ImpactViz** enhanced my effectiveness when debugging software.
0	1	2	3	4	5	Using **ImpactViz** made it easier to find the source of the bugs.
0	1	2	3	4	5	I found **ImpactViz** useful for debugging software.

n/a	strongly disagree	disagree	neutral	agree	strongly agree	
0	1	2	3	4	5	Learning to operate **ImpactViz** was easy for me.
0	1	2	3	4	5	I found it easy to get **ImpactViz** to do what I wanted it to do.
0	1	2	3	4	5	My interaction with **ImpactViz** was clear and understandable.
0	1	2	3	4	5	I found **ImpactViz** to be flexible to interact with.
0	1	2	3	4	5	It was easy for me to become skilful at using **ImpactViz** .
0	1	2	3	4	5	I found **ImpactViz** easy to use.

n/a	strongly disagree	disagree	Neutral	agree	strongly agree	
0	1	2	3	4	5	I found the ability to filter the revision history to show only a subset of the changes to be useful.
0	1	2	3	4	5	I found the graphical representation of the connection between classes to be useful.
0	1	2	3	4	5	I found the graphical representation of the classes that have changed within a revision range to be useful
0	1	2	3	4	5	I found the graphical representation of the impact of a change to be useful.
0	1	2	3	4	5	I found the ability to select classes and highlight the other classes that impact it to be useful.
0	1	2	3	4	5	I found the ability to move classes to a different location within the visual representation to be useful.

Please put an **x** below the method you find most effective for the following:

	ImpactViz	**Traditional Debugging Methods**
Understanding the software code in general		
Understanding the method call stack		
Understanding the revision history		
Understanding the effect of changes		
Overall, the system I prefer to use for debugging		

If you have any further comments or suggestions, you may write them below:

Appendix B

Field Trial Documents

On the proceeding pages are the documents given to the participants of the field trials. The first document is the consent form which every participant signed to confirmed they understood what the study entailed and where they can register complaints about this study if they feel it to be necessary. The second document was the post-task questionnaire asking information on the participants programming history. The third document was the post-task questionnaire, which was given after both tasks were completed to evaluate how the participant felt about ImpactViz as a debugging tool.

Informed Consent by
Subjects to Participate in

Visualizing the Impact of Changes in Software Code
(Field Trials)

I understand that this form and the information it contains are given to me for my protection and full understanding of the procedures of this research. My signature on this form signifies that I have received a copy of this consent form, that I understand the procedures to be used in this study, and the personal risks and benefits to me in taking part. I voluntarily agree to participate in this project. I understand that I may withdraw my participation in this study at any time, and that my decision to participate in this study, and my subsequent involvement. In it, will have absolutely no bearing on any other dealings I have with Mr. Follett or Dr. Hoeber.

I will not be required to write my name or any identifying information on the research questionnaires. My use of the system will be recorded (video and audio), and comments I make relevant to the assigned tasks or to the use of the software will be transcribed. I consent that the video recording of my use of the system will only be used for analysis purposes and that it will not be distributed or shared in its raw format. All research materials will be held confidential by Dr. Hoeber and kept in a secure location for five (5) years upon completion. I consent that the data gathered from this study will be used in research papers and thesis work and that no information that identifies me as a member of this study will be published.

I agree to participate by using the software provided to me to examine a project that I will provide and provide feedback on the experience. I understand that these activities will require approximately half an hour, and will be conducted in the User Experience Lab in the Department of Computer Science at Memorial University. I understand that there is compensation for my participation and time in the form of $20.

The proposal for this research has been reviewed by the Interdisciplinary Committee on Ethics in Human Research and found compliance with Memorial University's ethics policy. If you have ethical concerns about the research (such as the way you have been treated or your rights as a participant), you may contact the Chairperson of the ICEHR at icehr@mun.ca or by telephone at 737-8368.

I may obtain copies of the results in this study, upon completion, by contacting Dr. Hoeber, in care of the Department of Computer Science, Memorial University.

NAME (please print legibly): _____

SIGNATURE : _____

INVESTIGTOR: _____

DATE: _____

Investigators:
Mr. Matthew Follett
Masters of Science
Department of Computer Science
Memorial University

Dr. Orland Hoeber
Department of Computer Science
Memorial University
hoeber@cs.mun.ca
709-737-3222

Background Survey

Please answer the following questions in regards to your background. Circle the answer the best describes you.

1. What year of Computer Science are you in?

   a. First b. Second c. Third d. Fourth (or higher)

2. Have you ever worked on a group project that required the use of a software repository (CVS, SVN, Git, etc.)?

   Yes No

3. If yes, how many projects have you worked on that have used a software repository?

   1 2 3 4 5+ N/A

4. What is your level of understanding of software repository and the value they provide to software development team?

   Low Medium High

5. How would you rate your debugging skills in locating bugs in software?

   Terrible Below Average Average Above Average Exceptional

6. What Programming Language are you most comfortable with?

   Java C/C++ Visual Basic Other (Please list below)

   _____

7. What OS are you most comfortable with?

   Windows Platform Apple Platform Linux Other (Please list below)

   _____

Field Trial Questionnaire

The following questions relate to your experience using ImpactViz for in visualizing your software project. Your answers to the following questions allow for a more accurate analysis of the data collected for this study.

INSTRUCTIONS: Please rate how strongly you agree or disagree with the following statements by circling the appropriate number.

n/a	strongly disagree	disagree	neutral	agree	strongly agree	
0	1	2	3	4	5	Using **ImpactViz** for debugging enabled me to accomplish tasks more quickly.
0	1	2	3	4	5	Using **ImpactViz** improved my debugging performance.
0	1	2	3	4	5	Using **ImpactViz** for debugging increased my productivity.
0	1	2	3	4	5	Using **ImpactViz** enhanced my effectiveness when debugging software.
0	1	2	3	4	5	Using **ImpactViz** made it easier to find the source of the bugs.
0	1	2	3	4	5	I found **ImpactViz** useful for debugging software.

n/a	strongly disagree	disagree	neutral	agree	strongly agree	
0	1	2	3	4	5	Using **ImpactViz** for exploring my system enabled me to accomplish tasks more quickly.
0	1	2	3	4	5	Using **ImpactViz** improved my exploring my system performance.
0	1	2	3	4	5	Using **ImpactViz** for exploring my system increased my productivity.
0	1	2	3	4	5	Using **ImpactViz** enhanced my effectiveness when exploring my system software.
0	1	2	3	4	5	Using **ImpactViz** made it easier to explore my software system.
0	1	2	3	4	5	I found **ImpactViz** useful for exploring my system software.

n/a	strongly disagree	disagree	neutral	agree	strongly agree	
0	1	2	3	4	5	Learning to operate **ImpactViz** was easy for me.
0	1	2	3	4	5	I found it easy to get **ImpactViz** to do what I wanted it to do.
0	1	2	3	4	5	My interaction with **ImpactViz** was clear and understandable.
0	1	2	3	4	5	I found **ImpactViz** to be flexible to interact with.
0	1	2	3	4	5	It was easy for me to become skilful at using **ImpactViz** .
0	1	2	3	4	5	I found **ImpactViz** easy to use.

n/a	strongly disagree	disagree	Neutral	agree	strongly agree	
0	1	2	3	4	5	I found the ability to filter the revision history to show only a subset of the changes to be useful.
0	1	2	3	4	5	I found the graphical representation of the connection between classes to be useful.
0	1	2	3	4	5	I found the graphical representation of the classes that have changed within a revision range to be useful
0	1	2	3	4	5	I found the graphical representation of the impact of a change to be useful.
0	1	2	3	4	5	I found the ability to select classes and highlight the other classes that impact it to be useful.
0	1	2	3	4	5	I found the ability to move classes to a different location within the visual representation to be useful.

Please add any additional feedback to the back of this page.