A FRAMEWORK FOR EFFICIENT CREATION AND CUSTOMIZATION OF HIGH LEVEL PROGRAM VISUALIZATIONS

A DISSERTATION SUBMITTED TO THE GRADUATE DIVISION OF THE UNIVERSITY OF HAWAI’I AT MĀNOA IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN COMPUTER SCIENCE

AUGUST 2011

By

Jo-Han Wu

Dissertation Committee:

Jan Stelovsky, Chairperson
Edoardo Biagioni
Martha Crosby
Nancy Reed
Marie Iding

Keywords: ProViz, Software, Program, Visualization, Algorithm
Acknowledgements

I am indebted to my advisor, Dr. Jan Stelovsky, for originating the idea and allowing me to explore and develop this work. I am so grateful for his guidance, patience, encouragement, and friendliness. It is a privilege to have Jan as my advisor.

I am grateful to my committee members: Professors Edoardo Biagioni, Martha Crosby, Nancy Reed, and Marie Iding. Their advices and comments were extremely helpful, especially on my writing. Special thanks to Edo for helping me with the research, and to Marie for referring me to materials that I needed but was not familiar with.

I must also thank the UH ICS department for many years of generous financial support.

Most importantly, I want to thank my parents for believing in me, supporting me, loving me, and praying for me. Thank God for such wonderful parents, and thank God for everything He has done for me in my life.
Abstract

The purpose of software visualization is to facilitate program comprehension by visualizing artifacts of software programs using graphical representations. In particular, high-level, algorithm visualizations can be used to describe abstract concepts and algorithms to achieve effective understanding of programs. However, it has always been a difficult and time-consuming task to produce such high-level visualizations for programs. This diminishes programmers’ willingness to create visualizations for their programs and therefore limits the usage and application of program visualization.

To bridge this gap in the foundation of software visualization, we developed the “ProViz” framework that allows programmers to rapidly generate one or even several graphical depictions of a program and see the execution of this program as it runs. This is accomplished by (1) decoupling the visual programming from the target program through visual mapping tools, (2) utilizing data-driven design and method-driven capability, and (3) providing object-oriented software architecture of painters with high reusability. ProViz can facilitate the creation of versatile visualizations where the resulting visualizations are interactive and customizable, and the visualization can also be recorded to a file and played back. Being able to effortlessly create high-level visualizations for programs not only fulfills pedagogical purposes for educators but also is a fundamental step towards increasing the productivity of key elements in software development processes – from coding and debugging to maintenance.

A prototype of the ProViz framework was embedded in the popular integrated development environment Eclipse. In a pilot study, it was utilized in an introductory computer science course where the instructor was able to demonstrate algorithms visually and the students could visually debug their programs. Moreover, a usability test was administered to experienced programmers to evaluate the effectiveness of the framework for general purpose software development. The results showed that programmers can indeed use ProViz to create or customize program visualizations with reasonable effort and time.
Table of Contents

Acknowledgements .................................................................................................................................. ii
Abstract ............................................................................................................................................. iii
List of Tables ...................................................................................................................................... vii
List of Figures .................................................................................................................................... viii
List of Abbreviations ......................................................................................................................... ix
Chapter 1 Introduction ........................................................................................................................ 1
  1.1 Statement of Problems .................................................................................................................. 5
    1.1.1 Lack of Support for Understanding ....................................................................................... 5
    1.1.2 Difficulty in Creating Visualization ..................................................................................... 6
    1.1.3 Usability Issues ....................................................................................................................... 6
    1.1.4 Confusion Between Variables and Objects ........................................................................... 7
  1.2 Research Objective ...................................................................................................................... 7
    1.2.1 Visualizing Programs in High-Level Abstraction ................................................................. 8
    1.2.2 Data-Driven Design and Painters ......................................................................................... 10
  1.3 Contribution ................................................................................................................................... 11
  1.4 Brief Summary of this Dissertation ............................................................................................ 13
Chapter 2 Background and Literature Review ................................................................................... 14
  2.1 User Roles ..................................................................................................................................... 15
    2.1.1 Author .................................................................................................................................. 15
    2.1.2 Animator ............................................................................................................................... 16
    2.1.3 Viewer ................................................................................................................................... 16
    2.1.4 Visualization for Different Types of Viewers ....................................................................... 17
  2.2 Categorization of Software Visualization .................................................................................... 17
    2.2.1 Static Program Visualization ................................................................................................. 18
    2.2.2 Dynamic Program Visualization ........................................................................................... 19
    2.2.3 Algorithm Visualization ........................................................................................................ 21
  2.3 Abstraction ................................................................................................................................... 24
    2.3.1 Data Abstraction .................................................................................................................... 25
    2.3.2 Presentation Abstraction ......................................................................................................... 26
  2.4 Dynamic Program Visualization Techniques ............................................................................... 27
    2.4.1 Event Driven (Interesting Events) ......................................................................................... 28
    2.4.2 Data Driven (State Mapping) ............................................................................................... 29
    2.4.3 Automatic Visualization ........................................................................................................ 31
  2.5 Code Intrusion ............................................................................................................................. 31
  2.6 Related Work ............................................................................................................................... 34
Chapter 3 The ProViz Framework ........................................................................................................ 39
  3.1 Target Users ................................................................................................................................. 40
  3.2 Language ....................................................................................................................................... 41
  3.3 Environment ................................................................................................................................. 41
    3.3.1 Debugger ................................................................................................................................. 41
    3.3.2 Views ..................................................................................................................................... 43
  3.4 Methodologies ............................................................................................................................... 43
    3.4.1 Data-Driven Visualization ...................................................................................................... 43
    3.4.2 Method-Driven Visualization ................................................................................................. 44
List of Tables

Table 1 – A comparison between DPV and AV ......................................................... 24
Table 2 – Levels of code intrusion ............................................................................. 32
Table 3 – @Viz for classes and methods ................................................................... 59
Table 4 – Time (in minutes) for completing each task in the usability test ................. 119
Table 5 – Difficulty of each task as rated by the participants .................................... 119
List of Figures

Figure 1 – A snapshot of ProViz in the Eclipse IDE ......................................................... 3
Figure 2 – Different representations of bubble sort ........................................................... 9
Figure 3 – A different visualization of java.util.TreeMap .............................................. 10
Figure 4 – Static PV versus dynamic PV ............................................................... 19
Figure 5 – Jeliot3 .............................................................................................................. 35
Figure 6 – Categorization of ProViz in software visualization .......................................... 39
Figure 7 – ProViz’s data-driven process ............................................................................ 44
Figure 8 – ProViz’s program monitoring with a debugger .................................................. 46
Figure 9 – Circular-pointer display in debugger view ........................................................ 51
Figure 10 – An example of VOV visualization ............................................................... 51
Figure 11 – An example of VOO visualization .................................................................. 52
Figure 12 – A different example of VOO visualization ...................................................... 53
Figure 13 – The data-driven process and corresponding components of ProViz .............. 56
Figure 14 – The architecture of ProViz ............................................................................ 57
Figure 15 – Example of an instance stack frame ............................................................... 78
Figure 16 – Example of a static stack frame ..................................................................... 79
Figure 17 – Structure of the runtime data for the Demo program ..................................... 80
Figure 18 – demo’s variable tree and the painter tree that corresponds to it. ....................... 80
Figure 19 – Navigating variables and painters ............................................................... 81
Figure 20 – Accessing the field1 variable in MyTypePainter ........................................... 83
Figure 21 – Painter lifecycle and the corresponding methods .......................................... 88
Figure 22 – Basic tasks of handleChange(...) ................................................................... 90
Figure 23 – Stack’s structure shown in the Variable View of the debugger ....................... 92
Figure 24 – The conceptual design of stack visualization ................................................. 94
Figure 25 – StackPainter in action .................................................................................. 99
Figure 26 – Animation threads and the timeline for a program step ................................ 100
Figure 27 – An example of scheduling simultaneous animations ..................................... 101
Figure 28 – The lifetime of a method painter ................................................................... 103
Figure 29 – Sequence of processing in a same method step .............................................. 105
Figure 30 – Example of a context (right-click) menu on a painter .................................. 110
Figure 31 – Visualizations of a stack in different abstractions ......................................... 114
Figure 32 – Visualization of the Sieve of Erasthothenes algorithm .................................. 116
Figure 33 – Visualization of Sieve of Erasthothenes without color .................................. 116
Figure 34 – Example of inconsistency without graphical event-listeners ....................... 132
Figure 35 – The “theater” view of Jeliot3 ......................................................................... 133
Figure 36 – Mockup of a method-oriented view ............................................................... 133
List of Abbreviations

SV: Software Visualization
PV: Program Visualization
DPV: Dynamic Program Visualization
AV: Algorithm Visualization
IDE: Integrated Development Environment
OO: Object Orientation
VOV: Visualization Of Variables
VOO: Visualization Of Objects
VPM: Viz Painter Manager
Chapter 1
Introduction

Visualization
Vision is the most powerful sensor that allows humans to receive information, and how information is presented to human eyes will influence the effectiveness of information reception, comprehension, and learning. Mayer’s (2001, 2005) multimedia principle suggests that presenting information with words and pictures results in better learning than using words alone. This implies that graphical representation can be much more effective than plain text description. Textual representation has been a major communication and storage medium since the development of writing. However, the ability to create graphics or videos has only become widely available with the advancement of computer technology, which gave people the opportunity to efficiently transform information into graphical forms – the process of visualization (Diehl, 2007). From the use of simple tables and charts to represent numeric data, to 2D or 3D blueprints for architectural designs, and to sophisticated TV shows and films that are visual embodiments of stories and novels, visualization has become ubiquitous in many areas of our lives.

Program Comprehension
Software is an obvious target for visualization, as program source codes are written in text. Embedded within this textual source code is the distilled mental model of its creator, which consists of abstract concepts in the forms of algorithms, procedures, features, problem-solving techniques, architecture, and even intelligent tricks or shortcuts. These concepts are concretized using programming constructs, expressions, and statements of programming languages to form programs (Jerding & Stasko, 1994) that are ultimately formulated as program texts. Program comprehension aims to accomplish the opposite – to reconstruct the mental model from program texts in order to achieve comprehension. In software development, it is an extremely vital skill for programmers to be able to efficiently achieve program comprehension in order to use, modify, or maintain these programs (Pacione, 2004). Program comprehension usually occupies a
large portion of the software lifecycle, accounting for at least 40% of the time (Upchurch, 1997).

The purpose of program visualization is to achieve program comprehension (see, e.g., Price, Baecker, & Small, 1998; Roman & Cox, 1993; Diehl, 2007, among others). By visualizing software artifacts in pictorial forms, visualization technology has great potential to reduce the excessive amount of effort currently exerted on program comprehension. The expressiveness of graphical representation can be of great aid once abstract concepts and algorithms embodied in a program can be visualized in such a way that people can absorb them visually. For developers who prefer to view and learn through graphical representations, visualizations can assist comprehension more efficiently and reduce the time spent on reading and interpreting the textual source code.

Visualization is also invaluable in computer science education. It has been a prominent aid in expressing programming concepts and abstract algorithms for students to learn and understand. Needless to say, computer science textbooks are full of pictures and diagrams that portray data structures and algorithms in order for students to grasp the concept of how they work.

**Program Visualization**

Although software visualization has had a significant impact in computer science, not all aspects of program visualization (PV) are truly developed and have been successfully applied. There are several categories of PV and many types of PV systems (described in the next chapter) that have resulted from the various ways that programmers perceive software programs.

First of all, the textual representation of a program exists in the form of textual source code and is often written in Roman characters with special symbols such as parentheses or braces according to the definition of the particular programming language. Secondly, a program can also be compiled into byte code or machine code in binary representation where it contains the same information as the source code but is hardly legible for a human. Lastly, the dynamic aspect of a program – its behavior (Diehl, 2007) – only appears when it is executing in a computer. Here it has the form of electrical
current flowing through the CPU and the computer memory. PV is generally concerned with the textual and the dynamic representations of a program, as the intermediate byte code offers little beyond the human-readable source code.

Modern programming practice promotes modularizing software, which results in code modules with a high level of abstraction. In particular, the object-oriented paradigm postulates classes as abstract entities that incorporate actions or behaviors that give these entities some abstract, semantic meaning. These abstractions are defined in the source code, and when a program executes, these abstract concepts and behaviors are put into action. As a result, these abstractions can be extracted from the execution and be visualized. For example, the source code may define a Lion class where a Lion can hunt and sleep. Suppose that when a program that uses this Lion class executes, it creates two Lions and makes one Lion hunt and the other sleep. Such a high-level concept is extremely difficult, if not impossible, to convey by visualizing low-level bits and bytes in the machine. On the other hand, high-level, dynamic PV should monitor the

![Figure 1 – A snapshot of ProViz in the Eclipse IDE. A java.util.TreeMap object is visualized as a binary tree in the middle-right “ProViz View” pane.](image-url)
program states as they dynamically change, capture these abstractions, and visualize them on the same level of abstraction as defined in the source code. For example, “ProViz View” in Figure 1 displays a `java.util.TreeMap` object graphically as a binary tree, which accurately depicts the structure of its internal data.

Unfortunately, creating high, algorithm-level visualizations of programs has been a difficult and also time-consuming task (see, e.g., Reiss, 2006; Ihantola, Karavirta, Korhonen, & Nikander, 2005; Mukherjea & Stasko, 1993, among others) and therefore has been considered impractical for the purpose of comprehension. As a consequence, most PV tools are low-level in terms of abstraction (Price, Baecker, & Small, 1998) and can only be used for debugging and analysis purposes but not for program understanding. High-level PV tools, on the other hand, either have fixed representations, such as UML diagrams, object diagrams, sequence diagrams, and message-passing views, or are limited in the scope of programs they can visualize, such as educational tools for visualizing small scale programs. As a consequence, there is no PV tool in standard programming environments that supports the creation of dynamic visualizations that portray program behaviors in high-level abstractions. The aim of this study is therefore to create a high-level PV framework – *ProViz* – capable of creating any form of visualizations for executing programs. This framework is integrated into the Eclipse programming environment to simplify the creation and viewing of visualizations, as well as the viewer’s interaction with these visualizations.

We believe that the inability to create PV at will has long put a dent in its applications and even in research studies. Previous studies and experiments have focused on research questions such as: “Does algorithm animation aid learning?” and “Is animation effective in learning?” Experiments were performed typically with a single, fixed representation of a program and usually concluded with mixed results (Byrne, Catrambone, & Stasko, 1996; Rajala, Laakso, Kaila, & Salakoski, 2008). In order for future research in PV to evolve towards asking such questions as: “How do different representations of the same program affect learning?” and “How can a program be visualized so that it can be best understood by a specific category of viewers?” and even to better answer the type of questions mentioned earlier, the PV process must be
simplified to the level where creating not just one but multiple, customized representations for a program becomes a task free of the tedium that has discouraged programmers in the past. Therefore, the purpose of this study is to provide a framework that can turn a previously difficult and time consuming task into an efficient practice for the future.

Moreover, once the process of creating and modifying PV is simplified and becomes common practice, it could allow software developers to produce visual/video documentation for their programs. Given the expressiveness of graphical representation, video documentation can become a very effective way to facilitate program comprehension.

1.1 Statement of Problems

PV started to emerge around the late 1970s when computer displays gradually became available, and one would expect it to have advanced and popularized as the graphics technology improved in leaps. However, even now, no dynamic program visualization (DPV) tool has had a significant impact in either the industry or the academic field (Reiss, 2005). After surveying existing tools and research, we attributed this to three problems: (1) the visualization produced may not be what the viewer needs for comprehension; (2) creating visualization for a program is a difficult and laborious task (Reiss, 2006); and (3) there are usability issues, particularly code intrusion.

1.1.1 Lack of Support for Understanding

The first problem with DPV tools is that they “do not address the reality of program understanding” (Reiss, 2005, p. 60). That is, what they visualize does not reflect what the viewers truly need to achieve program comprehension. Static PV visualizes static aspects of programs, but it cannot reveal the dynamic behavior of programs (Shilling & Stasko, 1992); DPV tools also tend to be ineffective for comprehension because they often visualize the low-level aspects of programs and therefore exhibit low levels of abstraction (Price, Baecker, & Small, 1998). While low-level information that is accurate and
detailed may facilitate verification and analysis, it can hardly achieve the high-level understanding that is crucial to most program comprehension activities.

1.1.2 Difficulty in Creating Visualization

The reason that DPV tools generally offer only low-level abstractions is due to the following fundamental problem: creating visualization for a program is laborious and time-consuming (Reiss, 2006; Ihantola, Karavirta, Korhonen, & Nikander, 2005; Mukherjea & Stasko, 1993). Because a high-level visualization contains semantic representations and operations that must involve human authoring, it has always been a time-consuming and labor-intensive task to create such visualization for programs. Furthermore, because human perception of images is subjective, a single visual representation may not be sufficient unless it can be customized to meet the needs of different audiences (Schafer & Mezini, 2005). As a result, PV has been an impractical task because the cost required (in time and effort) usually greatly outweighs the benefits that the resulting visualization provides.

1.1.3 Usability Issues

Code intrusion is a common side-effect of programming language tools where the program source code needs to be modified in order for these tools to function. It is therefore a reason why a tool may not be applicable to existing programs because such modification could change the behavior of programs or even damage them. It is an undesirable effect, but many DPV tools require severe code intrusion in order to function.

Another usability issue is the use of special programming or scripting languages for producing visualization. Some visualization tools use dedicated visual or scripting languages to produce visualization. Such an approach could in fact reduce usability because it not only increases the learning curve for using such a tool (i.e., learning a new language) but also decreases people’s motivation to adopt it.
1.1.4 Confusion Between Variables and Objects

The programming process in an object-oriented language involves two different paradigms: variables and objects. Programs are written with variables referencing objects, but objects do not actually exist in the source code; they are dynamically allocated in memory during runtime. Therefore, when software developers look at programs and see variables, they have to imagine objects being referenced. But whether these references and pointers are correct cannot be easily verified. Uncovering the real identity of objects, therefore, is very important in avoiding referencing errors. Particularly when variables are used in method parameters or when objects are stored in complex data structures, confusion about object identity may arise and become a problem for programmers.

Intersecting these two paradigms is the effect of aliasing, where variable to object is in a many-to-one relationship. Visualization is perhaps the best instrument in revealing the connection between variables and objects, but this should not be accomplished with an overly-rigid design that relies solely on one paradigm. For instance, the variable view in a debugger works entirely in the variable paradigm, so the concept of objects cannot be easily grasped from this view, and as shown in Section 3.6.1, erroneous display about the program can occur.

In fact, aliasing affects not only programming practices but also the design and presentation of a PV system, as we came to realize during the development of ProViz. No other PV systems have ever ventured into this aliasing dilemma because they typically visualize in just one paradigm.

1.2 Research Objective

The objective of this research is to supply what has always been missing in the foundation of program visualization technology – a DPV framework that supports efficient creation of high-level visualization. The purpose of this study is to address the following questions: Can there be a dynamic program visualization system that supports rapid, efficient visualization creation for programs? What methodologies should be applied to such a system in order to accomplish this goal?
We have developed a prototype DPV framework – ProViz – that addresses the problems mentioned above. In the mean time, we explored visualization paradigms for visualizing object-oriented programs on a *storytelling* level of abstraction. When creating high-level visualization for programs is no longer a tedious task, it can be imagined that instructors will frequently make program visualizations for pedagogical purposes where students can benefit from learning visually, and students will use it to visually see and debug their own programs. Furthermore, software developers will be able to create visualizations that serve as *visual/video documentation* for their programs, where adding graphical depictions of a program to the mix of other documentation means could improve the efficiency and effectiveness of program comprehension.

The prototype of ProViz is implemented in Java as a plug-in to the Eclipse IDE (Integrated Development Environment). This is the first step towards practical usage for the framework as both the language of Java and the Eclipse IDE are popular and have real-world usages in the industry and the academic field. Hence ProViz is not merely a theoretical platform built for research but can also be adopted by educators and programmers for practical use.

**1.2.1 Visualizing Programs in High-Level Abstraction**

We propose that what the viewer wants to see is a high-level, algorithm type of visualizations. A program is like a scripted story where the execution follows the scripted instructions compiled from the source code. This story was formed by the program author’s mental model, which in fact gives each program procedure, module, or code segment a semantic meaning. Traditionally, such semantic information can only be conveyed through textual documentation or pictorial diagrams. This study begins with the fact that such semantic content can be best described with high-level visualization (Francik, 2002; Mukherjea & Stasko, 1993). The most prominent example of high-level visualization is the contribution of UML (Unified Modeling Language) in software engineering. UML uses abstract, graphical depictions to show how software is structured and how it dynamically operates. Its success suggests that high-level visualization is efficient and effective in facilitating comprehension.
Consequently, the visualizations that ProViz intends to create not only have high levels of abstraction but also express the semantic operations of programs. The purpose is to reconstruct the mental model out of programs with graphical representations and visual metaphors to achieve a conceptual level of comprehension. Consider the bubble sort example in Figure 2. 2(c) shows a low-level visualization made by the debugger, which simply displays variable structures and values. In 2(d) and 2(e), the high-level visualizations show that the array is a collection of numbers and that highlighted elements are the sorted ones. The purposes of variables i and j are also revealed graphically as they are used as indices underneath the array. i can also be related to the invariant – elements behind i are sorted. The levels of abstraction depart even further when time is factored in. The debugger view in 2(c) simply updates the values as the program executes, but in 2(d) and 2(e), animation is used to swap two elements to indicate the concept of exchange.

Figure 3 shows another example of high-level visualization that displays a different representation of the TreeMap object from Figure 1. Here the treeTable object is shown in the tabular form, which is the natural view of a hash table and therefore has...
higher abstraction than the tree representation in Figure 1, where tree is actually the underlying data structure of TreeMap rather than the semantic meaning of the data type.

Such high-level visualization has previously existed mainly in algorithm visualization systems, which visualize programs in a very limited scope (one algorithm per program). A true PV system offering this level of visualization for programs is difficult to make, but this is exactly the problem that ProViz aims to solve in this study.

1.2.2 Data-Driven Design and Painters

After examining different visualization techniques, we opted for a data-driven design for creating the ProViz framework. This led to the development of the painter paradigm where painters are visualizers designated to visualize the data they are assigned to. A painter can therefore be considered as a graphical function that maps a variable and its data to a certain graphical depiction. When multiple painters are created for a data type where each of them gives a different graphical representation, switching between these painters can result in displaying the same data in various forms. Such a design gives ProViz tremendous flexibility, customizability, and reusability, and is key in reducing the user’s effort in visualization creation.

Summary

Unless the benefit of PV outweighs the cost of creating it, a PV tool will still be unreal and impractical (Mukherjea & Stasko, 1993). Our intention is not only to reduce the cost of creating PV but also to increase the value of the visualization output by targeting a high-level, storytelling type of visualization in order to achieve semantic understanding for the viewer.
The prototype of ProViz is operational and has been used in an introductory programming class to demonstrate programs and algorithms visually to students. A usability test was used to evaluate its effectiveness in facilitating visualization creation.

1.3 Contribution

The design and goal of ProViz present several innovations that have never been seen in the literature or the application of PV.

Integrating Program Visualization and Algorithm Visualization

Traditionally, PV and algorithm visualization (AV) are two subfields of software visualization. They not only have different purposes but also exhibit different levels of abstraction. But in fact, algorithms are implemented within programs and can be considered as subsets of programs. When a program is visualized, its implemented algorithms will also be visualized, meaning that a PV tool can in fact fulfill the functions of AV as well. However, this has not been feasible because of the high level of abstraction required by AV. Therefore, with a PV tool like ProViz, which produces the same type of storytelling visualization as AV, one can break through the boundary between AV and PV and fulfill the goals of both areas.

Introducing the Concept of Video Documentation

The existing concept of video documentation is achieved in a forward-engineering fashion (Shilling & Stasko, 1992), but the video documentation in ProViz is a reverse-engineering process, which creates visualization from existing programs. When the visualization process is simplified, and making visualizations for programs is no more troublesome than writing textual documentation, one can create visualization for programs and use it as video documentation for these programs. This type of documentation will accurately demonstrate the program’s runtime behavior and therefore is complementary to other documentation means.

Supporting True Visual Debugging

Traditionally, visual or graphical debuggers refer to non-command-line debuggers, which use windows and some visual display (typically in tree views) to show the program
data. But such a low-level data display simply shows data values and does not visually reveal program errors. ProViz, on the other hand, accurately projects the program data structurally and behaviorally in graphical forms and program errors will be reflected in the graphical representation and seen visually. This can change the debugging activity from monitoring and comparing variable values to watching videos in search of erroneous program behaviors.

Incorporating Data-Driven Design and Event-Driven Design

Data driven and event driven are the two primary design principles for DPV or AV systems. The majority of AV systems employ event-driven design as their method, and only a few systems use the data-driven approach because a data-driven system is more difficult to implement and has been considered less powerful than an event-driven counterpart (Demetrescu, Finocchi, & Stasko, 2002; Demetrescu & Finocchi, 2006). In light of this fact, in this study we chose to use data-driven design knowing that its potential may have been underrated in the past. Taking a step further, ProViz includes some event-driven capability on top of the data-driven design and thus allows visualization to be created by incorporating both approaches, which has yet to be seen in any other PV system.

Revealing the Effect of Aliasing

Visualization is perhaps the most effective way to display the effect of aliasing, but it is rarely mentioned in the literature. As this study will point out, however, aliasing not only affects practical programming but also influences the design and output of a PV system. It forces a decision– whether the visualization will be based on variables or on objects – to be made when a PV tool visualizes a program. There will be an extensive discussion on these two paradigms in Chapter 3, which also explains ProViz’s current approach for resolving and displaying the aliasing effect.

Interaction with the Viewer

ProViz’s visualization is interactive with the viewer. The viewer can not only arrange the positions of graphical objects on screen with the mouse but also can switch
the visualization dynamically from one representation to another. This allows viewers to compose the best visual representations to suit their comprehension needs.

1.4 Brief Summary of this Dissertation

There are seven chapters in this dissertation. Chapter 2 describes the background and provides a literature review of PV, including its categorization, visualization techniques, and related systems. Chapter 3 lays out the environment for developing ProViz and the methodologies used to design the framework. It also describes two visualization paradigms that distinguish the visualization of variables and objects. Chapter 4 describes the architecture of the framework. Chapter 5 shows how the resulting framework can be put to use and visualizes a program. Chapter 6 describes the evaluation results of the framework. Chapter 7 concludes this dissertation by discussing some limitations of the framework and future research opportunities.
Chapter 2
Background and Literature Review

The purpose of software visualization (SV) is to achieve program comprehension (Jerding & Stasko, 1994; Pacione, 2004; Reiss, 2005) in order to help software developers and computer science students effectively learn about programs or algorithms by visualizing them graphically. This comprehension process is absolutely critical in the software development cycle and is the foundation for most development-related activities (Upchurch, 1997). How to facilitate and improve program comprehension has been an important subject in the field of software engineering, and the richness and expressiveness of graphical representations prompt researchers to utilize PV technology for achieving program comprehension. Dynamic PV (DPV) specifically visualizes executing programs and therefore their runtime behaviors.

DPV can be traced back to as early as 1966 with Knowlton displaying list operations on a series of films (Knowlton, 1966a, 1966b). Then in 1981, the influential “Sorting Out Sorting” video (Baecker, 1981, 1998) inspired researchers to seek means to visualize computer programs and algorithms. At that time, PV systems were mostly implemented for C/C++ languages. After Java was introduced in the mid 1990s, it became the most popular language for building PV systems because of its graphical support and its object-oriented aspect. As a result, PV research and tools have mostly been implemented in Java ever since (Shaffer, Cooper, & Edwards, 2007).

Despite high demand and interest from developers and outbursts of research and tool development, DPV has not yet been successfully adopted in programming environments, nor has creating and viewing visualizations for programs become a trend for programmers. In “The Paradox of Software Visualization” (2005), Reiss, a leading SV researcher, recognized the near-failure of PV. Reiss described the disappointed attempts to make this much-needed technology usable for the general development environment, and his work has provided the impetus for this research. Why has DPV failed? Why do we not see programmers visualizing their programs to demonstrate or
document their code? Why can’t teachers show how a program works by visualizing it on screen in the classroom?

This chapter’s discussion will show that existing PV tools either provide low-level visualization or support high-level PV of only limited scope. In addition, many high-level PV tools employ the event-driven visualization technique and therefore have usability issues that prevent them from simplifying the PV process.

2.1 User Roles

The SV environment involves three types of users: author, animator,\(^1\) and viewer (Roman & Cox, 1993; Price, Baecker, & Small, 1998). The author is the programmer who writes the program, the animator applies visualization to the program, and the viewer watches or interacts with the resulting visualization. Although these three roles and their functionalities are clearly defined, a user would in fact occupy multiple roles in most circumstances, as described below.

2.1.1 Author

The author is the programmer who writes the program to be visualized. In reality, the author is often tied to the role of the animator because of two factors: (1) how visualizations are created and (2) the purpose and the use of visualization.

First of all, an author would usually be the animator (Jeffery, 1999). A program should be visualized by someone who understands it so that the visualization produced accurately projects the program’s functionality. When a program is first written, typically only its author would have the right mental model and complete knowledge about it. If this program is to be visualized, it can only be done either by the author or by a third-party animator who needs to learn and understand the program first before she can visualize it in a sensible way.

\(^1\) Price et al. (1998) also use the term “visualizer” to refer to animator, because not all PV tools involve animation. However, in this study, “visualizer” later on refers to “painter” in the ProViz framework. To avoid confusion, we chose to use the term “animator” to describe the user role.
The second reason concerns the purpose and the use of visualization. Because visualization can facilitate many software development and maintenance tasks, such as analysis, debugging, documentation, and so forth, the author can create visualization for his programs in order to facilitate the development process.

2.1.2 Animator

The animator is the prime user of an SV tool who uses it to produce visualization out of programs. The work load for an animator thus depends on the functionality of the tool. For example, an automated SV tool does not require an animator, but creating high-level visualization for programs without a proper tool would be hard work for the animator.

Try-and-See Type of Visualization Creation

An implicit assumption is that to create sensible visualization, an animator has to understand the program well beforehand, so that the resulting visualization can describe the program accurately and reproduce effective comprehension for the viewer. However, this assumption does not hold with the try-and-see type of visualization creation. That is, a user who does not know anything about a program can progressively experiment with it by creating visualization that describes it, and the visualization can eventually help him to understand how this program works. In this case, the animator is also the viewer who attempts to comprehend the program.

On the other hand, when an SV tool provides the viewer with the option of modifying the visualization, then the viewer can engage in customizing the visualization to make it more suitable for him or others to understand. Thus, the viewer also takes on the animator’s task in this case.

2.1.3 Viewer

The viewer is the audience of the visualization. A viewer’s interaction with an SV framework would be customizing tasks to optimize the visualization to fit his comprehension needs, or the try-and-see type of interaction mentioned above. Traditionally, there are two types of viewers based on whether it is AV or PV. AV
viewers are generally students who try to learn algorithms; PV viewers are mainly software developers involved in the development environment.

2.1.4 Visualization for Different Types of Viewers

An important principle about viewers is that the ideal visualization for different types of viewers would be different. So the abstraction level should match the viewer’s knowledge about the software or his technical background (Bednarik, Moreno, Myller, & Sutinen, 2005). For example, visualization for program authors or peer programmers can be very detailed and exhibit low abstraction on specific tasks or algorithms; for students, the visualization may be high in abstraction, using more visual clues to convey the concepts instead of showing the absolute detail; visualization for clients could display at an architectural level for simple understanding and verification. As a consequence, visualization should be constructed and customized based on the perception level, the background knowledge, and the needs of its intended audience.

2.2 Categorization of Software Visualization

Price et al. (1998) categorized SV into two subfields:\(^1\) program visualization and algorithm visualization. PV is considered a software-engineering focus, which aims to assist software developers in developing and maintaining their programs. AV, on the other hand, is a subject for the teaching environment, especially pedagogical purposes. Algorithms are visualized in order to teach entry-level computer science students to learn about these algorithms (Mukherjea & Stasko, 1993). Traditionally, these two areas of SV not only serve different purposes but may also differ in their implementation methods. However, this categorization may be due to the lack of a comprehensive tool that can produce high-level visualization for programs. Although high-level PV systems, such as the object view in jGRASP, may implicitly combine the two areas (Hendrix, Cross, & Barowski, 2004), they still do not reach the storytelling level of abstraction of AV

\(^1\) Price et al. (1998) described three additional categories of SV: visual programming, programming by demonstration, and computation visualization (Stasko & Patterson, 1992). The first two are not related to the purpose of this study and therefore will not be discussed. As for computation visualization, we consider it a special type of PV that is not in the scope of this study.
systems. Nevertheless, the difference between the two fields will be examined below, and we will also describe how a PV tool can eliminate the boundary between PV and AV.

**Program Visualization**

In PV, it is a software program that is visualized (Jerding & Stasko, 1994). One cannot create visualization for a program separately from a program and claim it to be a PV process. The concept of PV can be as simple as visualizing some aspect of a program graphically – from program source code or the program’s dynamic behavior during execution to the architecture of a software system. The results are a variety of PV tools with different focuses and representation styles. Several taxonomies were proposed in the 1990s to categorize PV systems (Roman & Cox, 1993; Price, Baecker, & Small, 1998; Myers, 1990; Stasko & Patterson, 1992; Brown, 1988); these will be used selectively to describe the categorization in this section.

PV can be split into two major types: static PV and dynamic PV (Diehl, 2007). This distinction of static and dynamic for PV is not based on the type of visualization output but on the content of the program that is visualized (different researchers have used different terms: Roman and Cox’s scope [1993], Price et al.’s [1998] and Myers’ [1990] content, and Stasko and Patterson’s aspect [1992]).

### 2.2.1 Static Program Visualization

Static PV visualizes the information in the static portion of a program (i.e., the program source code). Examples of static PVs include pretty printing, syntax coloring and typography, UML diagrams of class structures and relationships, control flow graphs (e.g., flowcharts), method call graphs, views of software architectures, and so on (Diehl, 2007). All of these are static information that can be obtained from the source code.

In static PV, the data source is static, and thus the visualization output is also static, meaning that it does not change over time unless the source is modified. Generally, extraction and analysis of static data are relatively easy to achieve compared to the same

---

1 Myers (1990) did distinguish static and dynamic PV based on the output style.
processes in DPV, and the output presentation tends to have a fixed form that is specific to the data. Therefore, static PV tools have been well developed and have played a prominent role in software development. However, as useful as static PV may be, it cannot reveal the true behavior of a program (Reiss, 2003; Jerding & Stasko, 1994; Shilling & Stasko, 1992), which is only exposed when the program actually runs.

2.2.2 Dynamic Program Visualization

DPV visualizes the dynamic portion of a program – the program execution. Hence, DPV can be achieved only after the program has started executing, and the data to be visualized is called the runtime data. In contrast to static source code, the runtime data has the dimension of time where the data changes and updates dynamically, and thus its visualization also evolves over time. Therefore, the visualization output of DPV can be like a slideshow where each slide represents an update on the graphical representation of the data. Animation can be used to further emphasize graphical changes; hence, the term program animation (Kerren & Stasko, 2002) is used to describe visualizations that animate the progress of a running program.

Figure 4 – Static PV versus dynamic PV

Figure 4 shows the difference between static PV and DPV. In contrast to static PV where visualizations are relatively easy to produce, DPV is more difficult to generate because the runtime data is dynamic and exists only temporally in memory. So the first challenge of DPV is to get a hold of the data. Typically, this requires a program monitoring mechanism to monitor and extract the runtime data for visualization. Another approach is to record the execution trace of a program and then visualize the recording
data. These two approaches differ in the time factor of when the visualization can be generated and displayed. Therefore, based on the timing of visualization, this study distinguishes three types of DPV: live, real-time, and post-mortem.

**Live PV**

Live PV displays the visualization in-sync with the program execution. As the target program executes step-by-step, live PV also visualizes it step-by-step alongside the execution. Thus, in live PV, the target program’s execution is significantly slowed down to match the visualization’s speed so that humans can view the output. An example of a live-PV tool is the debugger, where the program is suspended whenever the user chooses to pause the execution and view the program data.

The difference between live and real-time PV is the execution speed of the target program. Live PV has no restriction on the speed as long as the visualization is viewable to humans. In fact, the time it takes for the viewer to look at a visualization scene in one execution step is astronomically long compared to the program’s original execution speed (seconds versus nanoseconds), especially when animations are involved.

**Real-Time PV**

In contrast to live PV, the goal of real-time PV is to not affect the target program’s execution speed (or to do so in the minimum degree) so that the program can run as close to its original speed as possible. The purpose is to (1) monitor the real performance of the target program and (2) be applicable to real-world systems, which should not be slowed down just for the purpose of visualization. Hence real-time PV must function discretely in certain time intervals and use minimum efforts to gather information so that the target program is not significantly slowed down. High-level visualizations or animations are therefore impossible for such systems, and real-time PV systems usually belong to the category of computation visualization where performance metrics are collected and displayed for analysis. Reiss’s JIVE and JOVE are the only real-time PV systems that visualize program data in slightly higher abstraction, displaying information about execution time, memory, methods, and threads using abstract views (Reiss & Renieris, 2005; Reiss, 2003; Reiss, 2006; Reiss, 2007).
Reiss proposed that a useful visualization system needs to be real-time (Reiss, 2003). While this is true for computation visualization tools, it cannot be applied to high-level PV tools that demonstrate program behaviors and algorithms using animation, because if the visualization were to update in real time (nanoseconds), the viewer would not be able see it. In addition, real-time PV displays in intervals, which is unfeasible for program comprehension because what happens between two intervals cannot be revealed, potentially allowing important changes of the program to be omitted.

Post-Mortem PV

Post-mortem PV records the runtime data during the program execution. After the program terminates and the recording process completes, the recorded log file can then be played back and visualized. While this is a straightforward process, the implementation of such a system always suffers from one problem: size of the recording file, which can easily become very large (Narayanasamy, Pokam, & Calder, 2005; Orso, Jones, & Harrold, 2003). Huge recording files can also make post-mortem analysis or visualization a challenging process, and so filtering is a necessary step to remove unnecessary data in order to reduce the size of the recording.

Video Documentation

Despite the drawback of the recording size, only a post-mortem system can be used to create video documentation. When the recording can be played back and visualized, it essentially becomes a video file, which is ubiquitous and can be used to document the program. Live or real-time PV systems can in fact incorporate the post-mortem feature alongside their original visualization schemes.

2.2.3 Algorithm Visualization

AV refers to the visual, graphical representation of algorithms. The most basic form of AV can be easily found in computer science textbooks, where sequences of figures and diagrams show graphically how algorithms work. This type of visualization is categorized as static AV. At today’s technology level, the more complex form of dynamic AV (Price, Baecker, & Small, 1998) is usually more desirable than static pictures. Dynamic AV essentially displays algorithms on screen as motion pictures, where steps of
an algorithm can either be displayed slide-by-slide or using animation. Therefore, dynamic AV produces visualization output that is essentially like a slideshow or a movie clip. When animation is used in AV, it is also called algorithm animation (Kerren & Stasko, 2002).

Whether static or dynamic, the purpose of AV is to let the viewer understand an algorithm, to learn about how it works and what happens in-between the steps of the algorithm. Essentially, AV attempts to achieve the storytelling type of understanding for its viewers so that they can learn about how an algorithm operates. Therefore, the visualization output of AV always exhibits a high level of abstraction, which can convey the semantic content and procedures of an algorithm. As a result, the major application of AV is in the academic field, where computer science students learn about algorithms visually and instructors use AV to enhance their pedagogical effort.

**Source of AV**

The biggest difference between PV and AV is the source of visualization. While PV requires a concrete software program as the source, the source of AV is abstract algorithms, which means that without a concrete data source, the visualization of AV only needs to conform to its creator’s vision. Therefore, AV can be created by any graphic creation tool or method, from hand drawing or videotaping to software like Adobe Flash. This type of AV creation has no software program involved; instead the visualization is simply a result of the imagination of its creator.

**Achieving AV through PV**

Another way of making AV is through PV. In fact, traditionally, many consider algorithm animation a type of PV (Diehl, 2007). But this study supports the view that only when the source of visualization is a program can it be categorized as PV: Not all types of algorithm animation are PV, but only those whose visualization sources are programs.

Because programs employ algorithms, if the portion of a program that involves an algorithm is visualized, the resulting visualization will specifically display this algorithm, which makes it an AV production even though it is produced out of a program. This is
where the distinction between AV and PV is blurry. It seems that the purpose of such visualization is for displaying an algorithm (AV), but the actual technology used is through visualizing a program (PV). Hence, an effective PV tool would not be bound by this categorization if it could produce an AV level of visualization for programs. Furthermore, if one is able to visualize a program easily, it means that one could visualize any algorithm embedded within this program as well. Therefore, such a high-level PV tool would serve as a generic SV tool, crossing the boundary between AV and PV.

What a PV tool can achieve even beyond normal AV is that, traditionally, there exist only visualizations for popular algorithms or data structures (Shaffer, Cooper, & Edwards, 2007). But in fact, each and every software program employs its own unique algorithms to solve its particular problems, which means that there are an enormous number of algorithms embedded in all programs in the world, but only a small fraction of them is covered in the existing AV set. If an effective PV tool could visualize any arbitrary algorithm in existing programs, AV would no longer be limited by the existing set of AVs.

Furthermore, the techniques that Demetrescu et al. (2002) discussed for creating AV actually produce visualization out of programs. So in fact, judging by the source of the visualization, what they described as techniques for creating AV are actually techniques for creating PV, even though their purpose was to show the visualization of algorithms. These visualization techniques will be discussed in Section 2.4.

Summary

Terms in the areas of DPV and AV, such as program visualization, program animation, and algorithm animation, have often been used interchangeably in the past. This study provides clear distinctions between these terms, and now lists the differences between DPV and AV in Table 1.
Table 1 – A comparison between DPV and AV

<table>
<thead>
<tr>
<th></th>
<th>Dynamic Program Visualization</th>
<th>Algorithm Visualization</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Objective</strong></td>
<td>To show information about a running program</td>
<td>To show how an algorithm works</td>
</tr>
<tr>
<td><strong>Source</strong></td>
<td>A concrete software program</td>
<td>An abstract algorithm</td>
</tr>
<tr>
<td><strong>Creation Means</strong></td>
<td>A designated DPV tool</td>
<td>Any graphical tool (indirect) or a PV tool (direct)</td>
</tr>
<tr>
<td><strong>Presentation Abstraction</strong></td>
<td>Low (compared to AV)</td>
<td>High</td>
</tr>
<tr>
<td><strong>Presentation Style</strong></td>
<td>Fixed views, such as sequence diagrams, trees, UML-like diagrams, etc.</td>
<td>Slideshows or movies demonstrating the steps of an algorithm</td>
</tr>
<tr>
<td><strong>Target Audience</strong></td>
<td>Software developers</td>
<td>Computer science instructors and students</td>
</tr>
</tbody>
</table>

The above distinctions would be obsolete if a DPV tool is able to support high-level, AV type of visualization creation for programs. Because of the fact that algorithms are embedded within programs, such a DPV tool will also be able to address the field of AV.

### 2.3 Abstraction

Roman and Cox (1993) describe abstraction as the kind of information conveyed by visualization, and we also use this understanding of the term when we refer to visualizations with high or low abstraction throughout this paper. However, abstraction does not apply merely to the output presentation in PV literature. For example, Pacione (2003, 2004) described six abstraction levels of PV systems based on the type of data that is visualized. We want to clarify the use of the term “abstraction” by dividing it into two different categories: *data abstraction* and *presentation abstraction*. 
2.3.1 Data Abstraction

Data abstraction refers to the abstraction of the specific content of a program (Roman and Cox’s scope [1993], content by Price et al. [1998] and Myers [1990], and Stasko and Patterson’s aspect [1992]). Even before a certain type of data is visualized, it will have a level of abstraction that is naturally perceived by humans. In other words, data abstraction is fixed for a particular type of data, but when this data is visualized in different ways, the visualizations can have different levels of presentation abstraction. Based on the state and the content of a program, we categorize six levels of data abstraction; these are given below in order from low to high:

1. Machine (e.g., CPU registers, RAM)
2. Intermediate code (e.g., executable code, Java byte code)
3. Program source code
4. Method calls/message passing
5. Classes/objects
6. Software architecture

Information Load

Data abstraction is the inverse of information load. In other words, data with lower abstraction will contain more information, and vice versa. There is a gradual loss of information as the abstraction goes up. For example, the source code contains everything that is needed to construct all higher level data, but one cannot reconstruct the source code using higher level data such as method calls or object diagrams. This loss of information for high-level data is necessary for specific comprehension needs because low-level data may contain too much information. For instance, it is extremely difficult to form high-level programming concepts by analyzing the binary data flow in CPU and its registers. Therefore, high-level data are extracted from data with low abstraction through interpretation or analysis.

Natural Representation

The abstraction level of a type of data will determine how its representation is usually perceived by humans. For instance, the natural view of CPU registers or RAM
data is in binary digits, program source codes use characters and symbols, and arguably, the most common view of classes is through UML-type of diagrams. Notice that the abstraction of a datum’s natural representation corresponds to its level of data abstraction. Therefore, to visualize a type of data in an abstraction that is different from its natural representation would require more work in data analysis and extraction. This will be further elaborated in the next section.

### 2.3.2 Presentation Abstraction

Throughout this dissertation, when abstraction is mentioned, it refers to presentation abstraction and not data abstraction. Presentation abstraction is indeed what Roman and Cox (1993) meant by “abstraction” in their taxonomy (and is equivalent to Brown’s content [1988] and Stasko and Patterson’s abstractness [1992]). It is the abstraction of the visual output that is perceived by humans and therefore is a relative measure that may be perceived differently by viewers. Hence we will only use very general and obvious examples to differentiate and discuss the levels of presentation abstraction.

**Data Abstraction vs. Presentation Abstraction**

Unlike data abstraction, presentation abstraction is not fixed by the content of the data. The same type of data can be visualized in many ways, and each visual representation will possess its own level of presentation abstraction. For example, binary data can be presented as a stream of 0’s and 1’s (which conforms to its data abstraction); it can be converted and represented in decimal or hexadecimal format, in strings, or in some graphical notation that represents the meaning of the data. The latter representations have higher abstraction than the 0’s and 1’s, and yet they cannot be produced as easily because some kind of conversion, interpretation, or analysis is necessary to produce the high-level representation. In fact, the further the presentation abstraction is from the data abstraction, the more difficult it is to construct the visualization because the information loss in higher level data would require more filtering of the low-level data. For example, a method call graph can be created by analyzing the source code. If one is to generate the same call graph directly from the intermediate code or the machine code, much more effort will be needed to interpret and analyze the low-level data.
Unfortunately, this is the reason why high-level visualizations are rare and difficult to make in PV. Because the program source code and the runtime data have relatively low levels of data abstraction, visualizing them in high-level, AV type of abstractions is very challenging and time-consuming.

2.4 Dynamic Program Visualization Techniques

The goal of DPV is to create visualizations from a program that accurately describe this program’s runtime behavior. Diehl (2007, pp. 97–98) described existing techniques (aka specification methods) for creating algorithm animation. As mentioned in Section 2.2.3, some of these techniques produce visualizations from programs and hence can be considered DPV techniques.

- **Ad hoc**: A program is written specifically to show the visualization without using a tool. This is the most tedious and laborious approach because everything, including the algorithm and the visualization code, is written from scratch.

- **Special data types**: Data types with built-in visualization are used to construct a program, and the visualization will display. The downside of this approach is implicit code intrusion – these data types are specially made for visualization purposes, and this prevents them from being used in real programs. Furthermore, programs that are not created with these special data types cannot be visualized, which makes them unsuitable for visualizing existing programs.

- **Post-mortem**: As discussed in Section 2.2.2, post-mortem systems record program executions or traces to files and play them back. Because of the recording characteristic, it is a technique that can be used in conjunction with other DPV techniques.

- **Semantics-directed**: This technique is similar to automatic visualization, where the program is displayed automatically in low abstraction. This will be discussed in more detail in Section 2.4.3.
The above methods, except for post-mortem, can hardly be adopted by a PV system. Instead, two other techniques are more commonly employed in PV systems: event driven (aka interesting events) and data driven (aka state mapping) (Kerren & Stasko, 2002; Demetrescu, Finocchi, & Stasko, 2002; Diehl, 2007; Brown & Sedgewick, 1998). As mentioned previously, these techniques are used for the purpose of high-level AV instead of PV, but because the visualization source is a software program, they can be considered as DPV techniques.

2.4.1 Event Driven (Interesting Events)

The event-driven technique was pioneered by the BALSA (Brown & Sedgewick, 1984) AV system in 1984 and later adopted by a series of AV systems (Demetrescu, Finocchi, & Stasko, 2002). The concept of the event-driven approach is simple, but the problem lies in its application. Given a target program to visualize, one needs to identify interesting points in the program where the visualization should take place, compose graphical routines, and insert them at these points to create visualizations. Then the visualization is produced by merely running the program. No additional tool is necessary in this technique, which makes it much simpler than the data-driven approach. However, the drawback is in its application, as the following issues pose great obstacles to this approach:

1. How can interesting points in the program be identified? For programs with moderate size or complexity, this would already pose a problem (Demetrescu, Finocchi, & Stasko, 2002).
2. Most graphical routines need to be composed individually to respond to their specific events. In other words, they are context-specific and offer little reusability when compared to the visualizers in the data-driven approach.
3. Code intrusion (invasiveness) – the event-driven approach causes a severe level of code intrusion to the original program (see Section 2.5).
4. Code ignorance allowance (Demetrescu, Finocchi, & Stasko, 2002) – one must understand the program before she can create a visualization for it using the event-driven approach. Therefore, the event-driven approach does not support the try-and-see type of visualization creation.
These issues with the event-driven approach are detrimental to its usability and are very difficult to resolve or simplify (Roman & Cox, 1993).

2.4.2 Data Driven (State Mapping)

The data-driven approach maps program states into their graphical representations. Each program data $x$ that is to be visualized is mapped to its pictorial form by a visual function $P(x)$. At any given time during program execution, the visualization is produced based on the state $x$. It is data driven because subsequent changes of $x$ create updates on the visualization from $P(x_{t1})$ to $P(x_{t2})$. The most prominent example of a data-driven system is the debugger, where values of the program data can be displayed at any given step of the execution.

In contrast to the simplicity of their event-driven counterparts, data-driven systems generally require additional tools in addition to a graphical framework because they need to acquire the runtime data. Diehl (2007, p. 12) presented a visualization pipeline: Data Acquisition $\rightarrow$ Analyses $\rightarrow$ Visualization. The first step of the process is obtaining the data. Some form of a program monitor is usually required in this step to control the program flow and retrieve the runtime data. Further down the pipeline, the acquired data is analyzed and mapped to visualization.

Visual Mapping

From the user’s point of view, what determines the characteristic of a data-driven PV system is the process of defining the mapping between data and their visual functions. The visual mapping process is necessary because a program’s execution consists of a large amount of data, typically only a fraction of which are of interest to the animator or the viewer. Therefore, to specify what data should be visualized and what visual functions should be associated with them will be the first task for the user. In many systems, this is accomplished using the declarative approach (Demetrescu, Finocchi, & Stasko, 2002; Diehl, 2007; Roman, 1998), which lets the animator specify such mapping through a certain mechanism in the source code. On the other hand, automatic visualization systems do not need the user to specify the mapping but may predefine the data to be displayed.
The Declarative Approach

The declarative approach lets the animator declare mapping between data and visualization in the source code, and in doing so it causes code intrusion. However, unlike event-driven PV, whose graphical routines are executed as part of the program, declarative systems typically use mechanisms like comments or annotations as their instrumentation method, which reduces the intrusion to compile level (see Section 2.5). The mechanism for declaring the visual mapping can be a program written in a specific language (e.g., LEONARDO [Demetrescu, Finocchi, & Stasko, 2002] and InspectJ [Khaled, Noble, & Biddle, 2003; Khaled & Noble, 2003]) or method calls to respond to data changes (e.g., Daphnis [Francik, 2002]).

Reusability

An advantage of state mapping over the event-driven approach is the reusability of the visual function. Once a type of data’s graphical representation is defined with a visual function, any instances of this data can be visualized with this function, whereas the visualization of events is highly context-specific and depends on the semantics of the program where the events occur.

Limitation

Not only does data-driven PV need more complex components for visual mapping and data extraction but also it has been considered more limited in generating visualizations (Demetrescu, Finocchi, & Stasko, 2002). This is due to the fact that the data-driven process depends solely on the change of data to update the visualization, which means that it cannot detect program processes where the data do not change, such as the control statements or certain operators. On the contrary, the event-driven approach can insert event calls virtually anywhere in the source code and therefore has higher granularity on the visualization.
2.4.3 Automatic Visualization

Automatic visualization\(^1\) (Kerren & Stasko, 2002) refers to automatic generation of visualization from programs (Roman and Cox [1993] call this predefinition, and Diehl [2007] calls it semantic-directed). Automation is actually the resulting characteristic of a PV system, in contrast to the above “methods” for specifying how to visualize programs. In fact, automatic visualization systems must employ other visualization techniques under the hood to produce visualization. Yet it can be considered as a technique from the user’s perspective because for an animator, automation means that virtually no effort is required to generate visualization.

**Automation Is the Reverse of Abstraction**

However, the problem with automatic visualization is that generating storytelling, high-level visualization automatically is currently impossible (Mukherjea & Stasko, 1993). Such systems generally can only produce low-level visualizations (Kerren & Stasko, 2002; Diehl, 2007) or high-level visualization in fixed, predetermined views. Therefore, automation is considered the inverse of abstraction for a PV tool (Francik, 2002). The debugger is a prime example of such a system, where the runtime data can be displayed automatically during execution. But the debugger can only display variables and their values in basic representation – texts. Some systems, such as JIVE (Gestwicki & Jayaraman, 2005), JAVAVIS (Oechsle & Schmitt, 2002) and the Jeliot family (Moreno, Myller, Sutinen, & Ben-Ari, 2004; Ben-Ari, Myller, Sutinen, & Tarhio, 2002) do provide automatic visualizations at higher levels, and yet they can only display fixed program mechanics such as object creation, method invocation, certain data structure views, and so on, instead of the algorithms or concepts of programs.

2.5 Code Intrusion

Code intrusion (Price, Baecker, & Small, 1998, p. 20) refers to the alteration of a program’s source code. It is a very important factor that influences whether a tool will be

---

\(^1\) Originally termed “automatic animation.”
adopted and used in practice. If a tool required developers to modify their source codes in order to achieve something, there would be a risk of introducing errors in the process (Khaled, Noble, & Biddle, 2003). In addition, code intrusion adds extra content to the source code, which reduces one of the most important properties of a program – readability. For these reasons, either the benefit of a tool should greatly outweigh the cost of modifying the code (as with JavaDoc), or it should have as little code intrusion as possible.

Based on the level of changes that affect a program’s behavior, we define four levels of code intrusion as shown in Table 2:

<table>
<thead>
<tr>
<th>Behavior (high)</th>
<th>The functionality/behavior of a program is changed(^1) when it runs. Event-driven PV belongs to this level.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Debug (medium)</td>
<td>For debug purposes only. Output or logging routines (such as <code>System.out.println()</code>) are added to display values for debugging. This changes the behavior of the program in the minimum degree.</td>
</tr>
<tr>
<td>Compile (low)</td>
<td>At this level, the modification to a program’s source code will not alter its behavior. The use of comments (e.g., JavaDoc) and Java annotations belongs to this level.</td>
</tr>
<tr>
<td>No Intrusion</td>
<td>There is no alteration on a program’s source code or its intended behavior.</td>
</tr>
</tbody>
</table>

**Behavior Level**

From Table 2 it can be seen that event-driven PV inserts custom-made graphical routines into the program. These hard-coded routines are compiled and run with the program, which changes the program’s original behavior. Any error in these added routines can be a serious problem.

\(^1\) We do not consider the added time or delay caused by the visualization a modification of the program behavior as long as it does not alter the program’s execution path. A debugger, for instance, does not change the program’s behavior but merely slows it down.
routines will also become the program’s error. Such an effect may not be serious to the program when applied with caution – such as making a copy of the program source code in order to avoid any possible damage to the original code. This effect, however, decreases the usability of such a PV tool, as well as its reusability, because graphical routines must be taken out in order for the program to function or to be used as a library.

**Debug Level**

The debug level of code intrusion occurs frequently during the programming process. It involves inserting output routines into the program for debugging purposes, but this is done on a small, manageable scale. Output routines would only be deployed in suspicious areas in the code, and when problems are solved, these routines could be taken out. Plain output routines generally do not change the behavior of the program, and yet they still are hard-coded to the original program. Furthermore, many post-mortem PV systems operate at this level by outputting execution traces to files.

**Compile Level**

Outside of event-driven tools, most programming development tools operate at the compile level. More specifically, many use comments as their instrumental method to accomplish their tasks (Demetrescu, Finocchi, & Stasko, 2002). Java, since Version 5, specifically added Java annotations to support such tools. Neither comments nor annotations change the behavior of the program, and this is the advantage of compile level. Nevertheless, at this level, the amount of text being inserted to the source code is generally large for the purpose of PV. This clutters the code and reduces readability of the program.

**No Intrusion**

Low-level PV tools and automatic visualization systems typically visualize without code intrusion because their visualizations are in a predetermined format and do not require customizing information. However, it is uncommon to create high-level PV or AV without code intrusion because some form of a mapping mechanism will be additionally required for animators to associate program data/events and the visualizations.
A declarative PV tool would arguably be best without any code intrusion. Such an approach is not absolutely the best because it may need to externalize the mapping between what is in the source code and visual functions, but when this information is externalized, it would become difficult for the animator to correlate it to the elements in the source code. Nevertheless, if a high-level, intrusion-free PV tool can improve the user’s awareness of the mapping information, this would be the optimal setting.

2.6 Related Work

Pavane (Roman, 1998; Roman, Cox, Wilcox, & Plun, 1992) is the earliest AV system to adopt the declarative, data-driven approach. Using a rule-based, declarative approach to map program data into visualization, Pavane can produce 3D algorithm visualizations. It utilizes its own runtime environment for the Swarm logic programming language and can visualize Swarm and C/C++ programs. LEONARDO (Demetrescu, Finocchi, & Stasko, 2002; Demetrescu & Finocchi, 2006) is an integrated development environment for C programs. It uses a declarative language, ALPHA, for specifying the mapping between program data and their visual representations and therefore is data-driven. ALPHA codes are declared within the comment blocks in the source code. Although such an approach only processes compile-level code intrusion, it does introduce a large quantity of code into the source code. PAVI (Ohki & Hosaka, 2003) uses a similar approach to produce 3D visualization, which focuses on assignment operations.

jGRASP (Hendrix, Cross, & Barowski, 2004; Cross, Hendrix, Jain, & Barowski, 2007) is a lightweight integrated environment for Java. Its debugger contains object viewers for visualizing data structures during the debugging session. This study considers jGRASP and its object viewers to be the system closest to ProViz, with the difference being that each object viewer visualizes a variable or object and there is no correlation between different views. Therefore, the visualization displays no semantic meaning about the program behavior or algorithm. Nevertheless, the high-level views of data structures in jGRASP have proven to be a valuable tool. jGRASP has made it clear that people desire to see their programs at a high level of abstraction, and the simplicity in using these views in jGRASP makes it an excellent example of a usable PV tool.
Jeliot 3 (Moreno, Myller, Sutinen, & Ben-Ari, 2004) is a standalone application for visualizing Java programs. It has designated areas for displaying methods, objects or arrays, constants, and expression evaluations. It performs automatic visualization by using a Java language interpreter, and animations are used to keep the viewers in focus and show the program’s progress. The abstraction of Jeliot 3 is quite high as it is able to express the semantic meaning of some operations. For example, the visualization can associate an integer with an array when the integer is used as an index of this array, as shown in Figure 5. The capability of expressing such high-level meaning about programs is what this study is aiming for, but as an automatic visualization tool, Jeliot 3 can only visualize a limited set of default language semantics and not custom data types or actions. Furthermore, the problems with using the interpreter are that (1) it may not be compliant with all data types in the Java library or custom-made classes; and (2) the interpreter is not updated alongside the language, which makes it obsolete with new language syntax (such as generic types in Java for Jeliot 3) and programs.

```java
public class MyClass {
    public static void main() {
        int[] array = new int[10];
        for (int i = 0; i < array.length; i++) {
            array[i] = i;
        }
    }
}
```

![Jeliot 3 visualization](image)

**Figure 5** – Jeliot 3 is able to display the relationship that a variable is used as an index of an array

The Daphnis (Francik, 2002) algorithm visualization system also uses the declarative approach to associate variables to visualizations, but unlike LEONARDO, which creates the mapping in comment blocks, Daphnis uses function calls to register
data, which results in the behavior code intrusion. A Petri net formalism is used to describe algorithm behaviors so that unnecessary temporal and spatial information of the animation can be automatically suppressed. Data changes, however, have to be manually identified and annotated with Play routines by the animator because the program is not monitored.

Noble and colleagues’ older system Tarraingím (Noble, 2002; Noble, Groves, & Biddle, 1995; Noble & Groves, 1991) and their newer InspectJ (Khaled, Noble, & Biddle, 2003; Khaled & Noble, 2003) are both object-oriented PV tools that can visualize programs in high-level abstractions and also produce visualizations in multiple abstractions. Tarraingím visualizes programs written in the SELF programming language, and InspectJ visualizes Java programs using the aspect-oriented language AspectJ. Therefore, InspectJ is in fact quite similar to ProViz, but its visualization process is not as straightforward and requires the animator to learn about AspectJ. The program monitoring of InspectJ is achieved by declaring pointcuts for the target program, where events should occur at these points. This makes it similar to the approach of Daphnis and in fact, the event-driven approach.

ViLLE (Rajala, Laakso, Kaila, & Salakoski, 2007) is a PV tool designed specifically for the educational environment. It supports different languages by using an interpreter that parses and runs the program based on a syntax pool, which by default contains syntax for Java, C++, and a user-defined pseudo language. A syntax editor allows the user to edit existing languages or add new languages. ViLLE also has facilities to let the instructor define questions associated with program execution. An evaluation showed that as an educational tool, ViLLE benefits primarily novice students (Rajala, Laakso, Kaila, & Salakoski, 2008).

GROOVE (Shilling & Stasko, 1992) is a graphical program specification and design editor. It allows the user to design classes and methods through manipulating graphical objects in the editor, and animations can be created and stored as scripts. Meanwhile, GROOVE can generate automatic codes according to the user’s design, and these codes can be used for real development. Then when programs created in such a way
execute, GROOVE can hook the designed visualization to the code to produce runtime PV. In addition, GROOVE introduced the concept of video documentation: the designed animation can also be used as documentation for the produced program. This type of video documentation, however, is the product of a forward-engineering process, meaning that (1) it is unable to visualize existing programs and (2) the graphical design for code generation could be overshadowed by commercial visual programming tools.

Many PV systems are integrated environments that provide multiple views simultaneously to display the state of a program. Some examples are: FIELD (Reiss, 2007), BLOOM (Reiss & Renieris, 2003), JIVE and JOVE (Reiss, 2007), Jinsight (De Pauw, Jensen, Sevitsky, Vlissides, & Yang, 2002), JIVE (Gestwicki & Jayaraman, 2005; this is distinct from Reiss’s JIVE) and a newer version as an Eclipse plug-in (Czyz & Jayaraman, 2007), JAVAVIS (Oechsle & Schmitt, 2002), GAMMATELLA (Orso, Jones, & Harrold, 2003), EVolve (Wang et al., 2003); each of them has a variety of sequence diagrams, object or reference diagrams, method/stack views, and other forms of computation visualization to collectively support the coding and comprehension process. These are visualization systems that provide fixed views, as described earlier, and so they do not support AV-type of visualization creation.

Event-driven PV systems, on the other hand, are considered AV systems as they are used to generate high-level, storytelling visualizations that are not fixed to some predefined format. Starting with BALSA, there were a string of event-driven systems: ZEUS (Brown, 1992), Polka (Demetrescu, Finocchi, & Stasko, 2002), TANGO (Stasko, 1990), and JCAT (Najork, 2001). TANGO is especially noteworthy because it alleviates code intrusion by providing an editor for adding event routines without directly changing the source code. However, the source code is still modified before the program runs by inserting the assigned graphical routines, so there will be intrusion on the program behavior even though the user will not be aware of such modification.

In terms of the capability of creating user-defined AV, Pavane, LEONARDO, and Daphnis are the only data-driven systems that use the declarative style to accomplish custom mapping. In terms of the visualization environment, the output style, and the
practical usage, however, this study considers jGRASP as the system closest to the ProViz framework.
Chapter 3
The ProViz Framework

In light of the fact that existing DPV tools are either low-level or are limited in the scope of program they visualize, this study proposes ProViz, a DPV framework that aims to support rapid creation of storytelling, AV-level visualization on software programs. Although the primary goal for ProViz is to generate high-level, AV-type visualization, ProViz is not limited to any particular form of visualization. Fixed visual representations, such as message passing, sequence/object diagrams, flowcharts, and other well-known DPV views are already available in other tools. ProViz, on the other hand, essentially offers its animator a blank canvas with predefined, customizable visualizers; based on the content of the target program, the animator can visualize the program freely with his imagination and creativity.

The prototype of ProViz is implemented in Java and is integrated as a plug-in to the Eclipse IDE. The objective is to provide tools and develop visualization paradigms to simplify the laborious process of visualization. Two key factors used to bring ProViz closer to its goals are the data-driven design and the object-oriented painter paradigm. With such a design, ProViz has exceptional customizability and reusability in creating visualization, which is crucial to reducing the effort and time of the animator.

Categorizing ProViz

The categorization of ProViz is shown in Figure 6. ProViz is a PV tool because it creates visualization for software programs. Yet its AV-like, high-level output and the fact that algorithms are implemented within programs make it an AV tool as well. In other words, it does not need to be bound by the distinction between PV and AV. Furthermore, because

![Figure 6 – Categorization of ProViz in software visualization](image-url)
ProViz’s goal is to visualize running programs and their behaviors, it also belongs to the DPV category. As stated in Section 2.2.2, a high-level DPV tool cannot function in real time, and therefore ProViz is a live PV tool. In addition, it incorporates the post-mortem feature by recording the data, and a designated player can play back the recording. Being post-mortem is an important feature in achieving the goal of video documentation because the recorded video needs to be independent of the program. What is worth mentioning is that the current playback of ProViz’s recording preserves the same interactivity as when the program is visualized live. This will be elaborated upon in Section 5.7.3.

3.1 Target Users

As mentioned in Section 2.2, AV is primarily used in the educational setting, whereas PV is used in the industry for software engineering purposes. As ProViz supports both AV and PV, its target audience will in fact span both areas. However, because visual support for program comprehension and video documentation has yet to become commonly available in the industry, the impact of such a PV framework on general software developers may have unforeseen results. Nevertheless, this section will discuss the target users of ProViz based on the environment and the user roles of animator and viewer.

Academia

In educational settings, ProViz can be used by instructors to create visualizations in order to dynamically demonstrate programs and algorithms so that students can visually learn about how they work. Students, on the other hand, can also actively create or customize visualizations for learning or debugging. In addition to the AV systems mentioned in the previous chapter, there are also AV tools that are built for this purpose, such as Alice (Cooper, Dann, & Pausch, 2003), BlueJ (Kouznetsova, 2007; Kölling, Quig, Patterson, & Rosenberg, 2003), and ALVIS (Hundhausen & Douglas, 2000; Hundhausen & Douglas, 2002), but these tools are graphical programs rather than PV tools and are mainly used to teach CS1 – the first class in computer science. With ProViz, however, students of all levels can utilize and customize visualizations for their programs;
instructors in high-level classes can also use it to produce visualizations for programs with more complexity.

**Industry**

Any programmer can be an animator and use ProViz to make visualizations for programs for the purposes of debugging, documentation, and program comprehension. ProViz’s target users would be all personnel involved in a software development environment, including programmers, designers, project managers, and even clients. Anyone who needs to understand or work with software through visual documentation or presentation is the target user.

### 3.2 Language

ProViz is implemented in Java and visualizes Java programs. The Java programming language is chosen for three main reasons:

1. It is an object-oriented (OO) programming language, where the OO paradigm makes it natural for objects to be visualized (Jerding & Stasko, 1994).

2. Its popularity and importance in both industry and academia can make the framework more useful and applicable in both areas.

3. Java is known for its abundant API, including the GUI library (AWT and Swing), which is familiar to many Java programmers.

ProViz utilizes one of the most popular programming languages and does not define any new language, as utilizing new languages may increase the learning curve for adopting the framework. Therefore, ProViz visualizes Java and the prototype uses the Java standard graphical library for visualization. This encourages instant usage for the majority of Java programmers without the need for learning a new language.

### 3.3 Environment

ProViz is implemented as a plug-in to the Eclipse IDE because it utilizes many components in Eclipse, including the debugger and several views, to facilitate both creation and viewing of the visualization. Instead of creating these components from
scratch, it is certainly reasonable to integrate with an IDE that already has these available. In addition, because Eclipse is free and open source, ProViz can be adopted without any expense to the user.

### 3.3.1 Debugger

As mentioned previously, ProViz requires a program monitor to acquire data from a running program. Common approaches include different instrumentation methods in the runtime environment (Jeffery, 1999, p. 12). But instrumented compilers or interpreters are not desirable for general purpose applications unless they can be built into the standard execution environment, which is rarely the case for mainstream languages. Instead, an approach that is becoming more and more popular is to take advantage of what is already available as a program monitor in virtually any programming environment – the debugger (Hendrix, Cross, & Barowski, 2004).

A debugger can control the program flow and display the state of a program to the user. The use of breakpoints marks where the program should suspend and the user can start looking for erroneous values. Once the program is suspended, the user can step through it line-by-line, step in and out of methods, skip over methods that are irrelevant, or resume the program until it hits the next breakpoint or terminates. The information display occurs when the program is suspended, where values of variables and the status of method calls can be displayed in various ways, depending on the view. In Eclipse, variables and their values are displayed in the Variable view, which visualizes them in tree format.

In fact, modern debuggers can be considered DPV tools and are often called “visual debuggers” because they display the program state in some graphical form. But this type of visualization is of a very low level, as values of variables are simply displayed in strings and numbers for the purposes of debugging and verifying the correctness of the program, rather than for program comprehension. Therefore, this study is not interested in the visualizing perspective of a debugger, but does want to make use of its capability of controlling the program flow.
3.3.2 Views

Source Code View
In PV, the knowledge gained from the visualization of a program needs to be mapped back to the program source itself because it is the program that the viewer wants to comprehend. Therefore, it is important to relate the visualization to the source code, and the source code view should be part of the PV process. The debugger in Eclipse shows where the execution is by opening the appropriate source files and highlighting the current executing line. ProViz uses the Eclipse debugger to control the program flow, because such a facility can show where the execution is in the source code.

Other Views
An important aspect of a PV system is that one single view of the visualization is insufficient for the viewer to gain effective understanding about the program. Contextual information of a program, such as the variable view, the method stack view, etc., must also be presented alongside the visualization (Hendrix, Cross, & Barowski, 2004; Pacione, 2004; Gestwicki & Jayaraman, 2005; Storey, 2005), so that the viewer can infer any necessary information from various displays during visualization.

The advantage of choosing Eclipse as the environment for ProViz is that it already has many of these views in place, including the debug view and the variable view in the Eclipse debugger. Furthermore, Eclipse is equipped with many powerful tools for static analysis, such as searching, browsing, and outline facilities that support code reading activities. All of these together, combined with ProViz’s dynamic visualization, the result is a complete environment that can support better program comprehension.

3.4 Methodologies

3.4.1 Data-Driven Visualization
The properties of the data-driven approach make it suitable for our purposes, particularly for the object-oriented paradigm, because each object can be mapped to its graphical representation and the state of the object will then drive the visualization. In contrast to
the event-driven approach, data-driven visualization is highly reusable, which is an important factor in reducing the animator’s effort in visualizing new programs.

The design of ProViz’s visualization process is similar to the visualization pipeline mentioned in Section 2.4.2 with the exception of an added first step: data identification (see Figure 7). In this step, the animator needs to identify “interesting variables or methods” in the source code and annotate them with painters. Then during runtime when these variables appear in the runtime data, corresponding painters are created to visualize them. This process will be further elaborated in the next chapter.

3.4.2 Method-Driven Visualization

A novel aspect of ProViz is that even though it is a data-driven PV framework, it has partial event-driven capability without the code-intrusion dilemma. This is accomplished through visualizing one type of program data – methods. Methods are represented by stack frames during runtime and contain local variables. In reality, they are part of the runtime data and can be extracted, but the concept of methods in the object-oriented paradigm is that they are actions or behaviors, and their invocations are temporal just like events. Hence, visualizing methods when they are invoked is equivalent to visualizing events occurring. Moreover, ProViz can accomplish its visualization process without code intrusion. Achieving this method-driven capability without code intrusion addresses the greatest problem of the event-driven approach.

This method-driven PV is added on to ProViz’s original data-driven capability, and together they can create more versatile visualization than any other data-driven PV system (to the best of our knowledge, no other PV tool has ever investigated or employed such a tactic). Therefore, this study also involves exploring how visualization can be effectively conceived through the collaboration between the two visualization schemes.
3.4.3 Code Intrusion

Code intrusion in data-driven tools typically occurs due to the need to associate data with their graphical functions (the data identification step in ProViz). ProViz provides two mechanisms to accomplish the task. One is Java annotations, which allows the user to add meta-information to the declaration of classes, methods, and variables in the source code. This facility is used in ProViz to specify which programming elements should be visualized and what type of painters will visualize them. Using Java annotations does result in compile-level intrusion, but this is exactly the purpose of annotations: to provide custom information for language tools.

The other mechanism is the use of external XML files to store the mapping between data and painters. In order not to let the user feel disassociated between the source code and the mapping information in XML files, user-interface tools are created to support the editing process. This approach is truly code-intrusion free, and it adds more flexibility and customizability to the composition of the visualization, which cannot be achieved by annotations. This will be discussed in the next chapter.

3.4.4 Program Monitoring With a Debugger

As described in Section 2.4.2, data-driven PV requires a program monitor to extract the runtime data. A perfectly logical and increasingly popular choice for a program monitor is the debugger. Many Java PV systems use JDI or JVMPI for this purpose (Oechsle & Schmitt, 2002; Czyz & Jayaraman, 2007). The downside of using a debugger is the decline in performance, as the overhead of a debugger is quite high. However, as discussed previously, for a live PV system such performance impact is negligible as long as it can catch up with the necessary frame rate.

A debugger has the data model for runtime data and the control mechanism for stepping programs. Because ProViz is made as a plug-in to Eclipse, it uses the Eclipse debugger to monitor program executions. Thus the ProViz framework is integrated into the debugging experience in an IDE, which is similar to the data structure visualization in jGRASP (Hendrix, Cross, & Barowski, 2004). The debugger’s mechanisms, such as
breakpoints and step operations, are therefore used by ProViz to create visualization. ProViz’s visualization is carried out alongside the debugger, and the programmer will be in fact visualizing a program and debugging at the same time.

No Behavioral Modification

By using the debugger and the data-driven approach, the execution for a single-thread program is not interfered with (as seen in Figure 8) but merely slowed down to the level where the visualization can take place and be seen. In fact, as the target program and the debugger (hence ProViz) are on two different virtual machines, under the single-thread environment, the target program’s behavior will not be modified in any way (except for time-related applications; see Section 7.1.3). So not only does ProViz boast no code intrusion, it also does not change the program’s behavior except for factors of speed and time.

Figure 8 – ProViz’s program monitoring with a debugger. The original program’s execution is broken into several segments of steps, and in-between these steps the monitored program waits until ProViz finishes the visualization and the debugger issues the next step command to continue the program.
Encapsulation

From the perspective of implementation, an advantage of ProViz’s data-driven design using a debugger is that ProViz is able to access the private fields and methods of objects and is not bound by the object’s encapsulation, which could in fact hinder the event-driven approach. Revealing the private members of objects means that the visualization can portray the data abstraction of their underlying structures, which encapsulation purposefully hides, so that the viewer can understand the inner workings of these objects.

3.4.5 The Painter Paradigm

The painter paradigm is plain object-oriented design. Each painter is designated to visualize a variable or an object in the runtime data. Once created, a painter fetches the values it needs in the runtime data to produce visualization.

Objects have hierarchical structures, as an object can have fields and these fields can have their own fields, and so on. Therefore, in ProViz, objects are modeled as trees of variables, where the root of a tree is a local variable, and the leaves of a tree are either null variables or primitive variables. When an object is to be visualized, its variable tree will then be visualized by a tree of painters, and this painter tree will be shaped like a pruned tree of the original variable tree because some field variables may not be visualized.

Modularization of the Painter Design

A painter can be designed to visualize the entire variable tree of an object, but such a design is too rigid, and our empirical work has shown that this would benefit little from ProViz’s functionality. Therefore, the painter design and structure should be modularized so that a painter concentrates on visualizing a variable. A tree of variables will then be visualized by a tree of painters instead of one single painter. Take the following class as an example:

```java
class Example1 {
    private int iField;
    private String sField;
}
```
public static void main(String[] args) {
    Example1 e1 = new Example1();
}
}

The local variable e1 points to an Example1 object with two fields, e1.iField and e1.sField. In the runtime model, there are a total of three variables forming a tree: e1 is the parent and the two fields are its children. To visualize the object of e1, one approach is to create only one painter P_e1 for e1 and let P_e1 also draw iField and sField. The other approach is to modularize the painters and let each field be visualized by its own field painter. In the latter approach, there will be three painters in total: P_e1, P_iField, and P_sField where P_e1 is the root painter and P_iField and P_sField are its field painters.

In practice, the first approach is suitable only for simple types, as it would be difficult for one painter to visualize the entire hierarchy of complex fields. The latter approach, on the other hand, encapsulates a painter’s task to visualize one single variable, and all painters work together for the final visualization. This presents a more modular design and makes better use of ProViz’s engine, which automates the chore of creating and deleting painters according to the state of the program.

3.4.6 Applicability to Any Program

The key to having ProViz applicable to any (currently Java) program is the property of being code-intrusion free; because the program monitoring approach uses the fundamental debugger in the programming language, any program can go through the visualization process without even the slightest modification. Therefore, as long as the source code of a program is available, ProViz can visualize it.

3.4.7 Customizable Visualization for Different Viewers

Different audiences possess different levels of perception, and therefore visualizations presented to them need to vary in presentation style and abstraction (as discussed in Section 2.1.4). ProViz is specifically designed so that it can make multiple visualizations out of one program by using different annotation configurations, called Viz sets. Each Viz
set is a configuration of mapping data to painters, where different painters may give different graphical depictions of the same variable or method. As a result, different Viz sets can give the same program a variety of looks, where each depiction conveys certain information or is tailored for a specific type of audience.

3.5 What ProViz Is Not

Tool Collaboration Leads to Program Comprehension

An important aspect of PV for program comprehension is that visualization is one of a collection of ways to assist the comprehension process. This study does not claim that the use of high quality PV will replace other program comprehension techniques. It is believed that the best program comprehension can be achieved by using a combination of all available tools, and the more effective these tools are, the better the comprehension process will be supported and simplified. ProViz can produce high-quality, storytelling types of visualization and therefore can help people understand programs effectively with its high-level conveyance. This is to be combined with textual documentation, architectural diagrams, and any other documentation methods available to achieve program comprehension. ProViz is not intended to function alone, which is why the Eclipse IDE is chosen to be its host environment.

Best Visualization?

This study does not claim that ProViz produces the best visualizations for programs. It is a tool for generating visualization from programs, but whether the resulting visualization is useful or comprehensible depends on the available graphics framework and the programming and artistic skills of the animator who creates it. This study does not invent fancy new graphics or revolutionary ways of visualizing programs but focuses on the process and the methodology of creating PV.

3.6 Visualization Paradigms

The main challenge in developing ProViz is to overcome the paradigm difference between two worlds: variables in program source codes and objects in program runtime. What sits between these two paradigms is the effect of aliasing.
Aliasing

In the Java source code, (reference) variables are often thought to represent objects, especially by naïve programmers. But in fact, objects do not exist until the program starts running. Therefore, the source code is implemented under the variable paradigm, and so is ProViz’s annotation mechanism, as both Viz Annotation and Viz XML annotate program elements from the source code. However, the reality of object-oriented programs is that during runtime, objects are dynamically allocated and occupy memory blocks, and variables are not objects but pointers referencing objects. When multiple variables point to the same object, aliasing occurs.

Aliasing is an important concept to be recognized by programmers during programming and debugging, and it is also an implementation issue that, as far as this study has gathered, only one other PV work has ever addressed – the Tarraingím system (Noble, 2002; Noble, Groves, & Biddle, 1995). However, because Tarraingím’s approach is top-down visualization, the aliasing problem and its solution are completely different from those in ProViz. Aliasing affects the identities of variables and objects and hence changes how they are represented in the visualization. So the question is: should a PV system visualize by variables or by objects?

3.6.1 VOV – Visualization Of Variables

In VOV, each variable is visualized independently by its own graphical depiction. When visualizing a number of aliasing variables that reference the same object, there will be the same number of graphical figures on screen for these aliasing variables even though there is only one object. Therefore, VOV circumvents the effect of aliasing and may confuse the viewer’s perception of the identity of objects.

An Erroneous Example of VOV

A simple example of how the variable paradigm can cause inaccurate visual depiction is when circular pointers are displayed in the debugger.

```java
public class CircularPointer { 
    private CircularPointer pointer = this;
    public static void main(String[] args) { 
        CircularPointer pointer = new CircularPointer(); 
    }
```
In this example there is only one object and one reference variable as the object’s field. However, because the debugger visualizes by variables, it can crawl through the reference variable endlessly without knowing it is a pointer to the same object (see Figure 9).

![Figure 9 – Circular-pointer display in debugger view](image)

**VOV Example**

```java
String[] sArray = new String[5];
Hashtable<Integer, String> hTable = ...;
String str = “smile”;
sArray[0] = str;
hTable.put(0, str);
```

In the above code segment, a single `String` object is referenced by a `String` variable, `str`, and is inserted into an array and a hash table. So a total of three aliasing variables are pointing to the object. Using VOV in Figure 10, there are three different figures of this object on the screen, and each figure can have a different representation for this string. The problem is that such a depiction may mislead the viewer into understanding the number of objects for “smile” to be three, when in fact there is only one.

**Advantages of VOV**

- As seen in Figure 10, the advantage of VOV is its versatility on the visualization

![Figure 10 – An example of VOV visualization. The String object is shown in three different graphical forms depending on the variables.](image)
output. One object can be visualized in different ways to suit the variable’s context or the viewer’s perception.

- VOV is much simpler to implement than VOO, where a painter is set to visualize one variable regardless of the presence of other aliasing variables.
- It is possible to raise the user’s awareness of aliasing in VOV through some visualization techniques, such as coloring or highlighting. For example, when the user selects a painter, other aliasing painters can be highlighted to show that they are drawing the same object. This is what ProViz currently implements to show the effect of aliasing.

### 3.6.2 VOO – Visualization Of Objects

In VOO, objects are depicted by their true identities, where there is only one graphical figure representing an object no matter how many aliasing variables are referencing it. This graphical uniqueness is precise and accurate for object-oriented programs. It is demonstrated in Figure 11 for the same code segment from the previous section.

In Figure 11, there is only a physical appearance of the string “smile,” and aliasing variables in the array and the hash table are indicated by drawing arrows to reference the string. The advantage of VOO is its *fidelity*, as it accurately displays the runtime state of a program and the effect of aliasing.

**Challenges with VOO**

The fact that the program source code uses the variable paradigm and that objects are dynamic entities generated along the program execution means that there is no conceivable way of mapping objects to painters. The only access to objects is through variables, but aliasing allows multiple variables referencing a single object. In ProViz, because painters are annotated by variables, aliasing applies to painters as well and multiple painters could be painting one object. To achieve VOO and have only one
A graphical figure on screen for a single object means that only one painter should exist despite other aliasing variables that have been annotated for visualization. The problem is that aliasing variables could be annotated with different painters, which creates a conflict that leads to the following question: which variable gets the ownership\(^1\) of this one painter and therefore gets to decide the type of the painter?

Figure 12 shows another VOO visualization of the above example. Here the actual String object resides in sArray’s visualization, and the variable sArray.[0] is the owner of the painter for this String object; whereas in Figure 11, str is the owner variable for the painter. As we can see, when a different variable owns the painter, it will determine the type of painter to be created and thus change the appearance of the object. However, who gets to determine that the array would draw the string in the vertical format, instead of drawing it in the hash table as a smiley face?

An implementation of pure VOO would necessitate a resolution scheme to deal with aliasing variables. As visualizations should be made to satisfy the viewer, animators would be the ultimate decision makers. However, it is unrealistic for the animator to set priorities for aliasing variables in a program, and having them do so would also be contrary to the objective of ProViz being easy-to-use. In contrast, VOV is simpler to implement, easy to understand, and can still raise the awareness of aliasing.

\(^1\) When a painter is assigned to a variable, it is this variable’s painter, and the variable is its “owner.”
3.6.3 VOV or VOO?

These two visualization paradigms are not only the outcome styles of visualizations but are also methodologies of the creating process. We define three different levels of design that can determine whether the visualization outcome is VOV or VOO.

**System Level**

The ProViz system decides the paradigm by managing painters differently. For VOV, a painter would be associated with a variable; whereas for VOO, a painter would be owned by several aliasing variables. ProViz uses VOV as its basis. Therefore, the task is to find out which types of variables should be visualized with VOO by default. For example, local variables that have been annotated with the same type of painter, method parameters, or instance variables in the class of the current executing method (see Section 6.2.4) can be candidates for VOO processing at the system level.

**Painter Level**

The implementation of a painter can determine the VOV or VOO outcome. When a painter is created with VOO, it is associated with a list of aliasing variables, so it can decide whether it should visualize these aliasing variables to produce the VOV visualization. However, this is rarely the case as the majority of painters in ProViz will be in VOV.

A much more common scenario is a painter that is created with VOV but can still produce the VOO visualization. First it needs to check whether there is an aliasing painter that visualizes the object. If there is, then this painter does not put another graphical component onto the canvas and perhaps draws an arrow to the existing painter instead. Otherwise, if the object has not been visualized yet, this painter would visualize it normally.

**User Level**

The animator can design the VOO visualization by annotating only one variable of an object (typically the very first variable that references this object) and never other aliasing variables of this object. However, this is only suitable for small-scale programs.
because it requires knowledge about where an object is first created. If the user annotates other aliasing variables, then the outcome style will be resolved by the design of the other two levels.

The researcher operated at the user level when testing and experimenting with the framework and disabled VOO at the system level. Further investigations are needed to decide whether ProViz should enforce VOO on the variables mentioned above or even on additional types of variables.
Chapter 4
Framework Architecture

The design of ProViz follows the data-driven visualization process in Figure 7 with each step encapsulated into a component in the system. The three components are called Viz Map, Viz Monitor, and Viz Painters (as seen in Figure 13).

![Diagram](image)

**Figure 13 – The data-driven process and corresponding components of ProViz**

The functionality of ProViz begins with Viz Map, which allows an animator to identify programming elements that she wants to visualize and also to create the mapping between these elements and the painters that will be responsible for visualizing them. This mapping can either be specified using the declarative approach with “Java annotations” or be composed in an editor and stored in XML files. The resulting mapping information is stored in Viz Map Model (as seen in Figure 14). During program runtime, Viz Monitor, which interacts with a debugger, monitors the program execution, extracts the program’s runtime data, and stores it in Viz Runtime (as seen in Figure 14). These two models are utilized by Viz Painters to create visualizations: using the mapping information in Viz Map Model, Viz Painters know which painter should be used to visualize which programming element; then each painter can visualize its element on the screen. The detailed framework architecture is shown in Figure 14.
4.1 Defining the Data

Before examining the components of the ProViz framework, we must first define the data to be visualized. In an OO language, there are four types of programming elements that are involved in ProViz visualization: class, field, local variable, and method.

Class

In Java, classes are blueprints for objects which are created at runtime, and when the visualization is defined for a class, instances of this class – i.e., its objects – can be visualized as well. Therefore, a painter that can visualize a class can visualize all the objects of this class. Consequently, declaring a painter for a class in ProViz is equivalent to declaring the default painter for visualizing any instance of this class.

Field

Fields, aka instance variables, are variables declared inside a class and outside any method block. In OO concept, they define the properties of objects, and visualizing an object is in fact accomplished by visualizing its fields. On the other hand, outside of the OO concept, fields often function as global variables for easy access between methods.
The visualization of global variables thus can be different from visualizing the fields of an object.

**Local Variable**

Local variables (including method parameters) are variables declared within methods and any control block in methods. They are different from fields in scope and lifetime, as they are only accessible within the block in which they are declared, and will be de-allocated once the execution exits the block.

**Method**

Inside classes there are methods, which are coding blocks that contain programming statements, including the declarations of local variables. They are the “actions” that objects can perform in the OO world. Unlike variables, methods generally are not mapped into visual objects. Instead they are operations that manipulate the runtime data while inter-calling other methods. Therefore, the visualization of a method is usually a series of graphical events reflected on existing data visualizations.

### 4.2 Viz Map

The goal of Viz Map is not just to identify WHAT to visualize but also to determine HOW to visualize by annotating programming elements with painters. Let $S$ denote all programming elements that can be visualized. $S$ consists of classes $C$, fields $F$, local variables $V$, and methods $M$ — i.e., $S = \{C, F, V, M\}$. Let $S_v$ be the set of programming elements that the user wants to visualize: $S_v \subseteq S$. Let $P$ represent the set of available painters. For a programming element $e$ that is identified to be visualized $e \in S_v$, $e$ should be annotated with a painter $p \in P$. Then this element will be visualized during runtime with $p$ serving as a visual function and mapping $e$ to its visualization: $p(e)$, and the type of $p$ will determine how $e$ is visualized. Therefore, the purpose of Viz Map is to let the user identify $S_v$ and annotate each $e \in S_v$ with $p$.

What is needed is a mechanism for creating this mapping without significant code intrusion. ProViz offers two different mechanisms: *Viz Annotations* and *Viz XML*, to accomplish this with minimum and zero code intrusion respectively.
4.2.1 Viz Annotations

As mentioned in Section 3.4.3, Java annotations are used as the mapping mechanism for the user to declare painters on programming elements. The advantage of using annotations is that their physical locations are right above the declarations of programming elements, so an animator can easily associate programming elements with their painters. Annotations exhibit compile-level code intrusion, which is standard amongst common programming language tools (but still pales compared to Viz XML, which is intrusion free). Two types of Viz annotations are defined in ProViz: @Viz and @DViz.

**Custom Visualization: @Viz ("p₁, p₂, p₃,...")**

@Viz requires a string as its parameter. The string is a list of painter names: “p₁, p₂, p₃” where each painter can visualize e and the first painter p₁ is the default painter. The painter names are separated by commas, and each painter name must be the fully-qualified class name of the painter. During visualization when e appears in the runtime data, ProViz then creates p₁ through Java reflection using its class name.

For example, if one wants to visualize an integer variable using a painter whose class name is IntegerPainter and is located in sample package, she would add the following annotation to the declaration of i:

```java
@Viz ("sample.IntegerPainter, sample.OtherPainter")
int i;
```

“sample.OtherPainter” is another painter that visualizes i presumably in a different way, and it can be selected to replace IntegerPainter in ProViz Editor (discussed in the next section) or dynamically during runtime. Adding @Viz to classes and methods is the same as the above by adding the annotations before their declarations:

<table>
<thead>
<tr>
<th>@Viz for a Class</th>
<th>@Viz for a Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Viz (&quot;sample.SamplePainter&quot;) class Sample {...}</td>
<td>@Viz (&quot;sample.MethodPainter&quot;) public void method() {...}</td>
</tr>
</tbody>
</table>
Default Visualization: @DViz

@DViz stands for the default visualization with no painter name required. When a variable \( v \in \{ F, V \} \) is annotated with @DViz, first ProViz will try to look up the annotation of \( v \)'s class type – \( C_v \). If \( C_v \) is annotated with a painter, then \( v \) will be visualized with such a painter; otherwise, it will be visualized with the system-default painter in ProViz.

4.2.2 Viz XML

Viz XML stores the product of Viz Map – Viz Map Model – in XML files. This externalizes the mapping information from the source code and results in zero code intrusion. Viz XML thus enables the concept of Viz sets, described in the next section.

Besides portability and customizability, the biggest advantage of Viz XML over Viz annotations is its capability of annotating program modules whose source codes are not available, such as external libraries and the Java API. Because Viz annotations require the presence of the source code, it can never be applied to library modules whose codes are unavailable or are not modifiable. On the contrary, Viz XML is decoupled from the source code and can be manually customized. Hence using Viz XML, an animator will be able to assign painters to elements in external libraries and visualize them.

As mentioned in Section 2.5, the problem with externalizing the mapping information is that it makes it more difficult for an animator to relate programming elements to painters. In addition, externalizing the programming elements could result in redundant data sources and thus introduce the synchronization problem. To overcome these inconveniences, Viz Map uses a user-interface tool, ProViz Editor, to facilitate the mapping process. The user can load Viz annotations from a Java file or a Viz XML file into this editor and manipulate the mapping information. Then the mapping model can be saved to a Viz XML file.
**ProViz Editor**

The purpose of ProViz Editor is twofold. First of all, what is loaded in the editor will be the Viz Map Model used for visualization lookup during runtime. Secondly, it is the workspace for the animator to compose Viz sets.

To load Viz annotations from a Java or a Viz XML file, one can open the file in the Eclipse editor and press the load button in ProViz Editor to load it. Furthermore, the editor automatically loads the “vizlib.xml” file at startup. This file is ProViz’s system file that contains painter mapping for fundamental data types in the Java API.

### 4.2.3 Viz Sets

A Viz set is a configuration, an instance of Viz annotations. With this configuration of visual mapping, ProViz can show one particular view or aspect of a program’s visualization. If another Viz set is constructed for the same program with different painter mappings, the resulting depiction can show a different perspective of the program. In other words, one can construct several Viz sets for the same program, and switching between them will display the program in multiple perspectives. For instance, a string painter may visualize a `String` object with the string being displayed horizontally, and another painter visualizes a `String` object vertically. If one Viz set employs the former painter and another Viz set uses the latter to visualize the same program, then the visualization with the first Viz set will display strings horizontally, while the other Viz set will show them vertically.

Viz sets are constructed by putting together Viz annotations from Java files or Viz XML files and then are customized to meet the user’s needs. Using ProViz Editor, an animator can load Viz annotations from different files, modify or reorder painters, delete unwanted programming elements (for better runtime efficiency), and so on. Upon completion, a Viz set can be saved as an XML file.

### 4.3 Viz Map Model

What is loaded in ProViz Editor is the Viz Map Model that will be used at runtime. Viz Map Model serves two purposes: first, it contains the mapping between $S_r$ and $P$; second,
it acts like a filter and determines which method executions will be monitored by ProViz during runtime.

### 4.3.1 Method Filtering

As described above, the second function of Viz Map Model is to tell ProViz which methods should be monitored. In ProViz, a method will be monitored only if it exists in Viz Map Model. Because a program’s execution is a series of method calls and some of the methods do not affect the visualization, these methods should be ignored and not monitored. Since ProViz needs to analyze and perform visualization at every program step, skipping irrelevant methods can result in a significant performance boost.

The access and retrieval on Viz Map Model are all done internally by ProViz. The animator making painters would not need to interact with this model. However, it is possible for the viewer to manipulate the model through ProViz Editor during runtime in order to alter the visualizations.

### 4.3.2 Model Structure

The programming elements form tree structures that resemble their original structures in the program source code. Classes are the top level containers that hold fields, methods, and inner classes, $C$: {$F, M, C$}. Methods contain local variables and inner classes, $M$: {$V, C$}. The leaves in a tree are fields, local variables, and, although quite unlikely, methods and classes that are blank.

**Accessing the Elements**

All elements are identified through their names. Classes have their complete path names as their ID (ex: `java.lang.String` for the String class); method names are constructed by their classes’ complete path names followed by a colon and the methods’ signatures (ex: `java.lang.String:indexOf(int)` for the `indexOf` method in String class). Class names and method names are thus unique throughout a program and can be directly accessed through such IDs. Fields and local variables, however, are indirectly accessed through their parent classes or methods. Our implementation of
ProViz builds two hash tables for classes and methods before runtime in order to speed up the access to these elements.

## 4.4 Viz Monitor

Viz Map and Viz Map Model are static tools – they are accomplished through the source code or XML files before the program runs. On the contrary, Viz Monitor starts working after the program begins to execute. It controls the execution of the target program in a step-by-step fashion using the Eclipse debugger. At each step it retrieves the runtime data from the Eclipse debug model (composed of IStackFrame, IVariable, and IVValue), converts them into ProViz’s runtime types (VizStackFrame and VizVariable), and then updates Viz Runtime.

### Debugger Controls

Like most debuggers, the Eclipse debugger provides three commands to control a program’s execution: step into, step over, and step return\(^1\). ProViz uses two of these commands to control the program flow: step into and step return.

### Ruling out Step Over

Step over allows a user to manually skip debugging a method prior to entering this method.

```
1  int num = 0;  \(\text{current execution point}\)
2  methodCall();
3  num++;  
```

Using the above code as example, suppose the current execution point is after line 1, meaning that line 1 has been executed but not line 2. At this moment, the debugger will highlight line 2 as shown, indicating that it is the next statement to be executed. If the user invokes the step over command now, `methodCall()` will be executed on the fly. In terms of program monitoring, this indicates that `methodCall()` is completely skipped.

\(^1\) Also known as “step out” in some debuggers.
Step over seems to be a perfectly good command for not monitoring unwanted methods. However, the problem is that Viz Monitor cannot know what method is going to be executed next because the execution has not entered the method. As what we will discuss in the next section, a method call is evidenced by the appearance of a stack frame in the runtime data. Since the execution has not entered the method, no stack frame is created for this method. As a consequence, Viz Monitor cannot foresee or decide to skip an incoming method, and the step over command is not used in ProViz.

Instead, the functionality of step over can be achieved by the combination of step into and step return. Upon executing a method, step into first requests the execution to enter this method. ProViz then identifies this method, attempts to locate it in Viz Map Model (recall the method filtering scheme from 4.3.1), and determines whether it is necessary to monitor this method. If this method is not in Viz Map Model, a step return command will be issued and the execution finishes the method and returns immediately, effectively skipping the method. In summary, ProViz uses step into throughout the program execution and uses step return only to break out of unwanted methods.

4.5 Viz Runtime

Viz Runtime is the runtime data model that will be visualized by painters during runtime. It represents the state of a program at any given step of the execution.

4.5.1 Runtime Data Model

Eclipse Data Model
For a single application, the top-level data is a thread, which consists of a stack of methods, and each method contains a number of variables and values. The Eclipse debug framework models the runtime data with its own types. The method stack is modeled by a stack of IStackFrame’s. Each IStackFrame contains a list of local variables modeled by IVariable. An IVariable represents a variable and has a value – IVValue. An IVValue contains a string value of the variable and a list of IVariable’s, which are fields of an object. As a result, variables and values form tree structures when modeling objects.
Viz Runtime’s Data Model

Viz Runtime follows a similar design and structure of Eclipse’s debug model but currently eliminates the use of a value type like IValue. It models methods with VizStackFrame, and VizVariable is used to model variables. A VizVariable has a string value and directly contains a list of VizVariable’s as fields, as opposed to the use of IValue in Eclipse. Nevertheless, the variable tree structure is the same as Eclipse’s model, and during runtime the integrity of Viz Runtime can be verified by comparing it to Eclipse debugger’s Variable View.

Unlike Viz Map Model that is used internally, Viz Runtime’s data is accessed by painters, so painter makers must understand its structure in order to traverse the model. This will be elaborated in Section 5.2.

Redundancy

At current stage, all reference variables in Viz Runtime have their own variable trees, and thus aliasing variables, i.e., variables that point to the same object, will have duplicate and redundant VizVariable trees. This is because Viz Runtime copies IVariable’s from Eclipse debug model during data extraction regardless of whether they are aliasing variables. In the future, we plan to undo such redundancy and model the runtime data with the exact same structure as the actual runtime data.

Identification for Variables

Every VizVariable has a unique object ID, and we can distinguish the type of a variable using this ID:

- Primitive variables: ID is null
- Null variables: ID is -1
- Non-null reference variables: IDs are positive integers

Since each object has a unique ID, variables pointing to the same object will possess the same ID. In other words, variables that have the same ID are aliasing variables.
4.5.2 Model Maintenance

Persistence
The VizVariable data type persists for the lifetime of the variable it represents (unless any of its ancestral variables is changed; this is discussed in the next subsection). This persistence is due to the event-listener model implemented to relay data changes to painters (further discussed in Section 4.6.2.2). Hence VizVariable has to be persistent storage in order to keep the list of listening painters.

As a consequence, the model update is performed variable-by-variable rather than replacing the old model with the new data extracted from the debugger. The downside is that it takes more time, but the benefit with such an approach is that during this update, Viz Runtime can also identify changed variables. Then only these changed variables will generate events and notify their listening painters, which update their visualizations accordingly.

4.5.3 Changed Variables

Changed variables can be categorized into three types: primitive variables, reference variables and a special type of reference variable – null variables. Changes to primitive variables are extremely simple to update and will not be discussed here.

On the other hand, the change to a reference variable can only mean one thing: assignment to another object or the null value. When this happens, many attributes of a VizVariable will change (except for its name and declared type, which are persistent throughout a variable’s lifetime). Furthermore, because objects are modeled by variable trees, assigning a variable to a different object means that the sub-trees under this variable will be changed as well. It is very troublesome and time-consuming to synchronize the old sub-trees with the new ones as it involves creating and deleting branches to match the new tree structure. Therefore, Viz Runtime does not perform this type of update but simply attaches new sub-trees to the changed variable. This leads to a very important principle for changed variables: all variable sub-trees under a changed variable are recreated as new VizVariable trees, which means that the old sub-trees
are discarded. In summary, the changed VizVariable itself will persist with only its attributes being updated, but all its field variables will be new VizVariable objects.

4.6 Viz Painters

Viz Painters consists a library of painter classes, an animation controller, and VPM (Viz Painter Manager) – the controller of painters. During runtime, VPM provides the storage for painters and also automates the process of creating and removing painters (thus determines VOV or VOO at the system level). After VPM updates the state of painters, painters will visualize and possibly schedule animations with the animation controller, which carries out the animation at the end of the execution step.

4.6.1 Painter Types

Principally there are two types of painters: variable painters and method painters. Structurally, however, there can be several types of variable painters depending on the types of variables they visualize.

4.6.1.1 Variable Painters

One rule that applies to all variable painters in ProViz is that a painter has at least one owner variable, i.e., no painter will exist without an owner variable.

Primitive Painters & Object Painters

Primitive painters visualize primitive data types in Java. There are three distinctive properties of primitive painters:

1. The ID of their variables is null.
2. They cannot have field painters.
3. They can only have one owner variable (except for those whose root variable is under VOO), because primitive data types do not have the issue of aliasing.

All other painters are object painters which visualize reference variables. All object painters would need to handle the case where their variables point to the null value.
**Root Painters & Field Painters**

Root painters visualize local variables and therefore are the roots of all painter trees; all other painters visualize fields descendent from local variables and are field painters. The distinctive property of root painters is that they do not have a parent painter (getParent() returns null); conversely, a field painter must have a parent painter due to the principle of *no dangling painter* (see Section 4.6.2.1).

**4.6.1.2 Method Painters**

A method painter is created and begins visualizing when the execution enters the method it is annotated to. As mentioned previously, methods calls are temporal events, and therefore visualizing methods is substantially different from visualizing variables. The preeminent difference is the time. While a variable painter visualizes throughout the lifetime of its owner variable, a method painter depicting events typically produces visualization only at the invocation of its method – the characteristic of an event. Once the visual event is accomplished, the method painter only needs to stop painting, and the execution either continues executing the method or exits it. Furthermore, the visualization of method painters often involves merely manipulation of other variable painters.

**4.6.2 VPM (Viz Painter Manager)**

The ProViz visualization is accomplished by a collection of painters that paint the variables they are assigned to. During runtime, these painters are stored and managed by VPM, which is responsible for creating and deleting painters to match the state of the runtime data in Viz Runtime.

**The Data Structure of VPM**

As the storage of all painters, VPM needs to support access to all painters as they will interact with one another during the visualization. VPM uses two different storages for primitive painters and object painters. Primitive painters are stored in a hash table with the owner variable as the key (since there is no unique ID for primitive variables) and the painter as the value. So to retrieve a primitive painter, one needs to find its owner variable in order to get the painter from VPM.
Object painters are stored in a chained hash table with the object’s ID as the key and a list of aliasing painters as the value. An object painter can therefore be accessed through the ID of its owner variable. But because VOV visualizes each variable with a painter regardless of aliasing, there will be aliasing painters that have the same ID. Therefore, each unique ID corresponds to a list of aliasing painters, and to find the exact painter is to check if the owner variable has the right stack frame and the correct variable name.

In VOV, the relationship between painters and objects is many-to-one, and it is one-to-one in VOO. To associate a painter with an object, this painter will be referenced by the object’s aliasing variables. Hence to implement the VOO aspect, a painter needs to be able to store multiple aliasing owner variables, and this is easily accomplished.

4.6.2.1 Managing Painters

At each program execution step, VPM performs several tasks to maintain the state of painters: (1) removing painters for variables that are de-allocated; (2) creating new painters when new variables that should to be visualized are allocated (the choice VOV or VOO is determined in this process); (3) updating changed variables and their sub-trees; (4) notifying painters listening to the changed variables so that these painters can update their visualizations based on the new data.

**Painter Creation**

After Viz Monitor updates Viz Runtime, VPM will manage the painters based on the data state in Viz Runtime. Any newly-allocated local variable is checked against Viz Map Model to see whether it is annotated with a painter. If so, a painter is created, and this variable’s fields will go through the same process recursively; otherwise if the variable is not annotated, no painter will be created for it or for any field under it.

**No Dangling Painter**

The result of the above creation scheme prevents the occurrence of dangling painters. That is, any painter tree must have a root painter as its root, and no field painter can ever exist without a parent painter. The reason for such a rule is quite simple. Consider the following example:
class Example {
    @DViz
    private String sField;

    public static void main(String[] args) {
        @Viz ("ExamplePainter")
        Example e1 = new Example();
        Example e2 = new Example();
    }
}

In class Example, the instance variable sField is annotated for visualization, which means that any painter that visualizes an Example object will have a field painter for sField. In main() there are two different Example objects, referenced by e1 and e2. e1 is set to be visualized with an ExamplePainter, and thus this ExamplePainter will have a field painter created for e1.sField. The variable e2 also has such a field, which is e2.sField, but should this field be visualized with a field painter?

Because e2, the root variable of e2.sField, is not annotated, the entire object referenced by e2 is not meant to be visualized. Therefore, e2.sField should not be visualized as well even though the field is annotated. Hence no painter will be created for e2.sField, and the rule of no dangling painter needs to be established. All field painters must have a parent painter that descends from some root painter. So in order to visualize an object, the annotation must begin with the root variable of an object – a local variable.

**Updating Changed Variables**

Recall from Viz Runtime that when a reference variable is changed, all its field variables are recreated as new VizVariable objects. Because a field painter is associated with a field variable, it shares the same fate as this variable. *When a reference variable is changed, its field painters will all be recreated as new painter objects.* The impact of this approach is that when an object painter handles the change event from its owner variable, its field painters will be newly created where the old ones have been removed and destroyed. Therefore, these new field painters must be added onto the canvas if they are to be seen on screen. This is further elaborated in Section 5.4.5.
4.6.2.2 The Event-Listener Model

Prior to the visualization, Viz Runtime and VPM have ensured that the state of both runtime data and painters are up-to-date. Lastly, ProViz needs to reflect data changes on painters so that they can update their visualizations. There are two general strategies for such a task. One is the brute-force approach that lets all painters examine their data and repaint. This approach is simple but can be very inefficient under two conditions: (1) the number of painters is large, and (2) the number of changed variables at each execution step is small. Keep in mind that the visualization update occurs at each and every step of the program monitoring. If the number of painters is large, refreshing all of them will certainly be undesirable. In addition, the number of changed variables in every step is generally small in reality, typically in the lower range of a single digit. Making all painters to check and update for a small number of changes is obviously inefficient.

As a result, we opted for the other approach, which is the event-listener model. Painters are listeners that can be registered with VizVariable’s in Viz Runtime. When a variable is changed, it generates a change event and notifies listening painters which can then visualize accordingly. Registering listeners can be manually specified in VizMap, or it can be done programmatically by painters.

The Waiting List

There is, however, one issue for the event-listener model. As programs are executed sequentially, when a painter is created and is going to be registered with variables, some of these variables may not yet be declared. Therefore, a waiting list is needed in VPM to store these painters so that later when the desiring variable is allocated, they can be registered with this variable.

Step Processing

What has been discussed so far in this chapter is considered the step processing. From Viz Monitor, Viz Runtime, to VPM, the data are extracted and painter updates are performed. After step processing is finished and all data models are up-to-date, visualizations can be updated by painters painting either statically or with animations.
4.7 User Scenarios

4.7.1 Scenario I – Painter Ready

The first scenario is under the assumption that all necessary painters for visualizing the target program are already made. So the tasks of an animator would be the following:

1. Identify the programming elements in the source code for visualization and annotate them with painters. This can be done in three different ways:
   a. Annotating them in the source code using Viz annotations and loading them into ProViz Editor.
   b. Loading related source files into ProViz Editor and annotate the elements in the editor.
   c. If an appropriate Viz XML files already exists, load it into ProViz Editor.
2. Put a breakpoint at the start of the program (e.g., the main method) or wherever the animator wishes to start visualizing in the program.
3. Start the debugger and debug the program. A save window should pop up, prompting whether the user wishes to record the visualization. Enter the file name for saving the recording or select “cancel” to void the recording.
4. Use ProViz’s animation controls to start the visualization.

4.7.2 Scenario II – Playing Recordings

If a recording file is made from a previous visualization session, Viz Player can be used to load the file and play back the visualization. That is the “Viz Player” button in ProViz tool bar. Once clicked, a file chooser pops up for selecting the recording file. Once selected, the user can use the animation controls to view the visualization.

4.7.3 Scenario III – Making Painters

If the painter library is insufficient or some programming elements need customized visualizations, the animator will need to make new painters. After new painters are made, one can follow the first scenario to visualize the program.
Chapter 5
Visualizing Programs and Making Painters

5.1 Annotating Programs with Painters

To visualize a Java program, one needs to identify which programming elements (including classes, fields, local variables, or methods) should be visualized and annotate those using Viz annotations: \texttt{@Viz ("P_1, P_2, \ldots").} \texttt{@Viz} takes a string parameter that contains a list of painter names, and the first one \(P_1\) in the list will visualize the element by default. \texttt{@DViz} stands for default visualization and has no parameter. Annotations are placed right before the declarations of programming elements.

The Format of \texttt{@Viz}

Because \texttt{@Viz} takes a string parameter, its usage is more versatile than the single use of \texttt{@DViz}. The format of \texttt{@Viz} is described as follows:

\[
\texttt{@Viz ("P_1, P_2, P_3 \ldots")}
\]

\(P_i\) is the fully qualified name of a painter class followed by an optional specification of the starting location for this painter, surrounded by parentheses:

\[
P_i := \{\text{painter’s fully qualified name}\}[ (x, y)]? \quad (?: \text{zero or one occurrence})
\]

Examples of \texttt{@Viz}

1. \texttt{@Viz ("viz.lib.StringPainter, viz.lib.AutoPainter")}
   
   In this example, the target variable will be visualized with a \texttt{StringPainter} by default. Once loaded into ProViz Editor, the animator will have the option of switching the order of these two painters. The viewer can also switch the painter dynamically during runtime from this list of painters.

2. \texttt{@Viz ("viz.lib.StringPainter(100, 100)")}
   
   In this example, the target variable is visualized with a \texttt{StringPainter} and will be initially placed at the coordinate (100, 100).
Special Usages

- "1D" is the keyword used to specify the default visualization. It is chosen as the keyword because no Java class name can begin with a number, and "D" denotes the default. Once a variable annotated with @DViz is loaded to ProViz Editor, the painter name for this variable is automatically replaced with "1D."

- @Viz ("") and @Viz ("1D") are equivalent to @DViz.

- @Viz ("(x, y)") is equivalent to @DViz plus the specification of a starting coordinate.

The following subsections will show the principles of how to annotate a program for visualization.

5.1.1 Annotation Begins with Local Variables

To visualize an object or any of its fields, one must begin by annotating a local variable that points to this object; otherwise the visualization will not display. Take the following code segment as example. Suppose the objective is to visualize the instance variable: array.

```java
class Example1 {
    private int[] array;
    public static void main(String[] args) {
        Example1 ex = new Example1();
    }
}
```

The natural thinking is to annotate the array variable as follows:

```java
@DViz
private int[] array;
```

However, during program runtime, array is not a standalone object – it is a field of the object referenced by ex. Hence to visualize array, the local variable ex needs to be annotated. Assume Example1 can be visualized by ExamplePainter. Then the following code shows the correct way of displaying array:
public class Example1 {
    @DViz
    private int[] array;
    public static void main(String[] args) {
        @Viz ("ExamplePainter")
        Example1 ex = new Example1();
    }
}

5.1.2 Class Annotation Stands for Default Visualization

An equivalent way of visualizing the above example is as follows:

    @Viz ("ExamplePainter")
    public class Example1 {
        @DViz
        private int[] array;
        public static void main(String[] args) {
            @DViz
            Example1 ex = new Example1();
        }
    }

When a class (e.g., Example1) is annotated with a painter (e.g., ExamplePainter), any variable of this class (e.g., ex) that is annotated with @DViz will be visualized with this painter. If a class is not annotated and a variable of this class is annotated with @DViz, it will be visualized with ProViz’s system default painter.

5.1.3 A Comprehensive Example

Given

1. ProViz’s default painters for String, array, and any primitive data type
2. ex.BubbleSortPainter – a painter for the BubbleSort class
3. ex.IndexPainter – a painter that visualizes a variable as an index of an array
4. ex.SwapMethodPainter – a method painter for the swap(int,int) method that animates the swapping of two array elements
5. The following program:

    public class BubbleSort {
        private int[] array;
public BubbleSort(int[] array) {this.array = array;}

class BubbleSort {
    private int[] array;

    public void sort() {
        for (int i = array.length - 1; i > 0; i--)
            for (int j = 0; j < i; j++)
                if (array[j] > array[j + 1])
                    swap(j, j + 1);
    }

    public void swap(int i, int j) {
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }

    public static void main(String[] args) {
        BubbleSort bs = new BubbleSort(10);
        bs.sort();
    }
}

What We Want to See
The array should be shown on screen with i and j below the array’s elements corresponding to the values of these indices. When array elements are being swapped, animations are used to move the corresponding elements.

Solution
@Viz("ex.BubbleSortPainter") //BubbleSortPainter will become the default painter of a BubbleSort object
public class BubbleSort {
    private int[] array;

    public BubbleSort(int[] array) {this.array = array;}

    public void sort() {
        for (@Viz("ex.IndexPainter") //IndexPainter will visualize i
            int i = array.length - 1; i > 0; i--)
            for (@Viz("ex.IndexPainter") //IndexPainter will visualize j
                int j = 0; j < i; j++)
                if (array[j] > array[j + 1])
                    swap(j, j + 1);
    }

    @Viz("ex.SwapMethodPainter") //This method painter will animate the swapping of array[i] and array[j]
    public void swap(int i, int j) {
        int temp = array[i];
array[i] = array[j];
array[j] = temp;
}

public static void main(String[] args) {
    @DViz //Calls for the default painter for BubbleSort to paint bs
    BubbleSort bs = new BubbleSort(10);
    bs.sort();
}

Visualizing the BubbleSort object, bs, and its field, array, is the same as shown in the previous example. The i and j variables in the sort() method are the key elements for the sort operation and should be shown as indices to the array. Hence they are annotated with IndexPainter. Lastly, the behavior of the swap method will be visualized with a SwapMethodPainter.

5.2 Navigating the Runtime Data and Painters

As a data-driven framework, ProViz extracts and provides the runtime data at every step of the execution. The job of a painter is to visualize based on the state of relevant data. Making a painter requires understanding of the runtime data structure in Viz Runtime in order to navigate and retrieve the right variables and values. This section describes the structure of the runtime data and demonstrates how a painter would access the data and other painters.

5.2.1 The Runtime Data

For a single thread program, the runtime data consist of a stack of method calls, and each method call contains a list of local variables declared in this method. Recall that a method call is modeled by a stack frame and that each local reference variable is the root of a variable tree where field variables are the descendents.

We will begin the discussion with the structure of stack frames. There are two types of stack frames depending on whether a stack frame represents an instance or a static method, which we will call instance stack frame and static stack frame respectively.
**Instance Stack Frame**

The structure of an instance stack frame is as follows:

```
[this][parameter]*[local variable]*     (* : zero or more occurrences)
```

An instance stack frame contains the “this” variable, which is a reference to the instance that this method is called up, followed by a list of parameter variables, if any, and a list of local variables, if any. Since the “this” variable references to the object on which this method is invoked, the field variables under “this” variable are the fields declared in the same class as the method. The following shows an example of an instance stack frame.

```java
class Example2 {
    private int e2Field;
    public void method(String param) {
        int local;  
    }
}
```

Given the above class definition, suppose the code below is executed.

```java
Example2 e2 = new Example2();
e2.method(" ");
```

When the execution is at the line indicated by the arrow above, the stack frame for method is as shown in Figure 15. The instance variable, e2Field, is stored under the “this” variable. The “this” variable is an alias to the object referenced by e2.

**Static Stack Frame**

The structure of a static stack frame is as follows:

```
[static variable][parameter][local variable]*     (* : zero or more occurrences)
```

A static stack frame does not have the “this” variable. Instead it begins with any static variable declared in the class, followed by the same parameters and local variables as those in an instance stack frame.

```java
class Example3 {
    private int e3Field;
    static int staticField;
```
public static void method(String param) {
    int local; //
}

When a call like Example3.method("") is executed, the resulting stack frame is as shown in Figure 16. Notice the absence of the “this” variable and also the instance variable e3Field. Moreover, the static variable staticField is stored directly in the stack frame.

A Comprehensive Example

class Demo {
    @DViz
    private int field1 = 0;
    @Viz ("MyTypePainter")
    private MyType field2 = new MyType();
    private static int staticVar;

    public void method(String paramA, int paramB) {
        MyType localVarC = new MyType();
        int localVarD; // execution pauses here
    }

    public static void main(String[] args) {
        @Viz ("DemoPainter")
        Demo demo = new Demo();
        demo.method("", 0);
    }

    class MyType {
        @DViz
        private int[] fieldE = new int[3];
        private int fieldF;
    }

    During runtime when the execution finishes executing line 9, the state of the runtime data is as shown in Figure 17.
This example visualizes the object referenced by `demo`, which has instance variables `field1` and `field2` annotated with painters. `field2` has its instance variables of `MyType` and `field2` is not annotated with painters, so they do not have painters.

**Figure 17 – Structure of the runtime data for the Demo program.**

**Figure 18 – demo’s variable tree and the painter tree that corresponds to it.**
variable fieldE visualized as an array. The resulting painters form a pruned tree of demo’s tree as shown in Figure 18.

5.2.2 Accessing Variables

Painter’s getVariable() method returns a painter’s owner variable (of type VizVariable) and serves as an entry point to the runtime data. Retrieving other variables often involves navigating the stack frame of this variable by calling getStackFrame() on this VizVariable.

Starting with the owner variable, a painter can traverse this variable’s tree and find other variables. Getting a field of a variable is to use the getField(String name) method in VizVariable where the parameter name is the name of the field variable. The parent of a variable can be accessed with the getParent() method. Using the demo object and the DemoPainter from the previous section as an example, navigations (1) from a painter to its variable, (2) within a painter tree, and (3) within a variable tree are shown in Figure 19.

![Diagram](image-url)

Figure 19 – Navigating variables and painters. In DemoPainter, the getVariable() method will return the demo variable on line 13 in the Demo class. The bold arrows show how DemoPainter can access the demo.field2 variable.

The following code demonstrates how the second element in the fieldA array can be accessed within any method in DemoPainter.

```java
class DemoPainter extends Painter {
  ...
```
... {
    //This bold line follows the bold arrows in Figure 19
    VizVariable f2Var = getVariable().getField("field2");
    VizVariable fEVar = f2Var.getField("fieldE");
    VizVariable secondElement = fEVar.getField("[1]");
}

5.2.3 Retrieving Painters

The painter for a VizVariable var can be retrieved using the getPainter(...) method in VPM:

    Painter varPainter = ProViz.getVPM().getPainter(var);

Therefore, in order to acquire a painter, we must locate its owner variable and then use it to retrieve the painter from VPM. But how a painter accesses a variable would depend on their relative positions in the program. Here we break down the access methods based on whether a painter visualizes a field or a local variable, i.e., based on a field painter or a root painter.

For a Field Painter

The access method for a field painter is simple because it can only access variables within scope, which are other instance variables. This section demonstrates how a field painter MyTypePainter, accesses another instance variable within the same class.

    class Demo {
        @DViz
        private int field1 = 0;
        @Viz ("MyTypePainter")
        private MyType field2 = new MyType();
        private static int staticVar;
        ... }

1 For simplicity purposes, the discussion in this section will assume that painters are annotated in the VOO fashion. In other words, for any object, only one of its aliasing variables is annotated and consequently has a painter. As a result, an object is visualized by at most one painter.
**Accessing field1 in MyTypePainter:**

Because `field1` and `field2` are fields of the same class, they share the same parent variable. Hence `field1` can be retrieved by traversing to `field2`'s parent and then getting the `field1` field of this parent. This traversal is accomplished through the following code and can be visualized in Figure 20.

```java
VizVariable f1Var = getVariable().getParent().getField("field1");
```

![Figure 20 – Accessing the field1 variable in MyTypePainter.](image)

**Accessing the painter for field1 in MyTypePainter:**

Because the painter structure mimics the variable structure, `field1`'s painter can be directly accessed through `MyTypePainter`'s parent (`getParent()`).

```java
Painter f1Painter = getParent().getFieldPainter("field1");
```

With `field1`'s painter in hand, we can also access the `field1` variable through this painter:

```java
VizVariable f1Var = f1Painter.getVariable();
```

The same access methods also apply to the static variable `staticVar` and its painter, if any.

**For a Root Painter**

In contrast to field painters, root painters visualizing local variables have a broader context because they are part of the executing program at runtime and are not just definitions like fields. Therefore, root painters have access to a wider range of variables.
and painters, from instance variables, other local variables, to even variables in other stack frames. Suppose the local variable, `localVarA` in method(...) is annotated with `LocalPainter` as shown below. This section describes how a root painter `LocalPainter` accesses different types of variables and painters.

class Demo {
    @DViz
    private int field1 = 0;
    @Viz ("MyTypePainter")
    private MyType field2 = new MyType();
    private static int staticVar;

    public void method(String paramA, int paramB) {
        @Viz ("LocalPainter")
        MyType localVarC = new MyType();
        int localVarD;
    }

    public static void main(String[] args) {
        @Viz ("DemoPainter")
        Demo demo = new Demo();
        demo.method("", 0);
    }
}

Accessing an instance variable, `field2`:

Painter has a built-in method for accessing instance variables, so accessing `field2` is as simple as the following line:

    VizVariable f2Var = getInstanceVariable("field2");

Access the painter for `field2`:

Because `localVarC` is a local variable in a method, it has no structural relationship to `field2`, i.e., they are in two different variable trees. Therefore, to access the painter for `field2`, `LocalPainter` needs to first obtain the `field2` variable as shown above and then use it to retrieve the painter from VPM.

    Painter f2Painter = ProViz.getVPM().getPainter(f2Var);
**Accessing a local variable in method(…), such as paramA:**

```java
VizVariable paramAVar =
    getVariable().getStackFrame().getVariable("paramA");
```

**Access the painter for paramA, if any:**

Assume the object referenced by paramA has a painter, but here paramA itself is not annotated. But to access the painter that is painting paramA’s object, we can still use paramAVar, an aliasing variable, to retrieve the painter from VPM:

```java
Painter paramAPainter = ProViz.getVPM().getPainter(paramAVar);
```

**Access demo’s painter:**

demo is a local variable in a completely different stack frame from localVarC and LocalPainter. The logical way for LocalPainter to obtain demo’s painter (DemoPainter) is to access the previous stack frame of localVarC (i.e., main(), since main(…) called method(…)), retrieve the demo variable, and then use it to get the DemoPainter.

However, all these can be simplified due to the fact that demo is the object that method(…) is invoked on, which means that the invisible “this” variable in method(…) references to demo’s object. Hence the code to access the DemoPainter is:

```java
VizVariable thisVar =
    getVariable().getStackFrame().getVariable("this");
Painter demoPainter = ProViz.getVPM().getPainter(thisVar);
```

### 5.3 Annotating Instance Variables

The examples above have relied on the fact that visualization will occur once local variables are annotated. In other words, for a given program, if an animator only annotated some instance variables of a class and executed the program, then she would see a blank screen.

Recall the rule of no dangling painter from Section 4.6.2.1. This rule enforces that ProViz visualization is top-down starting from the local variables in stack frames. Any
field variable whose parent is not visualized with a painter (i.e., does not have a painter) will not be visualized, either. This affects not only the visualization of objects but also the use of “global variables” that programmers often employ to simplify their code. When the execution enters a method, global variables for this method are the instance variables of its class. At runtime, these global variables will be stored under the “this” variable in the stack frame of the current method. Again the rule of no dangling painter prevents these variables from having painters.

This is demonstrated with the following code. Here the demo variable is not annotated with a painter. As a result, when the execution enters method(...), no painter is ever created for this code. Even though field1 and field2 are annotated with painters, their parent variable, the “this” variable in the stack frame, which points the object referenced by demo, is not visualized. Hence these instance variables are not visualized, either.

class Demo {
    @DViz
    private int field1 = 0;
    @Viz ("MyTypePainter")
    private MyType field2 = new MyType();
    private static int staticVar;

    public void method(String paramA, int paramB) {
        MyType localVarC = new MyType();
        int localVarD;
    }

    public static void main(String[] args) {
        Demo demo = new Demo();
        demo.method("", 0);
    }
}

However, the use of global variables is so common, and people sometimes do expect to see these variables once they are annotated. This topic will be further discussed in Section 6.2.4.
5.4 Making a Painter

A painter is a class that inherits from another painter class, which ultimately inherits from the Painter (viz.painters.Painter) class. Conceptually, a painter is equivalent to a rectangular graphical component (more accurately, Java Swing’s JComponent), and thus it has a location coordinate and a dimension. There is another class of painters whose base class is PainterWithNoComponent. As indicated by its name, such a painter itself has no graphical component; instead it performs visualizations by utilizing its field painters. In practice, an animator is likely to make subclasses of PainterWithNoComponent more often than those of Painter because when low-level data have been visualized, high-level painters would just make use or arrange the graphical depiction of low-level painters and therefore do not need visual components themselves.

The following subsections will discuss the five most important abstract methods in the Painter class that are relevant to the properties and the lifecycle of a painter: getComponent(), addToCanvas_userImp(), destroy_userImp(), paint_userImp(), and handleChange(Change, VizVariable). Figure 21 shows how these methods are involved in a painter’s lifecycle.

5.4.1 getComponent()

public JComponent getComponent()

This method is how Painter achieves the abstraction as a graphical component. The JComponent returned by this method is managed semi-automatically by ProViz. For instance, ProViz registers mouse listeners on this component when the painter is added to the canvas, and it automatically removes the component from the canvas when the painter is destroyed. When a painter needs to have more than one graphical component, either the components can be placed in a container, such as a JPanel, or the animator will need to manage them manually. PainterWithNoComponent, on the other hand, always returns null for this method because it does not have any component.
The following three methods have their names suffixed with "_userImp", indicating that they need to be implemented by the user. They are protected methods and are not meant to be publicly accessed. Instead, each of them has a corresponding public method without the suffix: addToCanvas(), paint(), and destroy(). In reality, each of these three public methods calls the corresponding protected methods.

### 5.4.2 addToCanvas()

**protected void addToCanvas_userImp()**

This method puts the visualization of a painter on the canvas. In other words, a painter would not appear on screen if this method were not called. There are three general tasks for this method.

1. Initialize and add the graphical component(s) of this painter to the canvas (PainterWithNoComponent can avoid this task).
2. Call addToCanvas() on the field painters of this painter.
3. Maintain the visual representation of the component(s) (very similar to the task of paint_userImp()).

Task 2 is the result of a design decision – when creating painters, VPM only calls addToCanvas() on root painters and never on field painters (as seen in Figure 21). This is due to the fact that not all field painters should always be added to the canvas. For example, consider a list that is based on an array. When the array is annotated and visualized, each field element will have a painter created for it. But as a list, the list painter may only want the elements that are present in the list to be displayed, and other array elements should remain invisible. Therefore, ProViz only calls addToCanvas() on root painters and lets every painter decide which of its field painters should be added to the canvas.

5.4.3 paint()

protected void paint_userImp()

This paint method (plus the draw() method, see Section 5.4.6) statically draws the visualization based on the state of the data. Given that a painter has been added to the canvas, the paint method needs to guarantee that the state of the graphical depiction matches the state of the runtime data that are related to this painter. In other words, it does static painting at any given time with no regards to the dynamic nature of the program data. Therefore, animations are not involved in this method.

The implementation of this method primarily involves computing and setting the sizes and locations of: (1) this painter, (2) any additional graphical components not returned by getComponent(), and (3) the field painters of this painter. Furthermore, this painter should also relay the paint() call on its field painters.

5.4.4 destroy()

protected void destroy_userImp()

This method is called by ProViz when a painter is destroyed and removed from the canvas. The public version of this method, destroy(), automatically removes the graphical component that is returned by getComponent() and also deregisters the
mouse listeners. Therefore, only when a painter has additional components, it needs to manually remove them in destroy_userImp(). Furthermore, unlike addToCanvas(), the destroy call does not need to be relayed down to the field painters. In fact, the implementation of this method is blank in most painters we have. And as mentioned, the majority of our painters are PainterWithNoComponent, which means that there is no graphical component that needs to be removed.

5.4.5 **handleChange(Change, VizVariable)**

```java
public void handleChange(Change, VizVariable)
```

ProViz implements the event-listener model for relaying data changes to painters, and it is in this method that a painter responds to changes. The first parameter describes the type of change, and the second parameter is the source variable that generates this change event.

**Calling addToCanvas() on Field Painters**

Recall from Section 4.6.2.1 that all field painters under a changed variable will be recreated as new painter objects. The old field painters are all destroyed and removed from the canvas. By the time handleChange(...) is called, new painters have already replaced the old ones, but they have not been added to the canvas. So the most important task of handleChange(...) is to call addToCanvas() on field painters. For PainterWithNoComponent, this can be easily accomplished by calling this painter’s addToCanvas() method. But for Painter, because addToCanvas() does add a graphical component to the canvas, it cannot be called again while the component is still on the canvas. So a subclass of Painter would need to call
addToCanvas() individually on its field painters, and then a paint() call would maintain the graphical integrity of this painter (as seen in Figure 22).

Performing Animations

Animations are made to show the transition of changes and therefore are carried out in handleChange(...). How animations are accomplished in ProViz is described in Section 5.5.

5.4.6 draw(Graphics)

protected void draw_userImp(Graphics)

Other than the five abstract methods mentioned above, the draw(...) method is also worth mentioning as it is related to the painting task. The existence of this method is the result of how Java Swing allows custom painting in graphical components. To paint on a Java Component (or JComponent), one needs to create a subclass of Component and override the paint(Graphics) method. Then custom painting and geometric transformations can be applied to the graphical context and finally be rendered and displayed. For a painter to directly draw on a canvas, it needs to register with the canvas (getCanvas().addPainterToBePainted()) so that this canvas will call the draw(...) method of this registered painter and forward the graphical context through the parameter. As a consequence, the painter can draw anything onto the canvas within this method. When a painter is destroyed, it is automatically de-registered with the canvas in destroy() so the animator does not need to be responsible for the de-registration.

5.4.7 Example: StackPainter

This section will demonstrate the detailed process of making a painter that visualizes the java.util.Stack class.

Annotating the Stack Class with Viz XML

How does ProViz visualize the Stack class (java.util.Stack) in Java? It is a library class and that one may not access or compile its source code. Hence an animator cannot assign painters to the fields or methods of this class through Viz annotations. The
only option is to use Viz XML to specify the painters for its programming elements. But the main question is: how do we know the structure of `Stack`, since it is a library class?

Without looking at the source code for `Stack`, we can still find out about its structure using the debugger’s Variable View by making a test program and stepping through it. From the snapshot in Figure 23, we can see that `Stack` uses (1) an array called “`elementData`” as its underlying data structure and (2) an integer called “`elementCount`” that serves as the size of the stack.

The XML code below specifies that the `Stack` class will be visualized with `StackPainter`. Furthermore, the `elementData` field (which is an array) will be visualized by a `BridgePainter`, which is a ProViz library painter that does nothing but simply bridges between its field painter(s) and its parent painter. In ProViz, when an array variable is annotated with a painter, ProViz will automatically generate field painters for this array’s elements. Thus the `BridgePainter` will have field painters for all the elements of `elementData`. Lastly, `elementCount` will not be visualized, but we need to know about it because later on, `StackPainter` will need to access it for size information.

```xml
<type name="java.util.Stack&lt;E&gt;">  
  <viz vc="viz.lib.java.list.StackPainter" />
  <field name="elementData">
```

---

1 In an XML file, the real name would be: `java.util.Stack&lt;E&gt;`;
2 The `field` element can have an optional “`type`” attribute. The actual type of `elementData` is unknown, but it could be an `E` array. If the type were to be included, the `field` element could be: `<field name="elementData" type="E[\]">`
This XML-defined annotation associates a painter with a class, which is equivalent to the class annotation discussed in Section 5.1.2. Consequently, it defines the default visualization for Stack, and @DViz can now be used to annotate any Stack variable for visualization.

**Conceptual Design**

Suppose a Stack variable in an arbitrary program is declared and annotated as follows:

```java
Stack<String> stackVar = new Stack<String>();
stackVar.push("A");
stackVar.push("B");
```

With the above scheme of annotating the Stack class, we can foresee the runtime data and the painter structure of stackVar as depicted in Figure 24. The StackPainter visualizes stackVar, BridgePainter is assigned to elementData, and as many StringPainter’s as the size of the elementData array will be the field painters of the BridgePainter (the size is initially 10 prior to any expansion).

The conceptual design of the stack is as depicted in the “ProViz View” in Figure 24. Each element of elementData has a painter assigned to it, but not all of these painters shall appear on the canvas. Only those that belong to the abstract stack should be displayed, and others should not be visible. This is the task of StackPainter, which needs to control these element painters by placing or not placing them on screen. Moreover, StackPainter will draw (1) the three lines that form a container shape, (2) the text “Stack stackVar” right underneath the container, and (3) the text “top = X” whose position and the number X should be updated whenever the stack is modified.
StackPainter

To begin with, StackPainter will inherit from ProViz’s VariablePainter, which paints a variable’s type and name (the number two item from the above list). Secondly, a label is needed for the “top = X” display, and this label will not be returned by getComponent(), which already returns the component for the variable’s type and name. Therefore, this label needs to be manually managed by StackPainter, and it will be added to the canvas in addToCanvas_userImp() and removed in destroy_userImp(). Lastly, the three lines forming the container will be drawn in the draw(Graphics) method. The remaining of this section will show and explain the implementation of StackPainter’s methods. First the class’ signature, fields, and destroy_userImp() are shown below.

public class StackPainter extends VariablePainter {

Figure 24 – Structures of the runtime data and painters and the conceptual design of stack visualization
//labelEnd’s text is “top = X”
private VLabel labelEnd = new VLabel("top");

//Used to keep track of whether labelEnd is on the canvas
private boolean isAdded = false;

//To hold the array’s size for the draw(Graphics) method
private int arraySize = -1;

//Width of the stack
private int width = 50;

protected void destroy_userImp() {
    super.destroy_userImp();
    //Removes the labelEnd from the canvas
    getCanvas().remove(labelEnd);
    isAdded = false;
}

... //see below

addToCanvas_userImp()

1 protected void addToCanvas_userImp() {
2     if (!getVariable().isNull()) {
3         super.addToCanvas_userImp();
4         if (!isAdded) {
5             getCanvas().add(labelEnd);
6             isAdded = true;
7         }
8     }
9     VizVariable sizeVar =
10         getVariable().getField("elementCount");
11     sizeVar.addListener(this); //Listens to the
12     addEventGenerator(sizeVar); //elementCount variable
13     //Gets the BridgePainter for elementData
14     Painter array = getFieldPainter("elementData");
15     int size = Integer.parseInt(sizeVar.getValueAsString());
16     for (int i = 0; i < size; i++) {
17         array.getFieldPainter("[" + i + "]").addToCanvas();
18     }
19     paint();
20     getCanvas().addPainterToBePainted(this);
21 }

• From line 4 to 7, labelEnd is added to the canvas. isAdded is used as a flag here to prevent multiple invocations of this method from throwing an exception because the canvas, a Java Swing container, does not allow adding the same
component multiple times. Note that `labelEnd`’s position is not set here, but it will be in the `paint()` method, which is called on line 16.

- Lines 8 through 10 register this `StackPainter` as a listener to the `elementCount` variable. So whenever `elementCount` changes, `StackPainter`’s `handleChange(…)` will receive a change event.

- Lines 11 through 15 add the element painters that belong to the stack to the canvas. These are field painters of the `BridgePainter` whose names are from `[0]` up to `[elementCount - 1].`

- Line 16 calls `paint()` to arrange the positions and sizes of all painters and components.

- Line 17 initiates the canvas to call the `draw(Graphics)` method of this painter.

**paint_userImp()**

```java
protected void paint_userImp() {
    if (!getVariable().isNull()) {
        Painter array = getFieldPainter("elementData");
        arraySize = array.getFieldPainters().size();
        VizVariable sizeVar =
            this.getVariable().getField("elementCount");
        int size = Integer.parseInt(sizeVar.getValueAsString());
        int y = getLocation().y - 25;
        for (int i = 0; i < size; i++) {
            Painter element = array.getFieldPainter("[" + i + "]");
            element.setLocation(getLocation().x, y);
            element.setSize(width, element.getHeight());
            y -= element.getHeight();
        }
        if (size > 0) {
            Painter last = array.getFieldPainter("[" + (size-1) + "]");
            labelEnd.setText("top = " + size);
            labelEnd.setBounds(last.getLocation().x,
                last.getLocation().y - 25,
                labelEnd.getPreferredSize().width, 15);
        } else {
            labelEnd.setSize(0, 0);
        }
    }
}
```
- On line 4, `arraySize` is updated so that the `draw(Graphics)` method can draw correctly.
- Lines 5 and 6 get the value of `elementCount`, and then lines from 7 to 13 iterate through each element painter in the stack and set their positions and sizes.
- The rest of the code calculates the position and updates the text of `labelEnd`. The position is above the top element in the stack, which is retrieved on line 15.

`handleChange(Change, VizVariable)`

```java
public void handleChange(Change change, VizVariable source) {
    if (source.getName().equals("elementCount")) {
        Painter array = getFieldPainter("elementData");
        for (int i = 0; i < dummy.getFieldPainters().size(); i++) {
            array.getFieldPainter("[" + i + "]").destroy();
        }
        int size = Integer.parseInt(source.getValueAsString());
        for (int i = 0; i < size; i++) {
            array.getFieldPainter("[" + i + "]").addToCanvas();
        }
        paint();
        getCanvas().repaint();
    } else if (change == Change.TO_NULL) {
        getCanvas().removePainterToBePainted(this);
        destroy();
    } else {
        addToCanvas();
    }
}
```

- This first if block from lines 2 to 12 handles the change event generated from the `elementCount` variable. Since `elementCount` represents the number of elements in the stack, this change occurs when the stack elements are being pushed or popped. The implementation here does not respond to individual push or pop events. Instead it simply refreshes the stack display by destroying all element painters of the array (lines 4 through 6) and then adding the stack elements back on to the canvas (lines 7 through 10). Although this strategy is less efficient, such an implementation is much safer because one cannot assume that the stack elements would only be pushed or popped one at a time (the fact is that `Stack` is a subclass
of `java.util.Vector` and supports other add and remove methods that do not operate at the top of the stack).

- When the variable of this painter is set to the `null` value, this results in the `Change.TO_NULL` event on line 13. The current implementation will take out the stack visualization completely by telling the canvas to stop calling this painter’s `draw(Graphics)` method on line 14 and removing all the components of this painter by calling `destroy()` on line 15. Note that all the element painters of the stack have been destroyed as a result of being the field painters of a changed variable (recall from Sections 4.6.2.1 and 5.4.5).

- In the else block from lines 16 to 18, the only possible event is a change in the owner variable `stackVar`. Since all field painters of this painter are recreated, the goal is to call `addToCanvas()` on each field painter of the stack. However, we have designed the `addToCanvas_userImp()` method to use the `isAdded` flag to prevent multiple invocations of this method from throwing exceptions. Therefore, we can simply call this painter’s `addToCanvas()` here without a problem.

```java
draw(Graphics)
1  public void draw(Graphics g) {
2      int x = this.getLocation().x - 2;
3      int y = this.getLocation().y - 2;
4      g.drawLine(x, y, x, y - arraySize * 25);
5      g.drawLine(x, y, x + width + 3, y);
6      g.drawLine(x + width + 3, y, x + width + 3, y - arraySize * 25);
7  }
```

This method simply draws three lines depicting like a container. The height of the vertical lines depends on the size of the `elementData` array (not the size of the stack). This is achieved through the instance variable, `arraySize`, which is updated in `paint_userImp()`.

Figure 25 shows an actual ProViz visualization of a Stack using what was described in this section.
The role of animation in PV was once a topic under debate. Many supported its usefulness in visualization (Tudoreanu, 2003; Yeh, Greyling, & Cilliers, 2006) while some studies concluded that it has little effect on learning (Byrne, Catrambone, & Stasko, 1996; Lawrence, Badre, & Stasko, 1994). This researcher believes in animation’s effect in raising awareness about graphical changes and the semantic meaning it can produce for actions. Furthermore, in today’s technology level, animations are relatively cheap and fundamental. Therefore, it is natural or even expected from the user to include animations in the visualization. In fact, ProViz has the option of turning off the animation and can satisfy both sides.

Animations are managed by the animation controller (viz.animation.AnimationController). There are two types of animations with different timings that can be performed by the animation controller. The first one is through the method scheduleAnimation(...), which accumulates sequential and simultaneous animations, and then the controller will carry them out at the end of an execution step. The second one is animateNow(...), which immediately performs one or several simultaneous animations.

5.5 Animation

The role of animations in PV was once a topic under debate. Many supported its usefulness in visualization (Tudoreanu, 2003; Yeh, Greyling, & Cilliers, 2006) while some studies concluded that it has little effect on learning (Byrne, Catrambone, & Stasko, 1996; Lawrence, Badre, & Stasko, 1994). This researcher believes in animation’s effect in raising awareness about graphical changes and the semantic meaning it can produce for actions. Furthermore, in today’s technology level, animations are relatively cheap and fundamental. Therefore, it is natural or even expected from the user to include animations in the visualization. In fact, ProViz has the option of turning off the animation and can satisfy both sides.

Animations are managed by the animation controller (viz.animation.AnimationController). There are two types of animations with different timings that can be performed by the animation controller. The first one is through the method scheduleAnimation(...), which accumulates sequential and simultaneous animations, and then the controller will carry them out at the end of an execution step. The second one is animateNow(...), which immediately performs one or several simultaneous animations.
Instant Animations

animateNow(boolean repaint, Motion... motions)

animateNow(...) performs the animation immediately using ProViz’s main thread, meaning that it will block the entire program until the animation finishes. It is suitable to use when the animator does not want other animations to interfere with this method’s animations.

The second parameter of animateNow(...) is a Java vararg, which is a list of parameters of Motion objects (viz.animation.motion.Motion) or an array of Motion. A Motion is a graphical transformation, which can be a movement, rotation, scaling, and so on. When a list of motions are passed into an animateNow(...) call, these motions will be performed simultaneously. The second parameter of scheduleAnimation(...) has the same usage and is described next.

Queued Animations

scheduleAnimation(IPainter monitor, Motion... motions)

At each execution step, the animations scheduled with scheduleAnimation(...) are performed at the end of the step after Viz Runtime and VPM have finished updating respective data in step processing, as demonstrated in Figure 26.

![Figure 26 – Animation threads and the timeline for a program step](image-url)
Like `animateNow(...)` the second parameters of `scheduleAnimation(...)` is a list of motions, and these motions will be visualized simultaneously (such as monitor A’s `enlarge A` and `move C` motions and monitor B’s `move B` and `rotate B` motions in Figure 26).

The first parameter of the method is a monitor. Any painter can act as a monitor for scheduling sequential motions. Multiple invocations of this method with the same monitor will result in sequential animations, and scheduling with different monitors will result in as many simultaneous animations. The example in Figure 26 has a total of six `scheduleAnimation(...)` calls on three monitors: two on monitor A, one on B, and three on C. These three monitors result in three simultaneous animations. Additionally, since each call can have its own list of simultaneous motions, there are five motions executing simultaneously at the beginning of this example.

**Simultaneous Animations**

An animation is made by creating an object of `Motion` and letting the animation controller (`ProViz.getAnimationController()`) execute this `Motion` object through either of the above two methods. Figure 27 uses `LinearPath` and `CurvePath` as examples to schedule two simultaneous animations (both `LinearPath` and `CurvePath` are subtypes of `Motion`). `LinearPath` moves a painter in a straight line, and `CurvePath` moves it in a curved, semi-circular path.

```
Motion m1 = new LinearPath(painterA, src, dest, null);
Motion m2 = new CurvePath(painterB, src, dest, Direction.UP);
ProViz.getAnimationController().
    scheduleAnimation(painterA, new Motion[] {m1, m2});
```

![Diagram](image.png)

**Figure 27 – An example of scheduling simultaneous animations.**
Since the second parameter of `scheduleAnimation(…)` is a vararg, the method call in Figure 27 is equivalent to the following:

```java
...scheduleAnimation(painterA, m1, m2);
```

**Sequential Animations**

To have animations queued up and be performed sequentially, one should call `scheduleAnimation(…)` multiple times on the same monitor, e.g.:

```java
...scheduleAnimation(painterA, m1);
...scheduleAnimation(painterA, m2);
```

### 5.6 Making a Method Painter

A method painter is a subclass of `MethodPainter(viz.painters.method.MethodPainter)`. A method painter, once created, is associated with a stack frame, which can be accessed using the `getStackFrame()` method.

#### 5.6.1 `methodInvoked()` and `methodReturned()`

The concept of a method painter is slightly different from that of variable painters due to the event-like nature of a method. Two events can occur to a method call: (1) a method is invoked (i.e., a new stack frame is created, and the painters for parameter variables are also created and added to the canvas) and (2) the method returns. These events are dispatched to two methods in `MethodPainter`: `methodInvoked()` and `methodReturned()`, respectively. Figure 28 shows the lifetime of a `MethodPainter` relative to the timeline of these methods.

`MethodPainter` uses the same implementation as the `Painter` class where a painter can be associated with a graphical component, returned by the `getComponent()` method, and the `addToCanvas()` and `destroy()` methods will operate on this component. `addToCanvas()` in `MethodPainter` initializes the graphical component (but not adding it to the canvas) and calls `methodInvoked()`, making `methodInvoked()` equivalent to `addToCanvas_userImp()` in `Painter`.
In MethodPainter, however, the destroy() and methodReturned() methods function differently from destroy() and destroy_userImp() in Painter. As shown in Figure 28, destroy() and methodReturned() are called separately by ProViz at different times when the method returns. methodReturned() is called first after the method is returned but before all the variable painters in this method are destroyed. Hence in methodReturned(), the animator can still access and manipulate these residual variable painters. Then at the end of a step after the step animations are carried out, destroy() is called to ensure that the graphical component of this MethodPainter is removed from the canvas. In a nutshell, destroy() does not call methodReturned() in MethodPainter.

Figure 28 – The lifetime of a method painter relative to the timeline of method processing in ProViz.
Although a method painter can be associated with a graphical component like a variable painter, the reality is that most methods are events and their visualizations merely involve manipulation of other variable painters. In our experience, method painters for recursive methods are more likely to have graphical depictions of their own. Other than that, method painters’ `getComponent()` generally returns null, which is what the default implementation of `MethodPainter` does.

### 5.6.2 `shouldContinueMethod()`

When the execution enters a new method, a method stack frame is created. If this method is annotated with a painter, a corresponding method painter will be created, and `methodInvoked()` will be executed. After `methodInvoked()` is executed, the animator will have a choice to either continue executing the method or skip it and return immediately using the *step return* command. This choice is specified by this method: `shouldContinueMethod()`. Because skipping a method is more efficient for ProViz, the default implementation for this method returns `false`, which will step return the target method immediately after `methodInvoked()` is executed. For a method painter that wants ProViz to execute this method and visualize the method body, its `shouldContinueMethod()` must be overridden and must return `true`.

### 5.6.3 Monitoring Variable Changes

The invocation and return events of a method can describe, at high level, what this method does. And theoretically, what happens in a method can be further detailed by the visualization of variable painters within this method. But in reality, high-level visualizations often require special visual events to be carried out, which may not be part of the design of these variable painters. As a consequence, new variable painters need to be made and customized to display these visualizations. In other words, such an approach is an attempt to achieve event-driven visualizations with variable painters, which may violate the data-driven design of painters, making them context-specific and un-reusable.
To counter such an inconvenience, method painters are given the capability of monitoring and responding to variable changes within its method. As a consequence, a method painter can become a monitor for its variables and can perform visualizations before or after they change. Method painters can create method actions (viz.painters.method.MethodAction) and make them respond to variable changes using two methods: addBeforeAction(...) and addAfterAction(...). During runtime when the execution is within a method and a monitored variable is changed, the before actions will be performed first before the variable painter visualizes this change event. Then after the variable painter finishes its visualization, the after action is executed. The sequence of these events is displayed in Figure 29.

Same Method Step & Variable Changed

I. Before Actions
   - animateNow()
   - scheduleAnimation()

II. Variable Painters Visualization
   - animateNow()
   - scheduleAnimation()

III. Step Animation

IV. After Actions
   - animateNow()
   - scheduleAnimation()

A. Animation performed instantly
B. Animation performed instantly
C. Sequential and simultaneous animations for step animation
D. Animation performed instantly
E. Sequential and simultaneous animations for after actions

Figure 29 – Sequence of processing in a same method step.
5.6.4 Animation Sequence

Figure 29 shows the correlation between the sequence of processes and the sequence of animations in a same method step. Any animateNow(...) call in a method painter’s before action (I) will be performed (A) before any animateNow(...) called by variable painters (II, B). Then any animation registered using scheduleAnimation(...) in either before actions or variable painters (I or II) will be carried out next in step animation (III, C). After actions’ (IV) instant animations (D) are performed subsequent to step animation. Finally, animations scheduled by after actions (E) are the last to be executed.

Note that the purpose of having before and after actions is to be able to visualize before or after the visualization of variable painters but not before or after the data changes. The entire runtime model in Viz Runtime has already been updated before VPM begins processing painters. Hence all the runtime data available to all painters are up-to-date.

5.6.5 Example: PushMethodPainter

Following up the StackPainter example, suppose we want to visualize the push method in the Stack class by making PushMethodPainter. Yet again, Viz XML is the only option for annotating the method because Stack is a library class. Annotating a method in Viz XML requires the exact signature of the method. To find out about this information is to use the debugger to step through a test program. Below is a skeleton of the Stack class, the two aforementioned fields, and the push method.

```java
class Stack<E> {
   E[] elementData;
   int elementCount;
   public E push(E arg0) {...}
}
```

This skeleton can help the maker of PushMethodPainter to know about the instance variables so that she can retrieve necessary data in the method painter.

To compose the annotation in XML, the ID of a method needs to be in a ProViz-specific format. As discussed in Section 4.3.2, the ID is the fully qualified class name
followed by a colon “:”, the method name, and a list of parameter’s types (also fully qualified) separated by commas and enclosed in parentheses. For the push method, its ProViz method ID would be:

```
java.util.Stack<E>:push(E)
```

With the method’s signature and ID in hand, we can compose the XML and assign PushMethodPainter to it:

```
<type name="java.util.Stack<E">">
    <viz vc="viz.painters.java.list.StackPainter" />
    <field name="elementData">
        <viz vc="viz.painters.DummyPainter" />
    </field>
    <method name="java.util.Stack<E>:push(E)">
        <viz vc="viz.lib.java.list.PushMethodPainter" />
        <param name="arg0" type="E">
            <viz vc="1D" />
        </param>
    </method>
</type>
```

In addition to associating the method painter with the method, we also want to visualize the parameter element, arg0. Hence it is assigned with the keyword for the default visualization, “1D.” To write the above XML equivalently using Viz annotations, it would be:

```
@Viz ("viz.lib.java.list.PushMethodPainter")
public E push(@DViz E arg0) {...}
```

Conceptual Design

What we want with the push method painter is simple. The new element that is being added to the stack will appear at the top of the three-line container and move downwards to the correct place at the top of the stack. This will be implemented in a single animation frame, which is better achieved through animateNow(...). If this were to be accomplished with scheduleAnimation(...), it would take the risk of mixing the animation sequence with other scheduled animations from, in this case, the parameter
painter. Moreover, the visualization will be composed in methodInvoked(), as we want the animation to be performed when the push method is called.

**PushMethodPainter**

The task of this painter is to: (1) locate the top of the three-line container and place the parameter painter there; (2) locate the spot above the current top element of the stack where the new element should be pushed to, and this will be the destination for the parameter painter; (3) with the above two coordinates, the parameter painter, i.e., arg0’s painter, will be moved with animation.

```java
1 public class PushMethodPainter extends MethodPainter {
2     //Constructor omitted here
3     public void methodInvoked() {
4         VizVariable arrayVar = getInstanceVariable("elementData");
5         Painter arrayPainter = ProViz.getVPM().getPainter(arrayVar);
6         VizVariable elementCount = getInstanceVariable("elementCount");
7         int count = Integer.parseInt(elementCount.getValueAsString());
8         Painter param = ProViz.getVPM().getPainter(getStackFrame().getVariable("arg0")).
9             getPainter(getStackFrame().getVariable("arg0")).
10            Point orig = arrayPainter.getParent().getLocation();
11            Point dest = new Point(orig);
12            if (count > 0) {
13                Painter topPainter = arrayPainter.getFieldPainter("[" + (count - 1) + "]");
14                dest.y = topPainter.getLocation().y;
15            }
16            dest.y -= param.getHeight();
17            param.setLocation(orig.x, orig.y - arrayVar.getFields().size() * 25 - 30);
18            param.setSize(arrayPainter.getParent().getWidth(),
19                param.getHeight());
20            LinearPath path = new LinearPath(param, param.getLocation(), dest, null);
21            ProViz.getAnimationController().animateNow(false, path);
22     }
```

- Lines 3 to 7 retrieve the relevant data and painters. The arrayPainter on line 4 is the DummyPainter for the elementData array, and we need it to access the top element painter of the stack. elementCount and its value are acquired on lines 5 and 6 and arg0’s painter is retrieved on line 7.
On line 8, `arrayPainter.getParent()` will return the `StackPainter`. Recall from Figure 25 that the location of this `StackPainter` is the bottom-left corner of the container.

Lines 8 through 14 are calculating the destination for `arg0`’s painter.

Lines 15 and 16 are placing `arg0`’s painter at the top of the container. The height of the container is calculated the same way it does in `StackPainter`’s `draw(...)` method.

Lines 17 and 18 create a linear path and perform the animation immediately.

### 5.7 Other Features and Discussion

#### 5.7.1 Connectors

It is expected that the visualization would frequently involve lines connecting between graphical components. So ProViz has a connector mechanism built-in to a canvas (`viz.views.VizCanvas`). Each `VizCanvas` has a connector manager (`viz.views.ConnectorManager`), and a painter can request a line connecting itself to another painter. Multiple connectors between two components (for example, a next link and a previous link between two nodes in a doubly-linked list) can also be made by giving a string identifier to each connector.

#### 5.7.2 Interactivity

One established aspect about PV systems is that interactivity is necessary to support effective learning (Tudoreanu, 2003; Lawrence, Badre, & Stasko, 1994; Bednarik, Moreno, Myller, & Sutinen, 2005). Passively viewing visualization rarely achieves learning for the viewer, especially for students who do not yet have the cognitive mechanics to derive the necessary knowledge (Furcy, Naps, & Wentworth, 2008). Interactivity allows the viewer to actively change and modify the visualization as they need so that the visualization can better fulfill their mental models for comprehension. The design of painters has yet again given us a great opportunity of enhancing the interactivity with the viewer without compromising the integrity of the program visualization.
Right-Click Menu

ProViz automatically registers mouse listeners on the graphical component returned by `getComponent()` in a painter, and the right-click menu is made standard for all painters. Right-clicking on a painter’s component on screen will reveal the context menu. This menu contains two types of options: system-defined options and painter-specific options. System-defined options include repaint (by calling the painter’s `paint()` method) and options to switch this painter to a different type. Painter-specific options, on the other hand, can be defined by a painter to perform painter specific tasks. For example, ProViz’s `ArrayPainter` has the options of changing the orientation of the array and also showing/hiding the indices for the elements.

To add options in a painter’s right-click menu, the `getActionCommands()` method should return an array of strings that are the labels of these options and also the action commands of these actions. Then an action listener should be returned by the `getRightClickActionListener()` method, and this action listener shall respond to the options.

Figure 30 shows a painter visualizing a `java.util.TreeMap` object. The painter defined two custom actions circled in red: `Change Orientation` and `Show Null` (hence its `getActionCommands()` returns the string array: “Change Orientation”, “Show Null”). The user can interact with the tree painters through these options, which will manipulate the tree accordingly.

![Figure 30 – Example of a context (right-click) menu on a painter.](image)
Dynamic Painter Switch

The advantage of a customizable mapping mechanism in a data-driven system is that the visualizers can be dynamically switched in order to display different depictions of a variable. In ProViz, a painter can be dynamically swapped to a different type of painter in the context menu (the green circle in Figure 30), and what type of painters it can be switched to is defined by the annotations in the Viz Map Model. For instance, suppose a variable is annotated with: @Viz ("PainterA, PainterB, PainterC"). During runtime, its default painter will be a PainterA. Then the switch-painter option in the right-click menu will show PainterB and PainterC as the choices to change the painter.

ProViz is still in the development stage with this painter switching mechanism because the painter dynamics with field painters would require extra hooks that swap field painters along with their parent painters. Otherwise the swap could be erroneous (as indicated with the text “w/ Caution!” in Figure 30). For example, suppose there are two painters A and B that can visualize the same data type. Painter A is designed to work with a field painter of type $F_A$ and painter B with $F_B$. Suppose painter A is to be switched to painter B, then the field painter of this object should also be switched from $F_A$ to $F_B$, otherwise B may not function as expected.

5.7.3 Recording and Playback

The recording mechanism in ProViz is very simple. First of all, ProViz directly records the stack frames and variable object to a file using Java’s ObjectOutputStream, meaning that raw objects are written to the file. Secondly, it records the stream of unprocessed runtime data (prior to being processed by Viz Runtime) to the file. This can be improved by recording only the changed, already processed data, but the challenge is to keep the persistency of VizVariable’s and the event-listener model. For the scope of this study, however, this improvement is given less priority and is considered as a future research opportunity.
Preserved Interactivity in Playback

The advantage of the current recording scheme is that the playback using Viz Player preserves the exact same interactivity (minus the connection with the source code and unenclosed painter classes) as the original program’s visualization. In fact, Viz Player can be considered an alternative of Viz Monitor. Instead of getting its data source from a program execution and the debugger, Viz Player reads from the recording file. Then the rest of the visualization process is exactly the same as that when ProViz visualizes the program live. As a result, what Viz Player plays back is not a compressed video but interactive visualizations. This allows different user interactions on each playback and thus can provide a different user experience every time.
Chapter 6
Evaluation

Recall that the objective of ProViz is to simplify the PV process for creating high-level program visualizations. The prototype has shown its capability of outputting high-level visualizations with animation. What we intend to find out is its effectiveness in facilitating the visualization process. First, a pilot study was conducted in an introductory computer science class to see if an instructor can use ProViz to create visualizations for pedagogical purposes. Then, a usability test was administered to investigate the effectiveness of ProViz in facilitating general program visualization tasks.

It is usually desirable for a test to compare a new system with other systems. However, for an innovative system like ProViz, there is no other system with which it can be directly compared. First of all, event-driven PV systems inherit the problems mentioned in Section 2.4.1, and ProViz is designed to avoid these. Therefore, a comparison with any event-driven PV system would be unfair. On the other hand, no other declarative PV systems have a flexible mapping mechanism like Viz Map, and their learning curve may be high because they require learning other programming languages: Pavane requires logic programming; LEONARDO uses the ALPHA scripting language; InspectJ requires aspect-oriented programming with AspectJ. It is not practical to recruit subjects to learn any of these systems plus ProViz in order to compare them. It is more realistic to perform a usability test on ProViz to see whether its concept and application in the visualization process can be adopted by programmers so that they can perform PV tasks efficiently.

6.1 Pilot Study

There are two usages of ProViz in this pilot study. First, sample programs mentioned in the lecture are visualized to demonstrate how they work. Second, visualizations are made for programming assignments and are displayed to students when the assignments are posted. The visualizations show students how the program should work and what they are
supposed to do before they begin programming. This section presents two case studies that have shown the value of ProViz visualization.

**Case Study I – Different Levels of Abstractions**

ProViz presents a unique opportunity for creating multiple visualizations for the same program or algorithm. Because the visualization process is simplified, in the pilot study we were able to create, for the same program, visualizations at different levels of abstraction for demonstration to students. Figure 31 shows several visualizations, from high to low levels of abstraction, for one stack program. 31(a) has the highest abstraction, which displays the stack as a stack of plates, because the concept of a stack can be demonstrated by the practical act of placing and removing plates one at a time; 31(b) replaces the plates with values and has the normal presentation abstraction that most computer scientists would perceive about a stack; 31(c) and 31(d) then show the low-level, underlying data structure for the stack program, which consists of an array and an index variable, \( \text{top} \) (the real visualization is dynamic with animation that demonstrates the operations of a stack, but only static snapshots can be displayed here).

These views are presented to students in the same order. The high-level view with plates relates the concept and operation of a stack to real-life plates so the students can grasp what a stack is like; 31(b) demonstrates that the stack data structure can hold a collection of values; 31(c) and 31(d) then tell them how this stack can be implemented using an array and an index. Furthermore, when the array is full, the stack will be copied to a bigger array in this stack program, and that is also visualized so that the students can see what happens when the array is full.

![Visualizations of a stack in different abstractions](image)

*Figure 31 – Visualizations of a stack in different abstractions*
Transforming high-level concepts into implementation is often confusing for students, and visualizations in different abstractions are the perfect tool for guiding them through the process. Once again, these visualizations can be made for a single program in a timely manner because ProViz has fulfilled its goal of reducing the time and effort required for visualization creation (Wu & Stelovsky, in press).

Case Study II – Rapid Customization

Figure 32 shows a snapshot of the Sieve of Erastothenes algorithm, which is a programming assignment for the students. The visualization was shown to students when the assignment was introduced. It was meant to illustrate the concept of the algorithm following the assignment statement:

Write a Java program that computes all the prime numbers that are less than n using the Sieve of Erastothenes algorithm. This is an informal description of the algorithm:

1. Write down all integer numbers between 2 and n. Color them blue.
2. Pick the first blue number (initially 2) - it is a prime, so color it green.
3. Color red all multiples of this prime.
4. Repeat from step 2 until the first blue number is greater than the square root of n.
5. When finished, all the green and blue numbers are prime.

Hint: You can use an array of boolean values, e.g., where false means "it's red, i.e., a multiple, i.e., not prime" and true means "it's green or blue, i.e., prime or possible prime". Whether it's blue or green is simply a matter of where you are in the list of numbers.

After students viewed the visualization and started programming, some students raised the question of how to represent three different colors. Within seconds in front of the class, the coloring in the visualization was taken out (as shown in Figure 33) to emphasize the fact that colors are just abstract concepts, and that the values of true and false are sufficient to represent the functionality of the three colors.

This example clearly shows the strength of ProViz, which allows the visualization to be customized easily in a timely manner to meet the perception needs of its viewers.
Figures 32 – Visualization of the Sieve of Erasthothenes algorithm for finding prime numbers

Figures 33 – Visualization of Sieve of Erasthothenes without color
6.2 Usability Test

The objective of this test is to find out if programmers can adopt the data-driven concept with painter annotations and can create or customize visualizations with a manageable amount of effort.

6.2.1 Experimental Setup

Six people participated in this usability test. They were asked to read the online documentation\(^1\) of ProViz days prior to the test, and this documentation was also used as reference material for them to look up information during the test. The experiments were carried out on a laptop that had Eclipse 3.6 (32 bit) and ProViz installed and ready for the test.

In order to use ProViz, the participants had to have knowledge of and programming experience with Java, but experience with the Eclipse IDE was not required as I would provide technical assistance for using Eclipse when necessary. Most importantly, all the participants were first-time users and had never used ProViz prior to the experiment. Hence the performances of the participants were not biased by any past experience with the framework.

At the beginning of each experiment, the participants were given a brief introduction on how to use ProViz and Eclipse to ensure that they had fundamental knowledge of the mechanics of using both tools. A checklist was given to them in order to assist them in the visualization procedure. The participants were also encouraged to write down comments and thoughts on the test sheets. During the experiment, I was available to assist with technical procedures, such as converting a string to an integer in Java, switching between different perspectives in Eclipse, making sure that Eclipse executes the right program, and so on. At the end of the experiment, a questionnaire was used to gather the participants’ background information regarding software development.

\(^1\) http://www2.hawaii.edu/~johnwu/ProViz. The content of the online documentation is similar to Sections 5.1 and 5.2 in this dissertation.
6.2.2 Tasks

Originally, six tasks were designed for the test, but in order to keep the experiment under an hour, Task 3 was considered optional and was only performed when the participants had time and were willing to continue. These tasks, described below, are designed to test a wide spectrum of concepts and functionalities of ProViz, which fall into three groups:

1. Annotation-related

   Task 1. Annotate necessary elements of a program and visualize it.

   Task 2. Modify the visualization by altering the Viz Map Model in ProViz Editor.

2. Visual modifications

   Task 3. Make vertical positions of array elements to their values in a bubble sort program to have the “bubble up” effect. (Two out of six participants did not perform this task due to time constraints).

   Task 4. Modify simple animations.

3. Painter implementation and application

   Task 5. Change elements of a stack from string values to images by implementing an image painter and applying it to the visualization.

   Task 6. Implement a painter that requires navigating the runtime model.

   The tasks in Group 1 employ a scenario where painters are pre-made and need to be annotated to the right elements. Group 2 requires graphical programming to be performed by the participants. They will need to comprehend the original programming of a painter before making changes. Group 3 requires the participants to create new painters by inheriting from existing painters. This involves understanding the runtime data model in ProViz and programming with the painter API.
6.2.3 Results and Observations

First of all, although all participants were asked to read the documentation beforehand, many of them had taken just a quick glance at it. It is understandable that without knowing certain key principles described in the documentation, these participants would spend time browsing and searching in the documentation during the test, and this certainly showed in the results. On the other hand, participants who took the time to comprehend the documentation were able to accomplish the tasks in a shorter amount of time. So even though all participants were using ProViz for the first time, we still had the distinction that some had already acquired necessary knowledge from the documentation and some had not.

Table 4 – Time (in minutes) for completing each task in the usability test. (Numbers marked with asterisks (*) are results without Task 3)

<table>
<thead>
<tr>
<th>Participant</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>15</td>
<td>14</td>
<td>7</td>
<td>6</td>
<td>7</td>
<td>15</td>
<td>10.67</td>
<td>4.41</td>
</tr>
<tr>
<td>Task 2</td>
<td>8</td>
<td>13</td>
<td>8</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6.83</td>
<td>3.66</td>
</tr>
<tr>
<td>Task 3</td>
<td>16</td>
<td>9</td>
<td>12</td>
<td>14</td>
<td></td>
<td></td>
<td>12.75*</td>
<td>2.99*</td>
</tr>
<tr>
<td>Task 4</td>
<td>17</td>
<td>20</td>
<td>15</td>
<td>16</td>
<td>10</td>
<td>13</td>
<td>15.17</td>
<td>3.43</td>
</tr>
<tr>
<td>Task 5</td>
<td>12</td>
<td>8</td>
<td>13</td>
<td>8</td>
<td>7</td>
<td>10</td>
<td>9.67</td>
<td>2.42</td>
</tr>
<tr>
<td>Task 6</td>
<td>15</td>
<td>16</td>
<td>13</td>
<td>11</td>
<td>11</td>
<td>9</td>
<td>12.5</td>
<td>2.66</td>
</tr>
<tr>
<td>Total</td>
<td>83</td>
<td>71</td>
<td>65</td>
<td>56</td>
<td>53</td>
<td>52</td>
<td>63.33</td>
<td></td>
</tr>
<tr>
<td>Total w/o Task 3</td>
<td>67</td>
<td>71</td>
<td>56</td>
<td>44</td>
<td>39</td>
<td>52</td>
<td>54.83</td>
<td></td>
</tr>
</tbody>
</table>

Table 5 – Difficulty of each task as rated by the participants. (Numbers marked with asterisks (*) are results without Task 3)

<table>
<thead>
<tr>
<th>Participant</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>Average</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task 1</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2.5</td>
<td>1.38</td>
</tr>
<tr>
<td>Task 2</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2.33</td>
<td>1.37</td>
</tr>
<tr>
<td>Task 3</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
<td>2.25*</td>
<td>0.96*</td>
</tr>
<tr>
<td>Task 4</td>
<td>1</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2.67</td>
<td>1.37</td>
</tr>
<tr>
<td>Task 5</td>
<td>2</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>2.33</td>
<td>1.21</td>
</tr>
<tr>
<td>Task 6</td>
<td>4</td>
<td>5</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>3.67</td>
<td>0.82</td>
</tr>
</tbody>
</table>

The time for completing each task is displayed in First of all, although all participants were asked to read the documentation beforehand, many of them had taken just a quick
glance at it. It is understandable that without knowing certain key principles described in the documentation, these participants would spend time browsing and searching in the documentation during the test, and this certainly showed in the results. On the other hand, participants who took the time to comprehend the documentation were able to accomplish the tasks in a shorter amount of time. So even though all participants were using ProViz for the first time, we still had the distinction that some had already acquired necessary knowledge from the documentation and some had not.

Table 4, ordered by the total time spent by each subject. After the test, the participants rated the difficulty of each task on the questionnaire; their ratings are displayed in Table 5.

**Task 1**

Task 1 comprised the most fundamental user scenario of ProViz – annotating a program with painters and following ProViz’s visualization process to visualize it. The task was to visualize an array instance variable in an array reversal program. Because the target is an instance variable, visualizing it is to follow the principle described in Section 5.1.1 (which is also in the online documentation) and annotate a local variable that refers to the object containing this array. Half of the participants were able to recognize this concept and annotated the local variables; others referred to the documentation after seeing a blank screen and were able to correct the mistake.

After annotating the program, the participants needed to run the visualization following the instructions described in the documentation. The researcher had also demonstrated this procedure at the very beginning of the test, and the checklist also helped remind them of any missed step in the process. Once the participants had run the visualization several times, they were able to get accustomed to the procedure.

1 http://www2.hawaii.edu/~johnwu/ProViz/ProViz_HowTo_2.html
Task 2

In Task 2, the participants needed to change the visualization of an array by changing its painter. This task was required to be performed in ProViz Editor without modifying the annotations in the source code. There were two solutions to this task. What most participants did was to directly change the painter of the array variable in ProViz Editor. But a more desirable method was to use the principle from Section 5.1.2 (which was also in the online documentation) and change the order of painters that were assigned to the array’s type. Two participants recalled this principle from the documentation and accomplished the task using the latter method.

Task 3

Four out of six participants performed this task, which required knowledge about (1) the structure of an array variable and its field variables and (2) accessing a variable from a painter through the getVariable() method. The time it took to accomplish this task was quite high because it required code reading first to understand how the painter works. Furthermore, the participants all took time to look for information about painters in the documentation. Overall, the task was very easy once they comprehended the program and knew how to access the variable of a painter.

Task 4

Task 4 proved to be the most laborious task, taking everyone the longest amount of time to complete. But the interesting fact is that participants found this task conceptually easy, as its difficulty rating is not far different from that of the other tasks despite the effort the participants spent on it. In reality, this task involved mostly graphical programming and, of all the tasks, was the one least concerned with ProViz’s functionality. Code reading was needed for the participants to comprehend how the original graphical programming was accomplished, and they needed to calculate the intermediate points for the animation path. The only part that involved ProViz’s API was the composition of intermediate animations.

It took the participants an average of 15 minutes to accomplish this task. A significant quantity of the time (four to six minutes observed) was spent in program
comprehension – just to understand the original programming. The laboriousness of this task and of Task 3 for the participants could indicate that graphical programming itself is a difficult task by nature, which reinforces the need of an effective framework to simplify the process of PV.

Task 5

In this task, the participants needed to implement an image painter and apply it to visualize the elements of an array, which was the underlying data type of a stack. An image painter simply needed the file path of an image to be specified, so the painter-making process was very straightforward and was quickly accomplished by all participants. Once they had created it, they needed to apply this image painter in order to visualize all the elements of the array by annotating the painter to the “[ ]” field of the array as described in the instructions and in the documentation. This task was also straightforward and was accomplished relatively quickly.

Task 6

Task 6 required the participants to traverse the runtime data and find the right variable for retrieving a painter. It called for knowledge about the program structure, the corresponding runtime structure, and more importantly, ProViz’s traversal methods. All participants accomplished this task by searching in the documentation for clues, copying and pasting relevant codes, and modifying them to fulfill the requirement.

This task was rated as the most difficult one. We believe this can be attributed to participants’ unfamiliarity with the painter API, as it was the first time they had learned and used it. Furthermore, the task did require understanding of the roles of the variables in the source code (e.g., instance or local variable) and their corresponding locations in the runtime data structure, and the participants could only acquire this knowledge from the documentation. Again, whether the participants had already comprehended the documentation made a difference in their performances.

6.2.4 Analyses

Several factors contribute to making the outcome of this usability test highly encouraging:
• It cannot be overemphasized that the test was performed by first-time users who had never used or operated ProViz before. Moreover, the tasks were not designed to be simple and effortless. On the contrary, they are realistic and practical scenarios for creating and customizing visualizations. Yet participants who had read the online documentation were able to exceed our expectations regarding their speed and performances on the tasks. Although it took them longer, participants who had not previously comprehended the documentation were able to use it to find the necessary clues during the test, and so were able to accomplish the tasks.

• In doing Task 1, the participants operated ProViz and learned about how to use it, including the annotations, for the first time. After acquiring the knowledge and experience that Task 1 provided, they were able to apply their newly learned skills in the subsequent tasks. Many of them had also realized that some steps in visualizing a program, such as placing the initial breakpoint and loading the annotations into ProViz Editor, needed to be performed only once. This indicates that once a user becomes more experienced with ProViz, she will be able to operate it at a faster pace and even with less effort by knowing what steps do not need to be repeated.

• Many participants had never seen a program actually running in front of their eyes and showing its exact behavior, and they made highly positive comments about its potential in education and in software development.

The usability test has also given us information about some issues with ProViz, in particular for novice users, that can be improved.

**Usability**

There was no serious usability issue reported in the experiment, but there were some sensible suggestions about making some processes simpler and also more intuitive to the user. First were the steps to run the visualization. Some participants suggested automatically loading annotations of a file to ProViz Editor, and some mentioned having a “do-it-all” button to load the annotations into ProViz Editor, place a breakpoint at the main method, and start the eclipse debugger. While these suggestions are reasonable,
ProViz Editor and the breakpoint are essentially tools for customization in ProViz that need to be provided separately. The editor is for composing Viz sets (Section 4.2.3), and the breakpoint serves as the starting point of the visualization. So for experienced users of ProViz, automations on these processes may not be desirable.

Another suggestion was to expand Viz annotations to include parameter passing capability so that information requested by painters can be specified directly in the annotation instead of creating new painters. This certainly is an intriguing feature that could increase usability, and Java annotations do support attribute specifications as parameters. Our only concern about over-utilizing the annotations is that if too much content were added to the annotations, it could clutter the source code and decrease the program’s readability. Therefore, instead of increasing the complexity of the annotations, either ProViz Editor can be made to provide more features, or as the future work section will discuss, user interface tools can be created to help the user compose painters.

**Annotating Instance Variables**

The most common mistake in Task 1 was, as expected, annotating the array instance variable but not the local variable that pointed to this array’s owner object. This reflects a misconception that general programmers have when they annotate an instance variable, not realizing that classes and instance variables are actually blueprints for defining instances of objects. Recall from the discussion in Section 5.3 that if an object were not to be visualized, i.e., no variable referencing it were annotated with a painter, then its fields would not be visualized even if they had been annotated.

However, because these misconceptions were observed to be the rule among the participants, this issue should be addressed. It is possible to modify ProViz to show the visualization of instance variables with or without its root variable being visualized. This calls for a special rule for instance variables that are under the “this” variable in a stack frame. Primarily, we think that the best solution is to implement VOO on such instance variables so that the resulting effects are as follows: (1) if an instance variable’s object has been visualized previously, no new painter will be created and this variable will
become an alias to the existing painter; (2) if this variable has not been visualized, a new painter will be created for it.

A possible problem with this approach is that ProViz, rather than the user, would be in control of what is visualized. If the user’s intention were to follow the correct principle ProViz has established, she would be forced to view instance variables that she did not wish to see. Eventually, this may be better resolved by making a new type of Viz annotation for instance variables that forces VOO on these variables, while other instance variables will still follow the general principle of ProViz.

**Understanding the Runtime Data**

Even though the annotations are applied on the static source code, the actual visualization of ProViz depicts the dynamic program data at runtime. In developing a painter, how this painter retrieves data or other painters is according to the runtime data structure described in Section 5.2.1. This test shows that general programmers may have insufficient knowledge about the runtime data. Therefore, in order to develop painters, a user may need to spend some time to learn about how ProViz models the runtime data and how painters are associated with them by reading the documentation.

**Conclusion**

The usability test was designed to test different aspects of ProViz using realistic scenarios to form the tasks. The outcome showed that even novice users were able to perform operations of ProViz and visualize programs. ProViz’s visual mapping mechanism, i.e., the annotations and ProViz Editor, requires a significantly lower learning curve than other data-driven systems, which require learning a different language in order to compose the visual mapping. This is made more evident by the fact that half of the participants were not Java programmers, meaning ProViz’s mapping mechanism is truly language-independent. Moreover, the participants saw different visualizations exhibited by different painters and also realized how easy it was to switch between one and another.

An undesirable factor in the test was that not all participants had read the documentation as requested, and that did affect their speed in completing the tasks. This
nonetheless showed the importance of reading a manual for first time users. Obviously, when a person gets familiar with the framework, she will be able to perform related tasks more easily and faster than the results of this usability test show. For instance, for an experienced ProViz user, Task 1 may take less than a minute to complete, and Task 2 can actually be achieved in a matter of seconds.
Chapter 7
Discussion and Conclusion

7.1 Challenges and Limitations

7.1.1 Speed and Efficiency

Size
Because of the effort needed to monitor and extract runtime data, the size of the collected
data is currently a limiting factor in the performance of ProViz. Large data structures,
such as arrays, collection types, and I/O objects, could all slow down or even cripple
ProViz. An update to remove the redundancy in the runtime data model (discussed in
Section 4.5.1) could improve efficiency to a certain degree because it will eliminate the
need to maintain duplicate structures in Viz Runtime.

Live Visualization
The overhead of ProViz’s step processing is quite large, typically taking dozens of
milliseconds for each step. As mentioned in Section 2.2.2, such a delay is negligible
when the PV is showed live with animations. However, it poses a problem when the
viewer wants to fast-forward the visualization, and the forwarding speed may not be as
fast as one would desire. Also, when the visualization is recorded without being shown
live, it may take a while just to run and record the program.

Post-Mortem
As ProViz can record program execution for future playback, it cannot avoid the
recording-size problem of post-mortem systems. But several aspects of ProViz can
mitigate the impact of these inefficiencies. First, ProViz updates and records only the top
stack frame at each step and not the entire runtime data. Second, when methods that are
irrelevant to the visualization are filtered out of Viz Map Model, their executions will not
be monitored or recorded, nor will any subsequent method calls originated from these
unwanted methods. Third, in contrast to recording low-level data, like BugNet
(Narayanasamy, Pokam, & Calder, 2005), ProViz records high-level data. As abstract
data contain less information, their volume is significantly smaller than that of low-level data.

**Scalability**

The scalability of ProViz is not confined by the number of stack frames in a program. In other words, the accumulation of a large number of method calls does not affect ProViz’s performance because Viz Runtime and VPM only update the top stack frame at every execution step, regardless of how many stack frames there are in a program. Therefore, although this has not yet been tested empirically, ProViz’s processing time for a program should be independent of the number of stack frames required by the program.

ProViz is, however, subject to slow down when it encounters large data structures. We currently filter out the creation and update for such data types, including some IO classes and `java.util.Scanner`, so that student programs using these classes can be visualized without delay. Further investigation into smart filtering mechanisms is necessary to avoid unnecessarily large data types or to update only relevant portions of relevant data.

### 7.1.2 Visualizing Multi-Threaded Programs

There is a profound problem inherent to high-level PV systems that offer animation: the visualization of a multi-threaded program is affected by the time spent in the animation. Ideally, a PV system should not change the target program’s behavior, and for single-thread programs, while the interruptions due to time-consuming animations at each step may slow down the program’s execution, they do not change the program’s behavior. For multi-threaded programs, however, stopping a thread for animation creates an imbalance between the speeds of the threads, which might in fact change the program’s behavior (Price, Baecker, & Small, 1998, p. 13). For example, assume a program has two threads: \( A \) and \( B \). At some point of the program, \( B \) is to receive a signal from \( A \) and if it did not for some time, \( B \) would move on to do something else. When visualized, \( A \) may be suspended during its animation; \( B \) is supposed to get a signal from \( A \) at a particular time,
but does not get it. Therefore, when visualizing multi-thread programs live, it is possible that the program’s behavior will change.

There is a way to visualize multi-threaded programs with relatively modest impact on their behaviors – the post-mortem feature. Instead of showing the visualization live, the execution data of multiple threads are recorded. The assumption is that the monitoring and recording time for each thread’s stream of data will not substantially affect the runtime of the different threads, and thus the recording can retain most of the program’s behavior even though its playback again faces the same problem as a live-PV system does. Nevertheless, more research on this topic will be necessary in the future.

7.1.3 Unsuitability for Time-Sensitive Programs

Although ProViz does not change the behavior of programs it visualizes, it does change the execution time of these programs. Therefore, if a program is time-sensitive, such as one that measures the runtime of some algorithms, then applying visualization is likely to yield unrealistic results. Currently, there is no way around this problem.

7.1.4 No Polymorphism for Interfaces

A Java interface is an abstract type that can be implemented by subtypes, and polymorphism allows a variable of an interface to reference a subclass object of this interface. However, because an interface defines behaviors and not data, not only is a painter visualizing this interface not guaranteed to visualize a subtype of this interface, but it is also unlikely to have a painter visualizing interfaces that rely on the subclasses to implement some form of data structure.

Take the Collection interface in Java as an example. Can there be a painter made to visualize the Collection interface, meaning that it is capable of visualizing all subtypes of Collection? The answer is no. Because subtypes of Collection use different data structures, such a painter must account for all these different data structures and so may be overly complex and unrealistic. Take two subtypes of Collection, ArrayList and LinkedList, as examples. ArrayList’s internal data structure is an array, and LinkedList uses circular, doubly linked nodes. Having
one painter that is capable of visualizing both of these data types, not to mention dozens of other subtypes of Collection, may not be practical. Therefore, painters cannot fulfill the polymorphism property of Java interfaces.

The reason that painters do not have the same polymorphism property for interfaces is the lack of data definition, and painters are data-driven. On the other hand, painters can satisfy polymorphism for abstract classes and regular class inheritance because data in super classes are inherited in subclasses. For example, suppose ArrayList had a subclass, a painter for ArrayList would be able to visualize this subclass because it inherits the underlying data structure from ArrayList.

### 7.2 Future Work

The prototype of ProViz was constructed to prove that the proposed strategies and visualization paradigms can achieve the goal of this study. The following subsections describe future opportunities for work that goes beyond the scope of the present study but would significantly improve the functionalities and applications of ProViz and enable its possible adoption by the educational and software developers’ communities.

#### 7.2.1 Video Documentation

ProViz can record visualizations and play them back, but if these recordings are just arbitrary files that are not related to the corresponding source programs, they can hardly qualify as documentation. To serve as documentation, a video must be associated with either the relevant parts of a program or it needs to be integrated with other documentation means.

To associate videos with a program, the rich environment of Eclipse again gives us an opportunity of implementing this as a plug-in. A conceptual design for such a tool would be as simple as inserting video links to the source code, and clicking on the link would pop up a Viz Player that plays the corresponding video that demonstrates the corresponding portion of code.
On the other hand, the concept of hypertextbook (Ross, 2008) integrates multimedia items into a web-based documentation in order to provide a richer, more dynamic environment for the reader. When technical documentations are made and distributed on the internet to describe a program, videos made for this program can be inserted into the documentation as applets or as compressed videos. As a result, the videos can demonstrate procedures and algorithms visually alongside the textual descriptions.

Additional benefits of video documentation can be derived from Mayer’s (2001, 2005) spatial contiguity principle, which postulates that “students learn better when corresponding words and pictures are presented near rather than far from each other on the page or screen” (Mayer, 2001, p. 81). Therefore, instead of having videos alongside some paragraphs of text in the documentation, we can use a video recording as the basis for a more comprehensive demonstration. Explanatory texts, captions, or audio narration can be included in the video. Such an enhanced video alone could be a complete learning environment by itself, but the fact that ProViz videos are interactive and customizable offers even more potential.

### 7.2.2 Tools to Facilitate Painter Creation

Currently, painters are Java classes, and making new painters requires programming. As the properties and characteristics of painters become normalized, user-interface tools can be developed and used in designing new painters. Furthermore, as learned from the usability test, certain painter-specific information may be passed to the painter via Viz annotations or ProViz Editor so that the user does not need to create several painters of the same type for the same program semantics under different contexts (most commonly, this refers to the same program structure or algorithm implemented in different programs with different variable names, such as a programming assignment implemented by a number of students).

### 7.2.3 Monitoring Static Variables

In Java, a class is able to directly access and modify static variables whose classes are accessible in this class, including classes in the same package and other classes that are
imported. However, these static variables will not appear in the debugger’s stack frames, and thus ProViz cannot monitor them effectively. Further research is needed to find out how the debugger (or even the virtual machine, if the debugger does not have such a mechanism) stores all the class variables.

### 7.2.4 VOO Implementation

As discussed in Section 3.6.2, implementing VOO requires more sophisticated painter management and user control in order to have a satisfying and effortless conflict resolution for the user. ProViz’s data structure is capable of handling both VOV and VOO paradigms. Therefore, once VOO is implemented, the user can mix the two paradigms in order to customize the visualization as she sees fit.

### 7.2.5 Graphical Event-Listener Model

Currently, the event-listener model has painters listening and reacting to variable/data changes in Viz Runtime. A further enhancement would be to create dependencies between painters themselves so that painters can react not only to data change events but also to graphical changes made by other painters. For instance, consider the visualization of an array and its indices. As seen in Figure 34, if the user moves the array using the mouse, ideally the index painter should move with the array and maintain its graphical relationship with this array.

![Figure 34](image.png)

(a)  
(b)  

**Figure 34** – Example of inconsistency without graphical event-listeners. In (b), the array painter has moved, but index painters, i and j, were not aware of the movement.
Currently, ProViz uses absolute coordinates to place all painters. While this is typically sufficient for smaller programs and gives complete freedom for user customization, developers of large programs with more painters will find it burdensome to place them manually (Myers, 1990). The design for a layout mechanism can start with the mechanics of a program, which can be learned from Jeliot 3. Jeliot 3 has four designated areas for displaying different program mechanics as shown in Figure 35. While ProViz should not impose a rigid layout like this, it can nevertheless supply designated areas for certain data types. For instance, it may be useful to organize views of method calls according to their stack frames. The mockup view in Figure 36 has the current executing method on the main view to the right of the screen. To the left are the thumbnails of previous method
calls. When a new method is called, the current method on the right pane will be shrunk to a thumbnail and will be pushed onto the method stack. Then the new method’s visualization will take place on the right pane. Once a method returns, it will be removed from the right pane, and the top method thumbnail in the method stack will be enlarged and popped back to the right pane.

Such a method-oriented view may be suitable for organizing large programs with many visual components, but it will likely face problems with depicting a large number of intermediate methods and aliased references and parameters among these method calls.

7.2.7 Static Content Support

There is a tendency to use the static content derived from the source code to support dynamic program visualization (Greevy, Lanza, & Wysseier, 2006). In the case of ProViz, static content like the method call graph can be used as a road map for visualization. This could replace the current Viz Map Model as the new visual mapping mechanism. A call graph lays out all of a program’s execution paths by traversing cascading method calls in a program. Therefore, it provides more accurate contextual information about the runtime data than the static definitions of classes and methods and allows greater customizability because the animator will be able to specify visual mapping based on a specific method in a possible call sequence. In addition, call graphs allow automation of filtering out methods that are irrelevant to the visualization, in contrast to the manual process used in ProViz now (as described in Section 4.3.1). This can be accomplished by removing branches that do not contain any painter annotations.

7.3 Conclusion

Dynamic program visualization is a technology with an intriguing past. Despite its uncontested potentials and benefits, it has never taken hold in standard programming practices. Research studies and tools in high-level PV have only had limited usage in academia and few such tools are in existence for the software engineering field. There is one simple explanation of this astonishing fact: creating such visualizations has been too laborious and thus impractical. ProViz addresses this fundamental problem of PV and
offers an opportunity that program developers have rarely experienced before – seeing a program running visually as they envision it.

The data-driven visualization technique has not been adopted by many DPV tools, and that means its potential is yet to be fully discovered. The characteristics of data-driven visualization make visualizers reusable and allow the separation of the visual functions from the data source. Moreover, the task of identifying interesting variables is substantially more feasible than identifying events in a program. Therefore, in contrast to the event-driven approach, data-driven visualization has a much better potential for simplifying the PV task.

Using data-driven design and the object-oriented aspect of painters, ProViz offers a high level of reusability because (1) a painter for a class can visualize any instance of this class, and (2) new painters can be built upon existing painters. Given ready-made painters for fundamental data types, the sample visualizations displayed in Appendix A were typically created in merely dozens of minutes. Such relatively manageable effort allowed us to create multiple abstract representations for the same program where each depiction conveys a certain level of detail and information for specific comprehension needs. This is particularly useful in a classroom situation where students can learn progressively from high-level concepts to low-level implementations. Furthermore, the mapping mechanism – together with the concept of Viz-sets – allows the user to seamlessly switch between different visualizations of a program. To our knowledge, such flexibility and customizability has yet to be seen in any other DPV tool.

As newly constructed painters expand and enrich the collection of painters, less and less effort will be needed for making new painters and visualizations. And as the visualization process becomes more and more effortless, we hope that PV will eventually fulfill its promise and greatly enhance program comprehension tasks and visual debugging. Furthermore, the concept of using videos to document software programs, while now only proposed in literature and virtually non-existent in practice, is poised to become an invaluable complement to existing documentation means.
The prototype of ProViz created in this study has functioned as expected. The usability test showed that general programmers indeed can learn to use the framework quickly. Last but not least, the opportunity of seeing a program running as truly envisioned by programmers has drawn very positive reactions from people who have seen ProViz in action.
Appendices

Appendix A – Examples of ProViz Visualizations

(a) Stack view

(b) Transparent tree view

(c) Tree view with counting the number of method calls. Animation moves each method call to the stacks below and increases the counters (the dashed-arrow is added manually to indicate movement).

(d) Binary tree view of the entire calling sequence. The colors are used to indicate the value for the parameter, n.
(e) Visualizations for a stack in different abstractions. (i) has the highest abstraction, showing a stack of plates; (ii) displays the values in the stack; (iii) is the underlying data structure of the stack (an array and an index variable). In (iii) the user can switch on/off the color that shows the portion of the array the stack occupies.

(f) Recursive binomial coefficient calculation; green signifies the left recursion and red signifies right

(g) A circular linked list

(h) Queue in two different abstractions. (i) shows a normal queue representation; (ii) displays the underlying data structure for the queue

(i) The multiplication of two fractional numbers: \( x = \frac{2}{3} \times \frac{5}{6} \). The middle frame uses animation to move y’s numbers over to x for multiplication.

(j) The sum of two fractional numbers: \( x = x + y \). The numerators were calculated first and then the denominator.
Appendix B – Usability Test Form

Usability Test for Evaluating the ProViz Framework

Please write down any question or problem you encounter during each task in the space provided below. Thank you!

Task 1 – Visualize a program by annotating the necessary elements of a program

Given an array reverse program and a list of painters, please annotate in the program source code or in Viz Set Editor with the right painters so that the visualization can display the array and the swapping animations.

Questions/Comments: …………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………

Task 2 – Modify the visualization by changing the visual mapping

The objective is to change an array representation to a bar-graph representation. A visualization of insertion sort uses an int[] array, which by default is visualized with an ArrayPainter. Your task is to change the visualization to a bar graph using ChartArrayPainter.

Q/C: ………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………
…………………………………………………………………………………………………………

(a) Original

(b) Target
Task 3 – Let image positions (heights) correspond to values in bubble sort

The classical visualization for bubble sort correspond values to positions of the elements. Your task is to change the original visualization to the target visualization.

Q/C:  

(a) Original  

(b) Target

Task 4 – Modify simple animations

A Towers of Hanoi program is visualized with the animation incorrectly moving rings from pole to pole in direct path. Please change the animation into a sequence of motions which correctly display the physics of moving the rings between poles.

Q/C:  

(a) Original  

(b) Target
Task 5 – Change certain graphical representations to images

Change the visualization of a stack from a list of numbers to images of plates by implementing a subclass of `viz.painters.lib.ImagePainter`. The image file is located at “C:/plate.jpg”. You may scale the image to size 80x30 in the image painter.

Q/C: ………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………

Task 6 – Navigate the runtime model and painters

Given an array sorting program, please create a painter that shows the indices of the array and is named `MyIndexPainter`, as specified in the pre-annotated source code. This painter should inherit from `viz.painters.java.array.IndexPainter`.

Q/C: ………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
………………………………………………………………………………………………
…………………………………………………………………………………………
Appendix C – Usability Test Questionnaire

**General Questions**
1. Are you currently a student? □ Yes □ No

2. Are you working in the software industry or in academia? □ Industry □ Academia □ Neither

3. What is your programming experience? □ < 2 years □ 2-5 years □ 5-10 years □ 10-20 years □ > 20 years

4. What is your primary programming language? __________________________

5. What is your familiarity with the object-oriented programming paradigm?  
<table>
<thead>
<tr>
<th>Basic</th>
<th>Moderate</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

6. How would you rate your Java programming skill?  
<table>
<thead>
<tr>
<th>Basic</th>
<th>Moderate</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

7. How would you rate your knowledge about the structure of Java runtime data, including threads, stack frames, and variables?  
<table>
<thead>
<tr>
<th>Basic</th>
<th>Moderate</th>
<th>Advanced</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

8. How familiar are you with the Eclipse IDE?  
<table>
<thead>
<tr>
<th>Not at all</th>
<th>Moderate</th>
<th>Very Familiar</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

**Test-Related Questions**

1. Is the visual mapping between annotations and painters easy to apply/understand?  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

2. Please rate the difficulty of each task
   
   **Task One:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Task Two:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Task Three:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Task Four:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Task Five:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

   **Task Six:**  
<table>
<thead>
<tr>
<th>Easy</th>
<th>Moderate</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
Bibliography


